# NOISeq: Differential Expression in RNA-seq

Sonia Tarazona (`starazona@cipf.es`)
Pedro Furió-Tarí (`pfurio@cipf.es`)
María José Nueda (`mj.nueda@ua.es`)
Alberto Ferrer (`aferrer@eio.upv.es`)
Ana Conesa (`aconesa@cipf.es`)

11 February 2016
(Version 2.14.1)

## Contents

# 1 Introduction

This document will guide you through to the use of the R Bioconductor package NOISeq, for analyzing count data coming from next generation sequencing technologies. NOISeq package consists of three modules: (1) Quality control of count data; (2) Normalization and low-count filtering; and (3) Differential expression analysis.

First, we describe the input data format. Next, we illustrate the utilities to explore the quality of the count data: saturation, biases, contamination, etc. and show the normalization, filtering and batch correction methods included in the package. Finally, we explain how to compute differential expression between two experimental conditions. The differential expression method NOISeq and some of the plots included in the package were displayed in [1, 2].The new version of NOISeq for biological replicates (NOISeqBIO) is also implemented in the package.

The NOISeq and NOISeqBIO methods are data-adaptive and nonparametric. Therefore, no distributional assumptions need to be done for the data and differential expression analysis may be carried on for both raw counts or previously normalized or transformed datasets.

We will use the "reduced" Marioni's dataset [3] as an example throughout this document. In Marioni's experiment, human kidney and liver RNA-seq samples were sequenced. There are 5 technical replicates per tissue, and samples were sequenced in two different runs. We selected chromosomes I to IV from the original data and removed genes with 0 counts in all samples and with no length information available. Note that this reduced dataset is only used to decrease the computing time while testing the examples. We strongly recommend to use the whole set of features (e.g. the whole genome) in real analysis.

The example dataset can be obtained by typing:

```
> library(NOISeq)
> data(Marioni)
```

# 2 Input data

NOISeq requires two pieces of information to work that must be provided to the readData function: the expression data (data) and the factors defining the experimental groups to be studied or compared (factors). However, in order to perform the quality control of the data or normalize them, other additional annotations need to be provided such as the feature length, the GC content, the biological classification of the features (e.g. Ensembl biotypes), or the chromosome position of each feature.

## 2.1 Expression data

The expression data must be provided in a matrix or a data.frame R object, having as many rows as the number of features to be studied and as many columns as the number of samples in the experiment. The following example shows part of the count data for Marioni's dataset:

```
> head(mycounts)
```

|  | R1L1Kidney | R1L2Liver | R1L3Kidney | R1L4Liver | R1L6Liver | R1L7Kidney | R1L8Liver |
|---|---|---|---|---|---|---|---|
| ENSG00000177757 | 2 | 1 | 0 | 0 | 1 | 2 | 0 |
| ENSG00000187634 | 49 | 27 | 43 | 34 | 23 | 41 | 35 |
| ENSG00000188976 | 73 | 34 | 77 | 56 | 45 | 68 | 55 |
| ENSG00000187961 | 15 | 8 | 15 | 13 | 11 | 13 | 12 |
| ENSG00000187583 | 1 | 0 | 1 | 1 | 0 | 3 | 0 |
| ENSG00000187642 | 4 | 0 | 5 | 0 | 2 | 12 | 1 |
|  | R2L2Kidney | R2L3Liver | R2L6Kidney |  |  |  |  |
| ENSG00000177757 | 1 | 1 | 3 |  |  |  |  |
| ENSG00000187634 | 42 | 25 | 47 |  |  |  |  |
| ENSG00000188976 | 70 | 42 | 82 |  |  |  |  |
| ENSG00000187961 | 12 | 20 | 15 |  |  |  |  |
| ENSG00000187583 | 0 | 2 | 3 |  |  |  |  |
| ENSG00000187642 | 9 | 4 | 9 |  |  |  |  |

The expression data can be both read counts or normalized expression data such as RPKM values, and also any other normalized expression values.

## 2.2 Factors

Factors are the variables indicating the experimental group for each sample. They must be given to the `readData` function in a data frame object. This data frame must have as many rows as samples (columns in data object) and as many columns or factors as different sample annotations the user wants to use. For instance, in Marioni's data, we have the factor "Tissue", but we can also define another factors ("Run" or "TissueRun"). The levels of the factor "Tissue" are "Kidney" and "Liver". The factor "Run" has two levels: "R1" and "R2". The factor "TissueRun" combines the sequencing run with the tissue and hence has four levels: "Kidney_1", "Liver_1", "Kidney_2" and "Liver_2".

Be careful here, the order of the elements of the factor must coincide with the order of the samples (columns) in the expression data file provided.

```
> myfactors = data.frame(Tissue = c("Kidney", "Liver", "Kidney", "Liver",
+     "Liver", "Kidney", "Liver", "Kidney", "Liver", "Kidney"), TissueRun = c("Kidney_1",
+     "Liver_1", "Kidney_1", "Liver_1", "Liver_1", "Kidney_1", "Liver_1",
+     "Kidney_2", "Liver_2", "Kidney_2"), Run = c(rep("R1", 7), rep("R2",
+     3)))
> myfactors

    Tissue TissueRun Run
1   Kidney  Kidney_1  R1
2    Liver   Liver_1  R1
3   Kidney  Kidney_1  R1
4    Liver   Liver_1  R1
5    Liver   Liver_1  R1
6   Kidney  Kidney_1  R1
7    Liver   Liver_1  R1
8   Kidney  Kidney_2  R2
9    Liver   Liver_2  R2
10  Kidney  Kidney_2  R2
```

## 2.3 Additional biological annotation

Some of the exploratory plots in NOISeq package require additional biological information such as feature length, GC content, biological classification of features, or chromosome position. You need to provide at least part of this information if you want to either generate the corresponding plots or apply a normalization method that corrects by length.

The following code show how the R objects containing such information should look like:

```
> head(mylength)

ENSG00000177757 ENSG00000187634 ENSG00000188976 ENSG00000187961 ENSG00000187583
           2464            4985            3870            4964            8507
ENSG00000187642
           6890

> head(mygc)

ENSG00000177757 ENSG00000187634 ENSG00000188976 ENSG00000187961 ENSG00000187583
           48.6            66.0            59.5            67.9            62.6
ENSG00000187642
           67.7

> head(mybiotypes)

 ENSG00000177757  ENSG00000187634  ENSG00000188976  ENSG00000187961  ENSG00000187583
      "lincRNA" "protein_coding" "protein_coding" "protein_coding" "protein_coding"
 ENSG00000187642
"protein_coding"

> head(mychroms)
```

```
               Chr GeneStart  GeneEnd
ENSG00000177757   1    742614   745077
ENSG00000187634   1    850393   869824
ENSG00000188976   1    869459   884494
ENSG00000187961   1    885830   890958
ENSG00000187583   1    891740   900339
ENSG00000187642   1    900447   907336
```

Please note, that these objects might contain a different number of features and in different order than the expression data. However, it is important to specify the names or IDs of the features in each case so the package can properly match all this information. The length, GC content or biological groups (e.g. biotypes), could be vectors, matrices or data.frames. If they are vectors, the names of the vector must be the feature names or IDs. If they are matrices or data.frame objects, the feature names or IDs must be in the row names of the object. The same applies for chromosome position, which is also a matrix or data.frame.

Ensembl Biomart data base provides these annotations for a wide range of species: biotypes (the biological classification of the features), GC content, or chromosome position. The latter can be used to estimate the length of the feature. However, it is more accurate computing the length from the GTF or GFF annotation file so the introns are not considered.

## 2.4 Converting data into a NOISeq object

Once we have created in R the count data matrix, the data frame for the factors and the biological annotation objects (if needed), we have to pack all this information into a NOISeq object by using the `readData` function. An example on how it works is shown below:

```
> mydata <- readData(data = mycounts, length = mylength, gc = mygc, biotype = mybiotypes,
+       chromosome = mychroms, factors = myfactors)
> mydata

ExpressionSet (storageMode: lockedEnvironment)
assayData: 5088 features, 10 samples
  element names: exprs
protocolData: none
phenoData
  sampleNames: R1L1Kidney R1L2Liver ... R2L6Kidney (10 total)
  varLabels: Tissue TissueRun Run
  varMetadata: labelDescription
featureData
  featureNames: ENSG00000177757 ENSG00000187634 ... ENSG00000201145 (5088 total)
  fvarLabels: Length GC ... GeneEnd (6 total)
  fvarMetadata: labelDescription
experimentData: use 'experimentData(object)'
Annotation:
```

The `readData` function returns an object of *Biobase's eSet* class. To see which information is included in this object, type for instance:

```
> str(mydata)
> head(assayData(mydata)$exprs)
> head(pData(mydata))
> head(featureData(mydata)@data)
```

Note that the features to be used by all the methods in the package will be those in the data expression object. If any of this features has not been included in the additional biological annotation (when provided), the corresponding value will be NA.

It is possible to add information to an existing object. For instance, `noiseq` function accepts objects generated while using other packages such as `DESeq` package. In that case, annotations may not be included in the object. The `addData` function allows the user to add annotation data to the object. For instance, if you generated the data object like this:

```
> mydata <- readData(data = mycounts, chromosome = mychroms, factors = myfactors)
```

And now you want to include the length and the biotypes, you have to use the `addData` function:

```
> mydata <- addData(mydata, length = mylength, biotype = mybiotypes, gc = mygc)
```

**IMPORTANT**: Some packages such as *ShortRead* also use the `readData` function but with different input object and parameters. Therefore, some incompatibilities may occur that cause errors. To avoid this problem when loading simultaneously packages with functions with the same name but different use, the following command can be used: `NOISeq::readData` instead of simply `readData`.

# 3  Quality control of count data

Data processing and sequencing experiment design in RNA-seq are not straightforward. From the biological samples to the expression quantification, there are many steps in which errors may be produced, despite of the many procedures developed to reduce noise at each one of these steps and to control the quality of the generated data. Therefore, once the expression levels (read counts) have been obtained, it is absolutely necessary to be able to detect potential biases or contamination before proceeding with further analysis (e.g. differential expression). The technology biases, such as the transcript length, GC content, PCR artifacts, uneven transcript read coverage, contamination by off-target transcripts or big differences in transcript distributions, are factors that interfere in the linear relationship between transcript abundance and the number of mapped reads at a gene locus (counts).

In this section, we present a set of plots to explore the count data that may be helpful to detect these potential biases so an appropriate normalization procedure can be chosen. For instance, these plots will be useful for seeing which kind of features (e.g. genes) are being detected in our RNA-seq samples and with how many counts, which technical biases are present, etc.

As it will be seen at the end of this section, it is also possible to generate a report in a PDF file including all these exploratory plots for the comparison of two samples or two experimental conditions.

## 3.1  Generating data for exploratory plots

There are several types of exploratory plots that can be obtained. They will be described in detail in the following sections. To generate any of these plots, first of all, `dat` function must be applied on the input data (NOISeq object) to obtain the information to be plotted. The user must specify the type of plot the data are to be computed for (argument `type`). Once the data for the plot have been generated with `dat` function, the plot will be drawn with the *explo.plot* function. Therefore, for the quality control plots, we will always proceed like in the following example:

```
> myexplodata <- dat(mydata, type = "biodetection")
```

```
Biotypes detection is to be computed for:
 [1] "R1L1Kidney" "R1L2Liver"  "R1L3Kidney" "R1L4Liver"  "R1L6Liver"  "R1L7Kidney" "R1L8Liver"
 [8] "R2L2Kidney" "R2L3Liver"  "R2L6Kidney"
```

```
> explo.plot(myexplodata, plottype = "persample")
```

To save the data in a user-friendly format, the `dat2save` function can be used:

```
> mynicedata <- dat2save(myexplodata)
```

We have grouped the exploratory plots in three categories according to the different questions that may arise during the quality control of the expression data:

- **Biotype detection**: Which kind of features are being detected? Is there any abnormal contamination in the data? Did I choose an appropriate protocol?

- **Sequencing depth & Expression Quantification**: Would it be better to increase the sequencing depth to detect more features? Are there too many features with low counts? Are the samples very different regarding the expression quantification?

- **Sequencing bias detection**: Should the expression values be corrected for the length or the GC content bias? Should a normalization procedure be applied to account for the differences among RNA composition among samples?

- **Batch effect exploration**: Are the samples clustered in concordance with the experimental design or with the batch in which they were processed?

## 3.2 Biotype detection

When a biological classification of the features is provided (e.g. Ensembl biotypes), the following plots are useful to see which kind of features are being detected. For instance, in RNA-seq, it is expected that most of the genes will be protein-coding so detecting an enrichment in the sample of any other biotype could point to a potential contamination or at least provide information on the sample composition to take decision on the type of analysis to be performed.

### 3.2.1 Biodetection plot

The example below shows how to use the `dat` and `explo.plot` functions to generate the data to be plotted and to draw a biodetection plot per sample.

```
> mybiodetection <- dat(mydata, k = 0, type = "biodetection", factor = NULL)

Biotypes detection is to be computed for:
 [1] "R1L1Kidney" "R1L2Liver"  "R1L3Kidney" "R1L4Liver"  "R1L6Liver"  "R1L7Kidney" "R1L8Liver"
 [8] "R2L2Kidney" "R2L3Liver"  "R2L6Kidney"

> par(mfrow = c(1, 2))
> explo.plot(mybiodetection, samples = c(1, 2), plottype = "persample")
```

Fig. 1 shows the "biodetection" plot per sample. The gray bar corresponds to the percentage of each biotype in the genome (i.e. in the whole set of features provided), the stripped color bar is the proportion detected in our sample (with number of counts higher than k), and the solid color bar is the percentage of each biotype within the sample. The vertical green line separates the most abundant biotypes (in the left-hand side, corresponding to the left axis scale) from the rest (in the right-hand side, corresponding to the right axis scale).

When `factor=NULL`, the data for the plot are computed separately for each sample. If `factor` is a string indicating the name of one of the columns in the factor object, the samples are aggregated within each of these experimental conditions and the data for the plot are computed per condition. In this example, samples in columns 1 and 2 from expression data are plotted and the features (genes) are considered to be detected if having a number of counts higher than k=0.
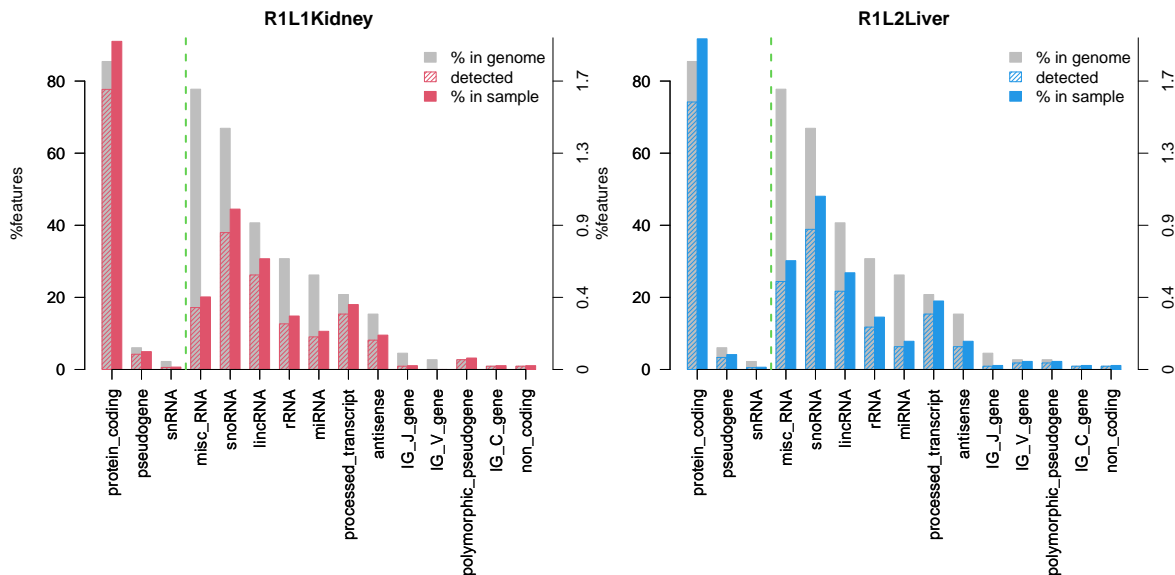


Figure 1: Biodetection plot (per sample)

When two samples or conditions are to be compared, it can be more practical to represent both o them in the same plot. Then, two different plots can be generated: one representing the percentage of each biotype in the genome being detected in the sample, and other representing the relative abundance of each biotype within the sample. The following code can be used to obtain such plots:

```
> par(mfrow = c(1, 2))
> explo.plot(mybiodetection, samples = c(1, 2), toplot = "protein_coding",
+     plottype = "comparison")
```

6

```
[1] "Percentage of protein_coding biotype in each sample:"
R1L1Kidney  R1L2Liver
      91.0        91.7
[1] "Confidence interval at 95% for the difference of percentages: R1L1Kidney - R1L2Liver"
[1] -1.799  0.423
[1] "The percentage of this biotype is NOT significantly different for these two samples (p-value = 0.23
```
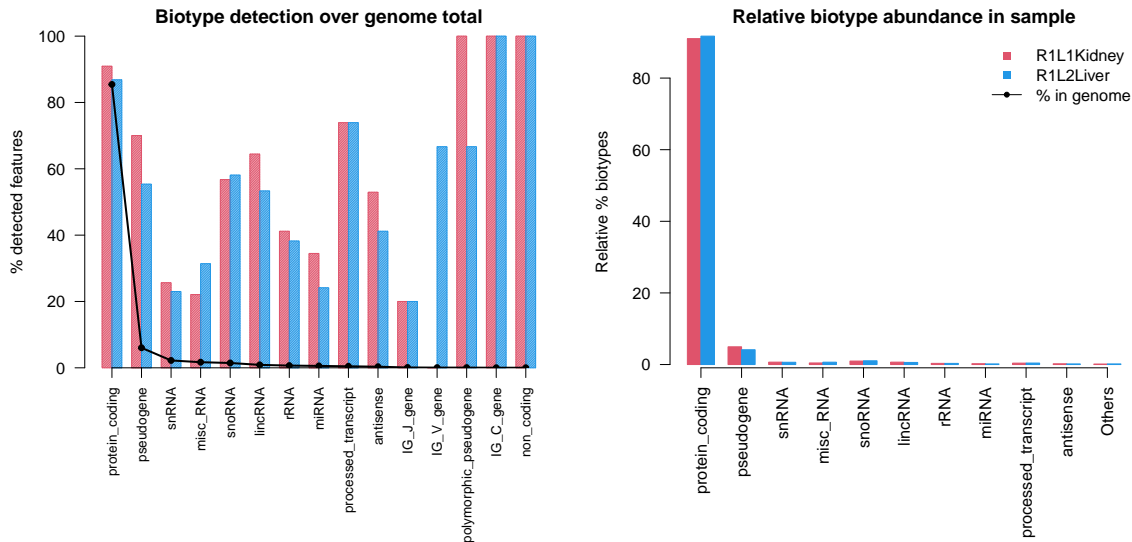


Figure 2: Biodetection plot (comparison of two samples)

In addition, the "biotype comparison" plot also performs a proportion test for the chosen biotype (argument `toplot`) to test if the relative abundance of that biotype is different in the two samples or conditions compared.

### 3.2.2   Count distribution per biotype

The "countsbio" plot (Fig. 3) per biotype allows to see how the counts are distributed within each biological group. In the upper side of the plot, the number of detected features that will be represented in the boxplots is displayed. The values used for the boxplots are either the counts per million (if `norm = FALSE`) or the values provided by the use (if `norm = TRUE`) The following code was used to draw the figure. Again, data are computed per sample because no factor was specified (`factor=NULL`). To obtain this plot using the *explo.plot* function and the "countsbio" data, we have to indicate the "boxplot" type in the `plottype` argument, choose only one of the samples (`samples = 1`, in this case), and all the biotypes (by setting `toplot` parameter to 1 or "global").

```
> mycountsbio = dat(mydata, factor = NULL, type = "countsbio")

[1] "Count distributions are to be computed for:"
 [1] "R1L1Kidney" "R1L2Liver"  "R1L3Kidney" "R1L4Liver"  "R1L6Liver"  "R1L7Kidney" "R1L8Liver"
 [8] "R2L2Kidney" "R2L3Liver"  "R2L6Kidney"

> explo.plot(mycountsbio, toplot = 1, samples = 1, plottype = "boxplot")
```

## 3.3   Sequencing depth & Expression Quantification

The plots in this section can be generated by only providing the expression data, since no other biological information is required. Their purpose is to assess if the sequencing depth of the samples is enough to detect the features of interest and to get a good quantification of their expression.

### 3.3.1   Saturation plot

The "Saturation" plot shows the number of features in the genome detected with more than `k` counts with the sequencing depth of the sample, and with higher and lower simulated sequencing depths. This plot can be
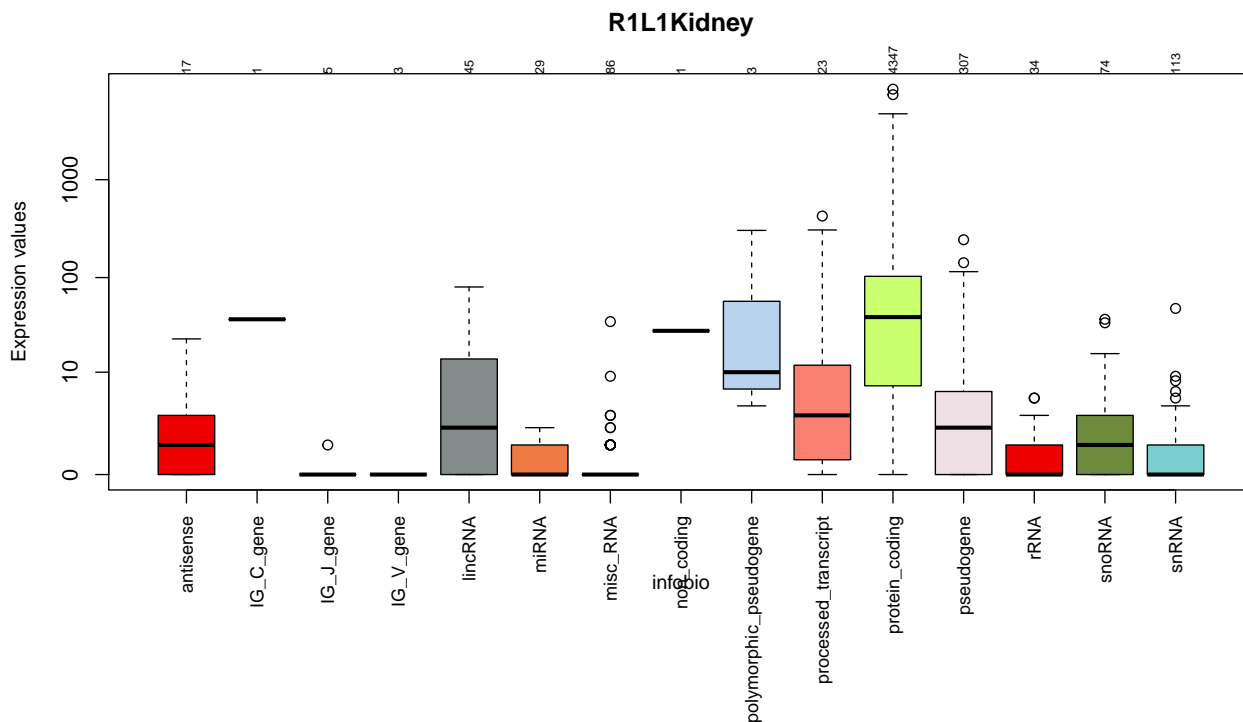
Figure 3: Count distribution per biotype in one of the samples (for genes with more than 0 counts). At the upper part of the plot, the number of detected features within each biotype group is displayed.

generated by considering either all the features or only the features included in a given biological group (biotype), if this information is available. First, we have to generate the saturation data with the function `dat` and then we can use the resulting object to obtain, for instance, the plots in Fig. 4 and 5 by applying `explo.plot` function. The lines show how the number of detected features increases with depth. When the number of samples to plot is 1 or 2, bars indicating the number of new features detected when increasing the sequencing depth in one million of reads are also drawn. In that case, lines values are to be read in the left Y axis and bar values in the right Y axis. If more than 2 samples are to be plotted, it is difficult to visualize the "newdetection bars", so only the lines are shown in the plot.

```
> mysaturation = dat(mydata, k = 0, ndepth = 7, type = "saturation")
> explo.plot(mysaturation, toplot = 1, samples = 1:2, yleftlim = NULL, yrightlim = NULL)

> explo.plot(mysaturation, toplot = "protein_coding", samples = 1:4)
```

The plot in Fig. 4 has been computed for all the features (without specifying a biotype) and for two of the samples. Left Y axis shows the number of detected genes with more than 0 counts at each sequencing depth, represented by the lines. The solid point in each line corresponds to the real available sequencing depth. The other sequencing depths are simulated from this total sequencing depth. The bars are associated to the right Y axis and show the number of new features detected per million of new sequenced reads at each sequencing depth. The legend in the gray box also indicates the percentage of total features detected with more than $k = 0$ counts at the real sequencing depth.

Up to twelve samples can be displayed in this plot. In Fig. 5, four samples are compared and we can see, for instance, that in kidney samples the number of detected features is higher than in liver samples.

### 3.3.2 Count distribution per sample

It is also interesting to visualize the count distribution for all the samples, either for all the features or for the features belonging to a certain biological group (biotype). Fig. 6 shows this information for the biotype "protein_coding", which can be generated with the following code on the "countsbio" object obtained in the previous section by setting the `samples` parameter to `NULL`.

```
> explo.plot(mycountsbio, toplot = "protein_coding", samples = NULL, plottype = "boxplot")
```
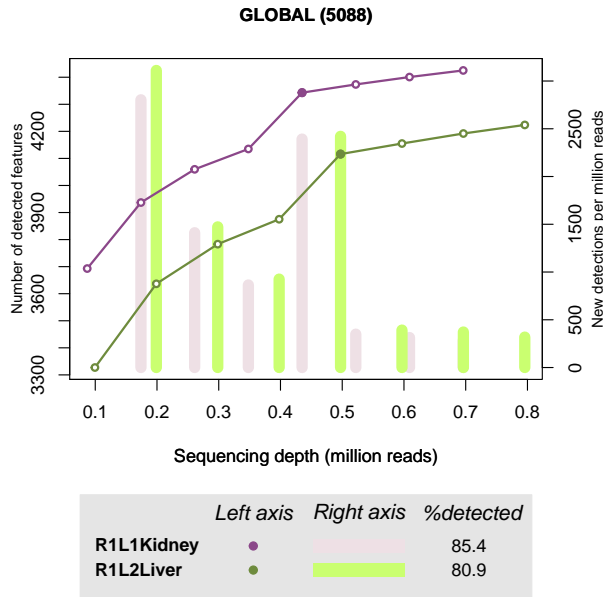
8

**GLOBAL (5088)**



Figure 4: Global saturation plot to compare two samples of kidney and liver, respectively.
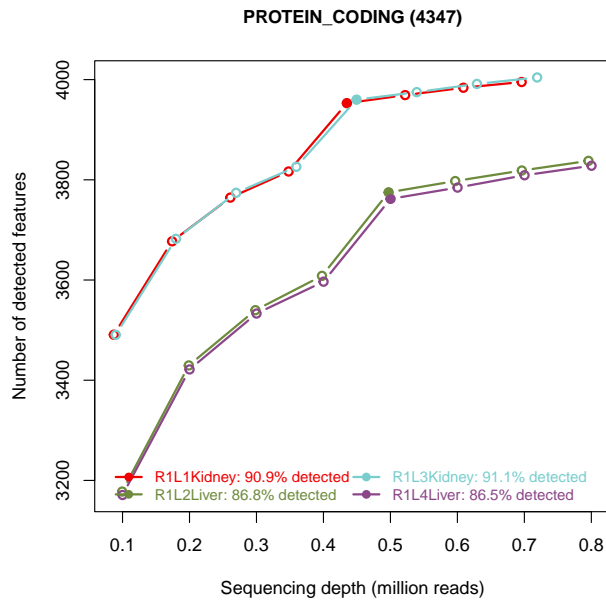
**PROTEIN_CODING (4347)**



Figure 5: Saturation plot for protein-coding genes to compare 4 samples: 2 of kidney and 2 of liver.

### 3.3.3 Sensitivity plot

Features with low counts are, in general, less reliable and may introduce noise in the data that makes more difficult to extract the relevant information, for instance, the differentially expressed features. We have implemented some methods in the `NOISeq` package to filter out these low count features. The "Sensitivity plot" in Fig. 7 helps to decide the threshold to remove low-count features by indicating the proportion of such features that are present in our data. In this plot, the bars show the percentage of features within each sample having more than 0 counts per million (CPM), or more than 1, 2, 5 and 10 CPM. The horizontal lines are the corresponding percentage of features with those CPM in at least one of the samples (or experimental conditions if the `factor` parameter is not `NULL`). In the upper side of the plot, the sequencing depth of each sample (in million reads) is given. The following code can be used for drawing this figure.

```
> explo.plot(mycountsbio, toplot = 1, samples = NULL, plottype = "barplot")
```
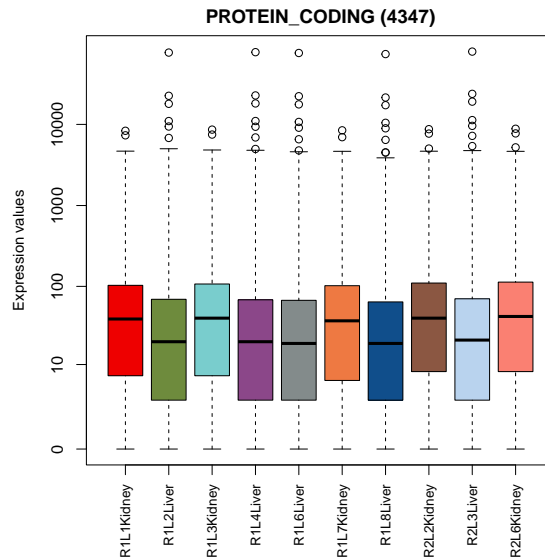
9

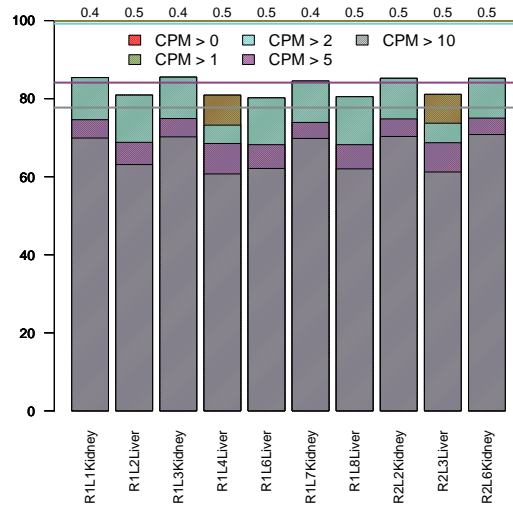Figure 6: Distribution of counts for protein coding genes in all samples.



Figure 7: Number of features with low counts for each sample.

## 3.4 Sequencing bias detection

Prior to perform further analyses such as differential expression, it is essential to normalize data to make the samples comparable and remove the effect of technical biases from the expression estimation. The plots presented in this section are very useful for detecting the possible biases in the data. In particular, the biases that can be studied are: the feature length effect, the GC content and the differences in RNA composition. In addition, these are diagnostic plots, which means that they are not only descriptive but an statistical test is also conducted to help the user to decide whether the bias is present and the data needs normalization.

### 3.4.1 Length bias

The "lengthbias" plot describes the relationship between the feature length and the expression values. Hence, the feature length must be included in the input object created using the `readData` function. The data for this plot is generated as follows. The length is divided in intervals (bins) containing 200 features and the middle point of each bin is depicted in X axis. For each bin, the 5% trimmed mean of the corresponding expression values (CPM if `norm=FALSE` or values provided if `norm=TRUE`) is computed and depicted in Y axis. If the number of samples or conditions to appear in the plot is 2 or less and no biotype is specified (toplot = "global"), a

diagnostic test is provided. A cubic spline regression model is fitted to explain the relationship between length and expression. Both the model p-value and the coefficient of determination (R2) are shown in the plot as well as the fitted regression curve. If the model p-value is significant and R2 value is high (more than 70%), the expression depends on the feature length and the curve shows the type of dependence.

Fig. 8 shows an example of this plot. In this case, the "lengthbias" data were generated for each condition (kidney and liver) using the argument `factor`.

```
> mylengthbias = dat(mydata, factor = "Tissue", type = "lengthbias")
> explo.plot(mylengthbias, samples = NULL, toplot = "global")
```
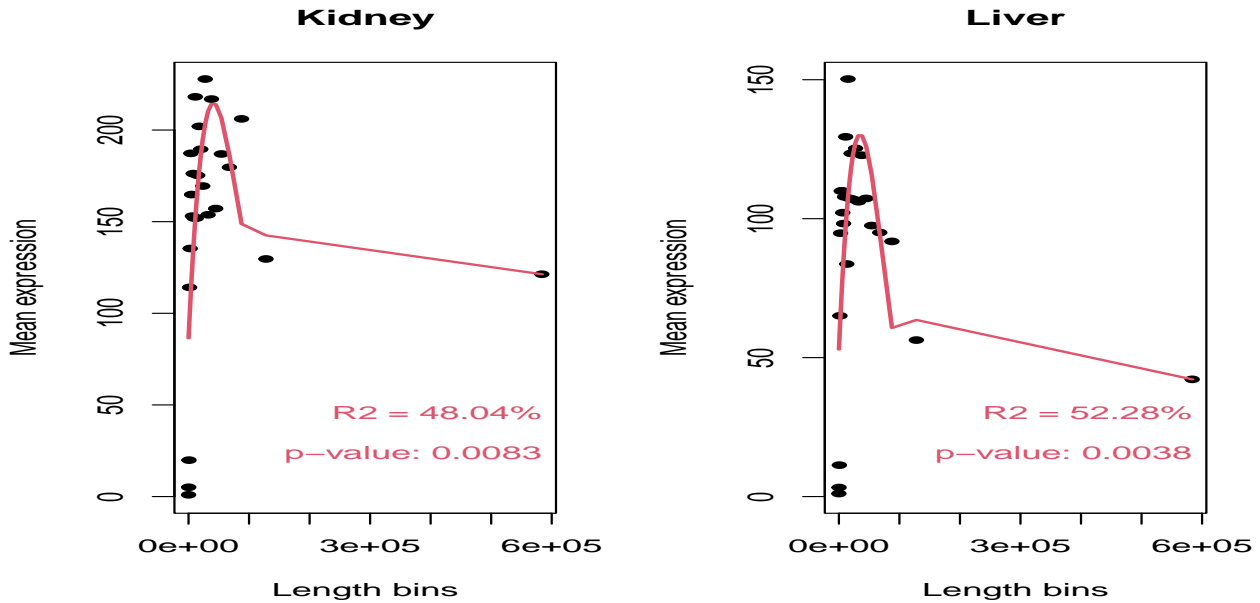


Figure 8: Gene length versus expression.

More details about the fitted spline regression models can be obtained by using the `show` function as per below:

```
> show(mylengthbias)

[1] "Kidney"

Call:
lm(formula = datos[, i] ~ bx)

Residuals:
   Min      1Q Median     3Q     Max
-85.40 -19.60   3.05  25.85  74.83


Coefficients: (1 not defined because of singularities)
            Estimate Std. Error t value Pr(>|t|)
(Intercept)    121.3       48.6    2.49   0.0215 *
bx1            -35.0       52.2   -0.67   0.5108
bx2            269.2       88.4    3.05   0.0064 **
bx3          -1301.1      719.7   -1.81   0.0857 .
bx4           6292.4     4655.4    1.35   0.1916
bx5              NA         NA      NA       NA
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 48.6 on 20 degrees of freedom
Multiple R-squared:  0.48,         Adjusted R-squared:  0.376
```

```
F-statistic: 4.62 on 4 and 20 DF,  p-value: 0.00833


[1] "Liver"

Call:
lm(formula = datos[, i] ~ bx)

Residuals:
   Min     1Q Median     3Q    Max
 -51.8  -18.2    0.0   21.3   42.3


Coefficients: (1 not defined because of singularities)
            Estimate Std. Error t value Pr(>|t|)
(Intercept)     42.2       29.7    1.42  0.17073
bx1             10.7       31.9    0.34  0.73952
bx2            222.1       54.0    4.12  0.00054 ***
bx3          -1141.7      439.3   -2.60  0.01716 *
bx4           5758.1     2841.2    2.03  0.05624 .
bx5               NA         NA      NA       NA
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 29.7 on 20 degrees of freedom
Multiple R-squared:  0.523,        Adjusted R-squared:  0.427
F-statistic: 5.48 on 4 and 20 DF,  p-value: 0.00382
```

### 3.4.2  GC content bias

The "GCbias" plot describes the relationship between the feature GC content and the expression values. Hence, the feature GC content must be included in the input object created using the `readData` function. The data for this plot is generated in an analogous way to the "lengthbias" data. The GC content is divided in intervals (bins) containing 200 features. The middle point of each bin is depicted in X axis. For each bin, the 5% trimmed mean of the corresponding expression values is computed and depicted in Y axis. If the number of samples or conditions to appear in the plot is 2 or less and no biotype is specified (toplot = "global"), a diagnostic test is provided. A cubic spline regression model is fitted to explain the relationship between GC content and expression. Both the model p-value and the coefficient of determination (R2) are shown in the plot as well as the fitted regression curve. If the model p-value is significant and R2 value is high (more than 70%), the expression will depend on the feature GC content and the curve will show the type of dependence.

An example of this plot is in Fig. 9. In this case, the "GCbias" data were also generated for each condition (kidney and liver) using the argument `factor`.

```
> myGCbias = dat(mydata, factor = "Tissue", type = "GCbias")
> explo.plot(myGCbias, samples = NULL, toplot = "global")
```


### 3.4.3  RNA composition

When two samples have different RNA composition, the distribution of sequencing reads across the features is different in such a way that although a feature had the same number of read counts in both samples, it would not mean that it was equally expressed in both. To check if this bias is present in the data, the "cd" plot and the correponding diagnostic test can be used. In this case, each sample $s$ is compared to the reference sample $r$ (which can be arbitrarily chosen). To do that, M values are computed as $log2(counts_s = counts_r)$. If no bias is present, it should be expected that the median of M values for each comparison is 0. Otherwise, it would be indicating that expression levels in one of the samples tend to be higher than in the other, and this could lead to false discoveries when computing differencial expression. Confidence intervals for the M median are also computed by bootstrapping. If value 0 does not fall inside the interval, it means that the deviation of the sample with regard to the reference sample is statistically significant. Therefore, a normalization procedure such as Upper Quartile, TMM or DESeq should be used to correct this effect and make the samples comparable before computing differential expression. Confidence intervals can be visualized by using `show` function.

See below an usage example and the resulting plot in Fig. 10. It must be indicated if the data provided are already normalized (`norm=TRUE`) or not (`norm=FALSE`). The reference sample may be indicated with the refColumn
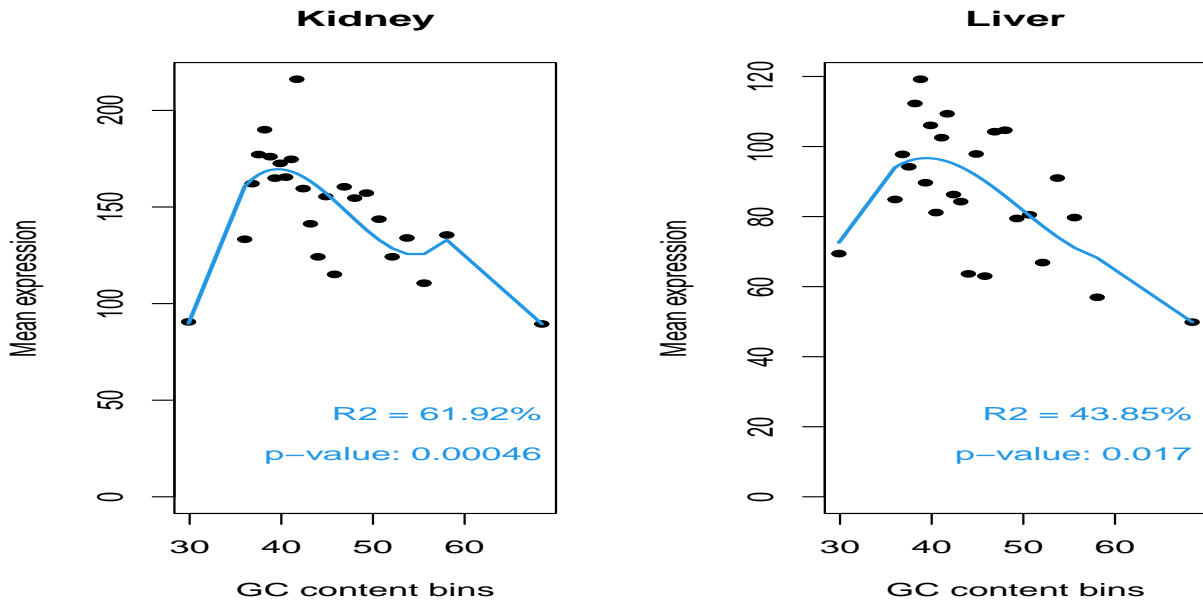
Figure 9: Gene GC content versus expression.

parameter (by default, the first column is used). Additional plot parameters may also be used to modify some aspects of the plot.

```
> mycd = dat(mydata, type = "cd", norm = FALSE, refColumn = 1)

[1] "Reference sample is: R1L1Kidney"
[1] "Confidence intervals for median of M:"
             0.28%                  99.72%                Diagnostic Test
R1L2Liver  "-0.894466721890836"   "-0.763330283804915"   "FAILED"
R1L3Kidney "-0.0471703644087826"  "-0.0471703644087824"  "FAILED"
R1L4Liver  "-0.879624287764932"   "-0.758379573704461"   "FAILED"
R1L6Liver  "-0.908076073429949"   "-0.763726449841552"   "FAILED"
R1L7Kidney "0.0348451027997053"   "0.0348451027997056"   "FAILED"
R1L8Liver  "-0.897765754709268"   "-0.759807457379734"   "FAILED"
R2L2Kidney "-0.0850229820491386"  "-0.0450542802279191"  "FAILED"
R2L3Liver  "-0.875456938855613"   "-0.753522173400496"   "FAILED"
R2L6Kidney "-0.0684076863358299"  "-0.0356829588903268"  "FAILED"
[1] "Diagnostic test: FAILED. Normalization is required to correct this bias."

> explo.plot(mycd)
```

In the plot can be seen that the $M$ median is deviated from 0 in most of the cases. This is corraborated by the confidence intervals for the $M$ median.

## 3.5 PCA exploration

One of the techniques that can be used to visualize if the experimental samples are clustered according to the experimental design or if there is an unwanted source of noise in the data that hampers this clustering is the Principal Component Analysis (PCA).

PCA is a dimension reduction method that does not require any distributional assumption, but it usually works better if data distribution is not too skewed, as happens in RNA-seq data. This is why, NOISeq package log-tranforms the expression data when users indicate that they have not already been log-tranformed.

NOISeq PCA function allows to plot the loading values, that is, the projection of the genes on the new principal components, or the scores, which are the projections of the samples (observations) on the space created by the new componets.

To illustrate the utility of the PCA plots, we took Marioni's data and artificially added a batch effect to the first four samples that would belong then to bath 1. The rest of samples would belong to batch2, so we also create an additional factor to collect the batch information.
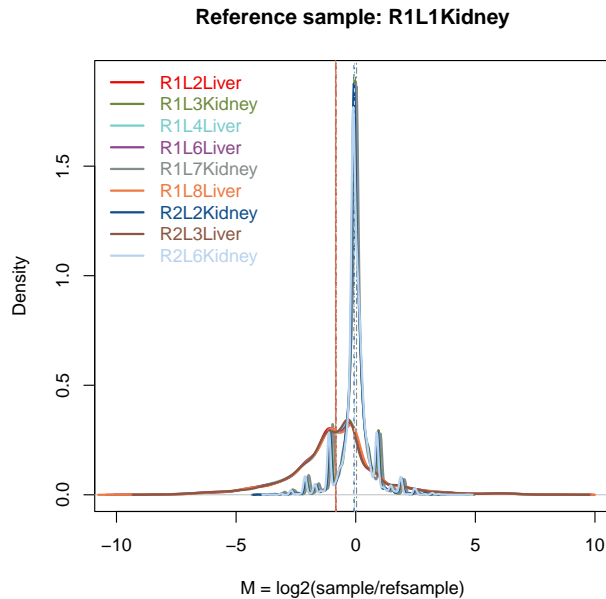
13

**Reference sample: R1L1Kidney**



Figure 10: RNA composition plot

```
> set.seed(123)
> mycounts2 = mycounts
> mycounts2[, 1:4] = mycounts2[, 1:4] + runif(nrow(mycounts2) * 4, 3, 5)
> myfactors = data.frame(myfactors, batch = c(rep(1, 4), rep(2, 6)))
> mydata2 = readData(mycounts2, factors = myfactors)
```

Now we can run the following code to plot the samples scores for the two principal components of the PCA and color them by the factor "Tissue" (left hand plot) or by the factor "batch" (right hand plot):

```
> myPCA = dat(mydata2, type = "PCA")
> par(mfrow = c(1, 2))
> explo.plot(myPCA, factor = "Tissue")
> explo.plot(myPCA, factor = "batch")
```

We can appreciate in these plots that the two batches are quite separated so removing the batch effect should improve the clustering of the samples. More information on how to do that with NOISeq can be found in Section 4.3.

## 3.6 Quality Control report

The `QCreport` function allows the user to quickly generate a pdf report showing the exploratory plots described in this section to compare either two samples (if factor=NULL) or two experimental conditions (if `factor` is indicated). Depending on the biological information provided (biotypes, length or GC content), the number of plots included in the report may differ.

```
> QCreport(mydata, samples = NULL, factor = "Tissue", norm = FALSE)
```

This report can be generated before normalizing the data (`norm = FALSE`) or after normalization to check if unwanted effects were corrected (`norm = TRUE`).

Please note that the data are log-transformed when computing Principal Component Analysis (PCA).

# 4 Normalization, Low-count filtering & Batch effect correction

The normalization step is very important in order to make the samples comparable and to remove possibles biases in the data. It might also be useful to filter out low expression data prior to differential expression analysis, since they are less reliable and may introduce noise in the analysis.
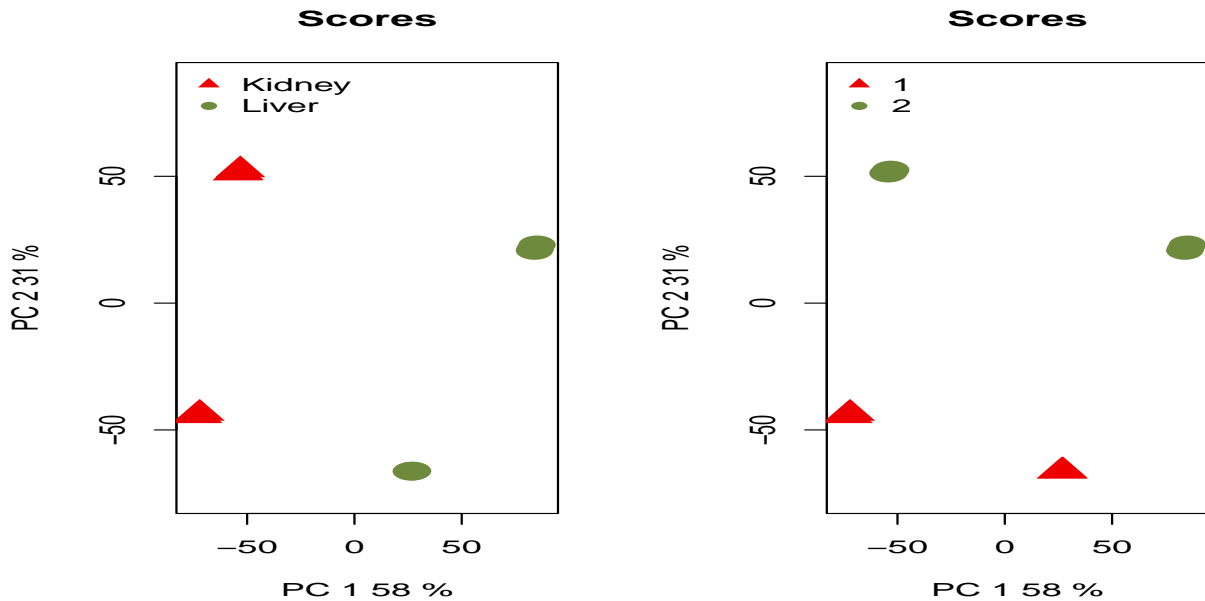
Figure 11: PCA plot colored by tissue (left) and by batch (right)

Next sections explain how to use NOISeq package to normalize and filter data before performing any statistical analysis.

## 4.1 Normalization

We strongly recommend to normalize the counts to correct, at least, sequencing depth bias. The normalization techniques implemented in NOISeq are RPKM [4], Upper Quartile [5] and TMM, which stands for Trimmed Mean of M values [6], but the package accepts data normalized with any other method as well as data previously transformed to remove batch effects or to reduce noise.

The normalization functions (`rpkm`, `tmm` and `uqua`) can be applied to common R matrix and data frame objects. Please, find below some examples on how to apply them to data matrix extracted from NOISeq data objects:

```
> myRPKM = rpkm(assayData(mydata)$exprs, long = mylength, k = 0, lc = 1)
> myUQUA = uqua(assayData(mydata)$exprs, long = mylength, lc = 0.5, k = 0)
> myTMM = tmm(assayData(mydata)$exprs, long = 1000, lc = 0)
> head(myRPKM[, 1:4])
```

```
                R1L1Kidney  R1L2Liver  R1L3Kidney  R1L4Liver
ENSG00000177757       1.87      0.816       0.000      0.000
ENSG00000187634      22.60     10.891      19.193     13.636
ENSG00000188976      43.36     17.664      44.265     28.926
ENSG00000187961       6.95      3.241       6.724      5.236
ENSG00000187583       0.27      0.000       0.262      0.235
ENSG00000187642       1.33      0.000       1.615      0.000
```

If the length of the features is provided to any of the normalization functions, the expression values are divided by $(length/1000)^{lc}$. Thus, although Upper Quartile and TMM methods themselves do not correct for the length of the features, NOISeq allows the users to combine these normalization procedures with an additional length correction whenever the length information is available. If $lc = 0$, no length correction is applied. To obtain RPKM values, $lc = 1$ in `rpkm` function must be indicated. If $long = 1000$ in `rpkm` function, CPM values (counts per million) are returned.

The $k$ parameter is used to replace the zero values in the expression matrix with other non-zero value in order to avoid indetermination in some calculations such as fold-change. If $k = NULL$, each 0 is replaced with the midpoint between 0 and the next non-zero value in the matrix.

## 4.2 Low-count filtering

Excluding features with low counts improves, in general, differential expression results, no matter the method being used, since noise in the data is reduced. However, the best procedure to filter these low count features has not been yet decided nor implemented in the differential expression packages. NOISeq includes three methods to filter out features with low counts:

- **CPM** (method 1): The user chooses a value for the parameter counts per million (CPM) in a sample under which a feature is considered to have low counts. The cutoff for a condition with $s$ samples is $CPM \times s$. Features with sum of expression values below the condition cutoff in all conditions are removed. Also a cutoff for the coefficient of variation (in percentage) per condition may be established to eliminate features with inconsistent expression values.

- **Wilcoxon test** (method 2): For each feature and condition, $H_0 : m = 0$ is tested versus $H_1 : m > 0$, where $m$ is the median of counts per condition. Features with p-value $> 0.05$ in all conditions are filtered out. P-values can be corrected for multiple testing using the `p.adj` option. This method is only recommended when the number of replicates per condition is at least 5.

- **Proportion test** (method 3): Similar procedure to the Wilcoxon test but testing $H_0 : p = p_0$ versus $H_1 : p > p_0$, where $p$ is the feature relative expression and $p_0 = CPM/10^6$. Features with p-value $> 0.05$ in all conditions are filtered out. P-values can be corrected for multiple testing using the `p.adj` option.

This is an usage example of function `filtered.data` directly on count data with CPM method (method 1):

```
> myfilt = filtered.data(mycounts, factor = myfactors$Tissue, norm = FALSE,
+      depth = NULL, method = 1, cv.cutoff = 100, cpm = 1, p.adj = "fdr")

Filtering out low count features...
4406 features are to be kept for differential expression analysis with filtering method 1
```

The "Sensitivity plot" described in previous section can help to take decisions on the CPM threshold to use in methods 1 and 3.

## 4.3 Batch effect correction

When a batch effect is detected in the data or the samples are not properly clustered due to an unknown source of technical noise, it is usually appropriate to remove this batch effect or noise before proceeding with the differential expression analysis (or any other type of analysis).

ARSyNseq (ASCA Removal of Systematic Noise for sequencing data) is an R function implemented in NOISeq package that is designed for filtering the noise associated to identified or unidentified batch effects. The ARSyN method [7] combines analysis of variance (ANOVA) modeling and multivariate analysis of estimated effects (PCA) to identify the structured variation of either the effect of the batch (if the batch information is provided) or the ANOVA errors (if the batch information is unknown). Thus, ARSyNseq returns a filtered data set that is rich in the information of interest and includes only the random noise required for inferential analysis.

The main arguments of the ARSyNseq function are:

- `data`: A Biobase's eSet object created with the `readData` function.

- `factor`: Name of the factor (as it was given to the `readData` function) to be used in the ARSyN model (e.g. the factor containing the batch information). When it is NULL, all the factors are considered.

- `batch`: TRUE to indicate that the `factor` argument indicates the batch information. In this case, the `factor` argument must be used to specify the names of the onlu factor containing the information of the batch.

- `norm`: Type of normalization to be used. One of "rpkm" (default), "uqua", "tmm" or "n" (if data are already normalized). If length was provided through the `readData` function, it will be considered for the normalization (except for "n"). Please note that if a normalization method if used, the arguments `lc` and `k` are set to 1 and 0 respectively.

- `logtransf`: If FALSE, a log-transformation will be applied on the data before computing ARSyN model to improve the results of PCA on count data.

Therefore, we can differentiate two types of analysis:

1. When batch is identified with one of the factors described in the argument `factor` of the `data` object, `ARSyNseq` estimates this effect and removes it by estimating the main PCs of the ANOVA effects associated. In such case `factor` argument will be the name of the batch and `batch=TRUE`.

2. When batch is not identified, the model estimates the effects associated to each factor of interest and analyses if there exists systematic noise in the residuals. If there is batch effect, it will be identified and removed by estimating the main PCs of these residuals. In such case `factor` argument can have several factors and `batch=FALSE`.

We will use the toy example generated in Section 3.5 to illustrate how `ARSyNseq` works. This is the code to use `ARSyNseq` batch effect correction when the user knows the batch in which the samples were processed, and to represent a PCA with the filtered data in order to see how the batch effect was corrected (Figure 12:

```
> mydata2corr1 = ARSyNseq(mydata2, factor = "batch", batch = TRUE, norm = "rpkm",
+     logtransf = FALSE)
> myPCA = dat(mydata2corr1, type = "PCA")
> par(mfrow = c(1, 2))
> explo.plot(myPCA, factor = "Tissue")
> explo.plot(myPCA, factor = "batch")
```
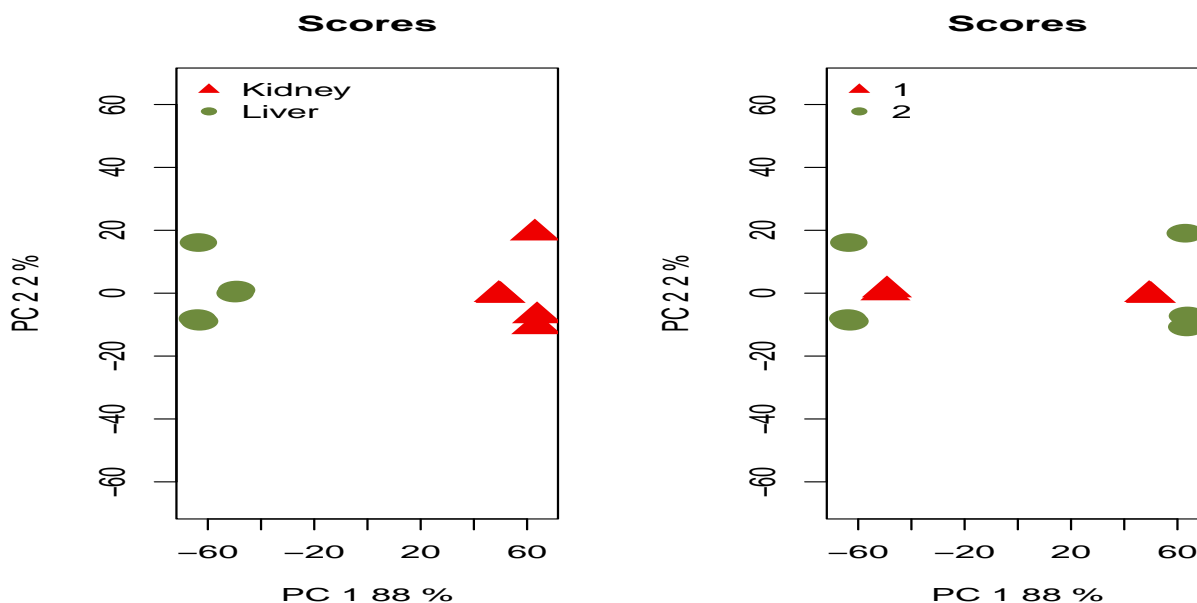


Figure 12: PCA plot after correcting a known batch effect with `ARSyNseq`. The samples are colored by tissue (left) and by batch (right)

Let us suppose now that we do not know the batch information. However, we can appreciate in the PCA plot of Section 3.5 that there is an unknown source of noise that prevents the samples from clustering well. In this case, we can run the following code to reduce the unidentified batch effect and to draw the PCA plots on the filtered data:

```
> mydata2corr2 = ARSyNseq(mydata2, factor = "Tissue", batch = FALSE, norm = "rpkm",
+     logtransf = FALSE)
> myPCA = dat(mydata2corr2, type = "PCA")
> par(mfrow = c(1, 2))
> explo.plot(myPCA, factor = "Tissue")
> explo.plot(myPCA, factor = "batch")
```
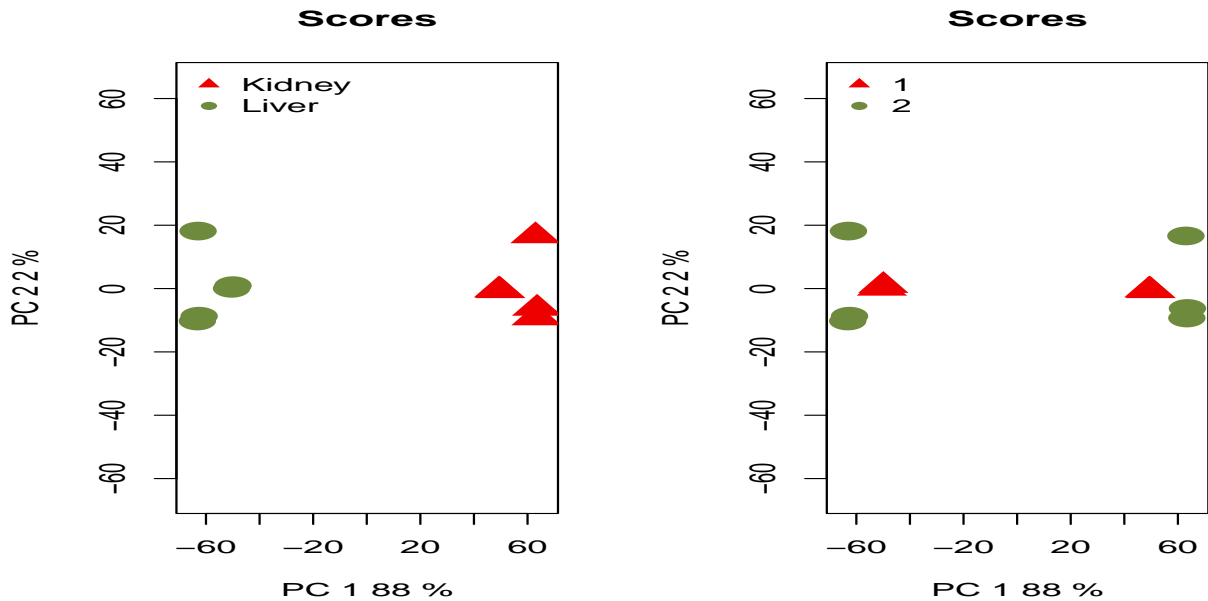
Figure 13: PCA plot after correcting an unidentified batch effect with `ARSyNseq`. The samples are colored by tissue (left) and by batch (right)

# 5 Differential expression

The `NOISeq` package computes differential expression between two experimental conditions given the expression level of the considered features. The package includes two non-parametric approaches for differential expression analysis: `NOISeq` [1] for technical replicates or no replication at all, and `NOISeqBIO` [2], which is optimized for the use of biological replicates. Both methods take read counts from RNA-seq as the expression values, in addition to previously normalized data and read counts from other NGS technologies.

In the previous section, we described how to use normalization and filtering functions prior to perform differential expression analysis. However, when using `NOISeq` or `NOISeqBIO` to compute differential expression, it is not necessary to normalize or filter low counts before applying these methods because they include these options. Thus, normalization can be done automatically by choosing the corresponding value for the parameter `norm`. Furthermore, they also accept expression values normalized with other packages or procedures. If the data have been previously normalized, `norm` parameter must be set to "n". Regarding the low-count filtering, it is not necessary to filter in `NOISeq` method. In contrast, it is recommended to do it in `NOISeqBIO`, which by default filters out low-count features with CPM method (`filter=1`).

The following sections describe in more detail the `NOISeq` and `NOISeqBIO` methods.

## 5.1 NOISeq

`NOISeq` method was designed to compute differential expression on data with technical replicates (NOISeq-real) or no replicates at all (NOISeq-sim). If there are technical replicates available, it summarizes them by summing up them. It is also possible to apply this method on biological replicates, that are averaged instead of summed. However, for biological replicates we strongly recommend `NOISeqBIO`. `NOISeq` computes the following differential expression statistics for each feature: $M$ (which is the $log_2$-ratio of the two conditions) and $D$ (the value of the difference between conditions). Expression levels equal to 0 are replaced with the given constant $k > 0$, in order to avoid infinite or undetermined $M$-values. If $k = NULL$, the 0 is replaced by the midpoint between 0 and the next non-zero value in the expression matrix. A feature is considered to be differentially expressed if its corresponding $M$ and $D$ values are likely to be higher than in noise. Noise distribution is obtained by comparing all pairs of replicates within the same condition. The corresponding $M$ and $D$ values are pooled together to generate the distribution. Changes in expression between conditions with the same magnitude than changes in expression between replicates within the same condition should not be considered as differential expression. Thus, by comparing the $(M, D)$ values of a given feature against the noise distribution, `NOISeq` obtains the "probability of differential expression" for this feature. If the odds Pr(differential expression)/Pr(non-differential expression) are higher than a given threshold, the feature is considered to be differentially expressed between conditions. For instance, an odds value of 4:1 is equivalent to $q = $ Pr(differential expression) $= 0.8$ and it means

that the feature is 4 times more likely to be differentially expressed than non-differentially expressed. The NOISeq algorithm compares replicates within the same condition to estimate noise distribution (NOISeq-real). When no replicates are available, NOISeq-sim simulates technical replicates in order to estimate the differential expression probability. Please remember that to obtain a really reliable statistical results, you need biological replicates. NOISeq-sim simulates technical replicates from a multinomial distribution, so be careful with the interpretation of the results when having no replicates, since they are only an approximation and are only showing which genes are presenting a higher change between conditions in your particular samples.

Table 1 summarizes all the input options and includes some recommendations for the values of the parameters when using NOISeq:

Table 1: Possibilities for the values of the parameters

| Method | Replicates | Counts | norm | k | nss | pnr | v |
|---|---|---|---|---|---|---|---|
| NOISeq-real | Technical/Biological | Raw<br>Normalized | rpkm, uqua, tmm<br>n | 0.5<br>NULL | 0 | - | - |
| NOISeq-sim | None | Raw<br>Normalized | rpkm, uqua, tmm<br>n | 0.5<br>NULL | $\geq 5$ | 0.2 | 0.02 |

Please note that norm = "n" argument should be used in noiseq or noiseqbio whenever the data have been previously normalized or corrected for a batch effect.

### 5.1.1 NOISeq-real: using available replicates

NOISeq-real estimates the probability distribution for M and D in an empirical way, by computing M and D values for every pair of replicates within the same experimental condition and for every feature. Then, all these values are pooled together to generate the noise distribution. Two replicates in one of the experimental conditions are enough to run the algorithm. If the number of possible comparisons within a certain condition is higher than 30, in order to reduce computation time, 30 pairwise comparisons are randomly chosen when estimating noise distribution.

It should be noted that biological replicates are necessary if the goal is to make any inference about the population. Deriving differential expression from technical replicates is useful for drawing conclusions about the specific samples being compared in the study but not for extending these conclusions to the whole population.

In RNA-seq or similar sequencing technologies, the counts from technical replicates (e.g. lanes) can be summed up. Thus, this is the way the algorithm summarizes the information from technical replicates to compute M and D signal values (between different conditions). However, for biological replicates, other summary statistics such us the mean may be more meaningful. NOISeq calculates the mean of the biological replicates but we strongly recommend to use NOISeqBIO when having biological replicates.

Here there is an example with technical replicates and count data normalized by rpkm method. Please note that, since the factor "Tissue" has two levels, we do not need to indicate which conditions are to be compared.

```
> mynoiseq = noiseq(mydata, k = 0.5, norm = "rpkm", factor = "Tissue", pnr = 0.2,
+     nss = 5, v = 0.02, lc = 1, replicates = "technical")

[1] "Computing (M,D) values..."
[1] "Computing probability of differential expression..."

> head(mynoiseq@results[[1]])

                Kidney_mean Liver_mean     M      D  prob ranking Length   GC Chrom GeneStart
ENSG00000177757       1.448      0.493 1.553  0.955 0.621    1.82   2464 48.6     1    742614
ENSG00000187634      19.860     11.706 0.763  8.154 0.751    8.19   4985 66.0     1    850393
ENSG00000188976      42.631     24.290 0.812 18.341 0.787   18.36   3870 59.5     1    869459
ENSG00000187961       6.289      5.225 0.267  1.064 0.428    1.10   4964 67.9     1    885830
ENSG00000187583       0.419      0.143 1.553  0.276 0.401    1.58   8507 62.6     1    891740
ENSG00000187642       2.524      0.412 2.616  2.113 0.782    3.36   6890 67.7     1    900447
                GeneEnd        Biotype
ENSG00000177757  745077         lincRNA
ENSG00000187634  869824  protein_coding
ENSG00000188976  884494  protein_coding
ENSG00000187961  890958  protein_coding
```

```
ENSG00000187583  900339 protein_coding
ENSG00000187642  907336 protein_coding
```

NA values would be returned if the gene had 0 counts in all the samples. In that case, the gene would not be used to compute differential expression.

Now imagine you want to compare tissues within the same sequencing run. Then, see the following example on how to apply NOISeq on count data with technical replicates, TMM normalization, and no length correction. As "TissueRun" has more than two levels we have to indicate which levels (conditions) are to be compared:

```
> mynoiseq.tmm = noiseq(mydata, k = 0.5, norm = "tmm", factor = "TissueRun",
+       conditions = c("Kidney_1", "Liver_1"), lc = 0, replicates = "technical")
```

### 5.1.2  NOISeq-sim: no replicates available

When there are no replicates available for any of the experimental conditions, NOISeq can simulate technical replicates. The simulation relies on the assumption that read counts follow a multinomial distribution, where probabilities for each class (feature) in the multinomial distribution are the probability of a read to map to that feature. These mapping probabilities are approximated by using counts in the only sample of the corresponding experimental condition. Counts equal to zero are replaced with $k>0$ to give all features some chance to appear.

Given the sequencing depth (total amount of reads) of the unique available sample, the size of the simulated samples is a percentage (parameter $pnr$) of this sequencing depth, allowing a small variability (given by the parameter $v$). The number of replicates to be simulated is provided by $nss$ parameter.

Our dataset do has replicates but, providing it had not, you would use NOISeq-sim as in the following example in which the simulation parameters have to be chosen ($pnr$, $nss$ and $v$):

```
> myresults <- noiseq(mydata, factor = "Tissue", k = NULL, norm = "n", pnr = 0.2,
+       nss = 5, v = 0.02, lc = 1, replicates = "no")
```

### 5.1.3  NOISeqBIO

NOISeqBIO is optimized for the use on biological replicates (at least 2 per condition). It was developed by joining the philosophy of our previous work together with the ideas from Efron *et al.* in [8]. In our case, we defined the differential expression statistic $\theta$ as $(M + D)/2$, where $M$ and $D$ are the statistics defined in the previous section but including a correction for the biological variability of the corresponding feature. The probability distribution of $\theta$ can be described as a mixture of two distributions: one for features changing between conditions and the other for invariant features. Thus, the mixture distribution $f$ can be written as: $f(\theta) = p_0 f_0(\theta) + p_1 f_1(\theta)$, where $p_0$ is the probability for a feature to have the same expression in both conditions and $p_1 = 1 - p_0$ is the probability for a feature to have different expression between conditions. $f_0$ and $f_1$ are, respectively, the densities of $\theta$ for features with no change in expression between conditions and for differentially expressed features. If one of both distributions can be estimated, the probability of a feature to belong to one of the two groups can be calculated. Thus, the algorithm consists of the following steps:

1. Computing $\theta$ values.
   $M$ and $D$ are corrected for the biological variability: $M^* = \dfrac{M}{a_0 + \hat{\sigma}_M}$ and $D^* = \dfrac{D_s}{a_0 + \hat{\sigma}_D}$, where $\hat{\sigma}_M^2$ and $\hat{\sigma}_D^2$ are the standard errors of $M_s$ and $D_s$ statistics, respectively, and $a_0$ is computed as a given percentile of all the values in $\hat{\sigma}_M$ or $\hat{\sigma}_D$, as in [8] (the authors suggest the percentile 90th as the best option, which is the default option of the parameter "a0per" that may be changed by the user). To compute the $\theta$ statistic, the $M$ and $D$ statistics are combined: $\theta = \dfrac{M^* + D^*}{2}$.

2. Estimating the values of the $\theta$ statistic when there is no change in expression, i.e. the null statistic $\theta_0$.
   In order to compute the null density $f_0$ afterwards, we first need to estimate the values of the $\theta$-scores for features with no change between conditions. To do that, we permute $r$ times (parameter that may be set by the user) the labels of samples between conditions, compute $\theta$ values as above and pool them to obtain $\theta_0$.

3. Estimating the probability density functions $f$ and $f_0$.
   We estimate $f$ and $f_0$ with a kernel density estimator (KDE) with Gaussian kernel and smoothing parameter "adj" as indicated by the user.

4. Computing the probability of differential expression given the ratio $f_0/f$ and an estimation $\hat{p}_0$ for $p_0$. If $\theta = z$ for a given feature, this probability of differential expression can be computed as $p_1(z) = 1 - \hat{p}_0 f_0(z)/f(z)$.

To estimate $p_0$, the following upper bound is taken, as suggested in [8]: $p_0 \leq \min_Z\{f(Z)/f_0(Z)\}$.
Moreover, it is shown in [8] that the FDR defined by Benjamini and Hochberg can be considered equivalent to the *a posteriori* probability $p_0(z) = 1 - p_1(z)$ we are calculating.

When too few replicates are available for each condition, the null distribution is very poor since the number of different permutations is low. For those cases (number of replicates in one of the conditions less than 5), it is convenient to borrow information across genes. Our proposal consists of clustering all genes according to their expression values across replicates using the k-means method. For each cluster $k$ of genes, we consider the expression values of all the genes in the cluster as observations within the corresponding condition (replicates) and then we shuffle this submatrix $r \times g_k$ times, where $g_k$ is the number of genes within cluster $k$. If $r \times g_k$ is higher than 1000, we compute 1000 permutations in that cluster. For each permutation, we calculate $M$ and $D$ values and their corresponding standard errors. In order to reduce the computing time, if $g_k \geq 1000$, we again subdivide cluster $k$ in subclusters with k-means algorithm.

We will consider that Marioni's data have biological replicates for the following example. In this case, the method 2 (Wilcoxon test) to filter low counts is used. Please, use `?noiseqbio` to know more about the parameters of the function.

```
> mynoiseqbio = noiseqbio(mydata, k = 0.5, norm = "rpkm", factor = "Tissue",
+     lc = 1, r = 20, adj = 1.5, plot = FALSE, a0per = 0.9, random.seed = 12345,
+     filter = 2)
```

## 5.2 Results

### 5.2.1 NOISeq output object

NOISeq returns an `Output` object containing the following elements:

- **comparison**: String indicating the two experimental conditions being compared and the sense of the comparison.

- **factor**: String indicating the factor chosen to compute the differential expression.

- **k**: Value to replace zeros in order to avoid indetermination when computing logarithms.

- **lc**: Correction factor for length normalization. Counts are divided by $length^{lc}$.

- **method**: Normalization method chosen.

- **replicates**: Type of replicates: "technical" for technical replicates and "biological" for biological ones.

- **results**: R data frame containing the differential expression results, where each row corresponds to a feature. The columns are: Expression values for each condition to be used by `NOISeq` or `NOISeqBIO` (the columns names are the levels of the factor); differential expression statistics (columns "M" and "D" for `NOISeq` or "theta" for `NOISeqBIO`); probability of differential expression ("prob"); "ranking", which is a summary statistic of "M" and "D" values equal to $-sign(M) \times \sqrt{M^2 + D^2}$, than can be used for instance in gene set enrichment analysis (only for `NOISeq`); "Length" of each feature (if provided); "GC" content of each feature (if provided); chromosome where the feature is ("Chrom"), if provided; start and end position of the feature within the chromosome ("GeneStart", "GeneEnd"), if provided; feature biotype ("Biotype"), if provided.

- **nss**: Number of samples to be simulated for each condition (only when there are not replicates available).

- **pnr**: Percentage of the total sequencing depth to be used in each simulated replicate (only when there are not replicates available). For instance, if pnr = 0.2 , each simulated replicate will have 20% of the total reads of the only available replicate in that condition.

- **v**: Variability of the size of each simulated replicate (only used by NOISeq-sim).

For example, you can use the following instruction to see the differential expression results for `NOISeq`:

```
> head(mynoiseq@results[[1]])
```

|  | Kidney_mean | Liver_mean | M | D | prob | ranking | Length | GC | Chrom | GeneStart |
|---|---|---|---|---|---|---|---|---|---|---|
| ENSG00000177757 | 1.448 | 0.493 | 1.553 | 0.955 | 0.621 | 1.82 | 2464 | 48.6 | 1 | 742614 |
| ENSG00000187634 | 19.860 | 11.706 | 0.763 | 8.154 | 0.751 | 8.19 | 4985 | 66.0 | 1 | 850393 |
| ENSG00000188976 | 42.631 | 24.290 | 0.812 | 18.341 | 0.787 | 18.36 | 3870 | 59.5 | 1 | 869459 |

```
ENSG00000187961        6.289        5.225 0.267  1.064 0.428        1.10    4964 67.9        1        885830
ENSG00000187583        0.419        0.143 1.553  0.276 0.401        1.58    8507 62.6        1        891740
ENSG00000187642        2.524        0.412 2.616  2.113 0.782        3.36    6890 67.7        1        900447
                  GeneEnd         Biotype
ENSG00000177757   745077            lincRNA
ENSG00000187634   869824  protein_coding
ENSG00000188976   884494  protein_coding
ENSG00000187961   890958  protein_coding
ENSG00000187583   900339  protein_coding
ENSG00000187642   907336  protein_coding
```

The output `myresults@results[[1]]$prob` gives the estimated probability of differential expression for each feature. Note that when using NOISeq, these probabilities are not equivalent to p-values. The higher the probability, the more likely that the difference in expression is due to the change in the experimental condition and not to chance. See Section 5.2 to learn how to obtain the differentially expressed features.

### 5.2.2 How to select the differentially expressed features

Once we have obtained the differential expression probability for each one of the features by using `NOISeq` or `NOISeqBIO` function, we may want to select the differentially expressed features for a given threshold $q$. This can be done with `degenes` function on the "output" object using the parameter `q`. With the argument `M` we choose if we want all the differentially expressed features, only the differentially expressed features that are more expressed in condition 1 than in condition 2 (M = "up") or only the differentially expressed features that are under-expressed in condition 1 with regard to condition 2 (M = "down"):

```
> mynoiseq.deg = degenes(mynoiseq, q = 0.8, M = NULL)

[1] "1614 differentially expressed features"

> mynoiseq.deg1 = degenes(mynoiseq, q = 0.8, M = "up")

[1] "1289 differentially expressed features (up in first condition)"

> mynoiseq.deg2 = degenes(mynoiseq, q = 0.8, M = "down")

[1] "325 differentially expressed features (down in first condition)"
```

Please remember that, when using `NOISeq`, the probability of differential expression is not equivalent to $1 - pvalue$. We recommend for $q$ to use values around 0.8. If `NOISeq-sim` has been used because no replicates are available, then it is preferable to use a higher threshold such as $q = 0.9$. However, when using `NOISeqBIO`, the probability of differential expression would be equivalent to $1 - FDR$, where $FDR$ can be considered as an adjusted p-value. Hence, in this case, it would be more convenient to use $q = 0.95$.

### 5.2.3 Plots on differential expression results

**Expression plot**

Once differential expression has been computed, it is interesting to plot the average expression values of each condition and highlight the features declared as differentially expressed. It can be done with the `DE.plot`.

To plot the summary of the expression values in both conditions as in Fig. 14, please write the following code (many graphical parameters can be adjusted, see the function help). Note that by giving $q = 0.9$, differentially expressed features considering this threshold will be highlighted in red:

```
> DE.plot(mynoiseq, q = 0.9, graphic = "expr", log.scale = TRUE)

[1] "888 differentially expressed features"
```

**MD plot**

Instead of plotting the expression values, it is also interesting to plot the log-fold change ($M$) and the absolute value of the difference in expression between conditions ($D$) as in Fig. 15. This is an example of the code to get such a plot ($D$ values are displayed in log-scale) from `NOISeq` output (it is analogous for `NOISeqBIO` ouput).

```
> DE.plot(mynoiseq, q = 0.8, graphic = "MD")

[1] "1614 differentially expressed features"
```
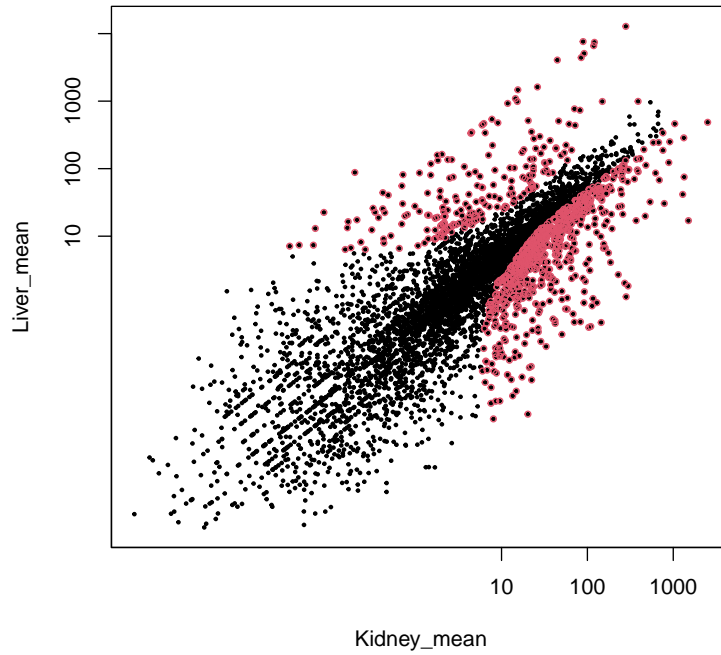
Figure 14: Summary plot of the expression values for both conditions (black), where differentially expressed genes are highlighted (red).
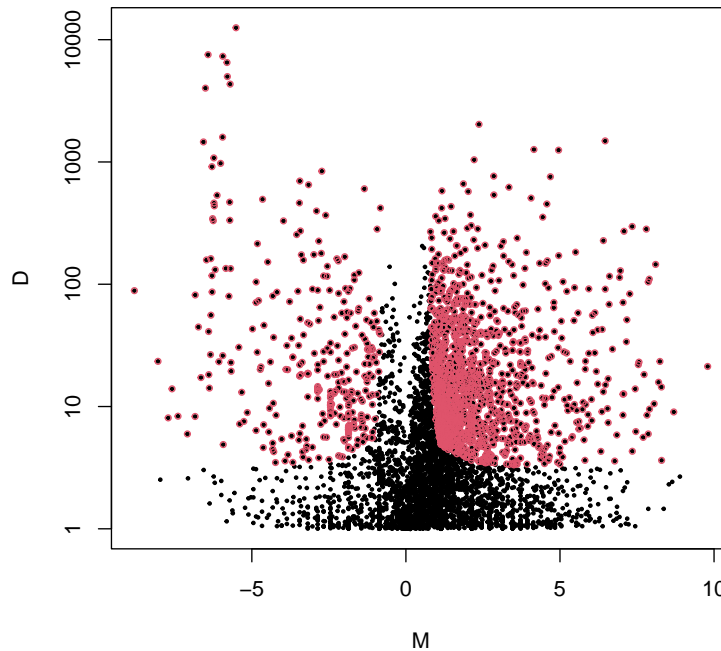


Figure 15: Summary plot for (M,D) values (black) and the differentially expressed genes (red).

**Manhattan plot**

The Manhattan plot can be used to display the expression of the genes across the chromosomes. The expression for both conditions under comparison is shown in the plot. The users may choose either plotting all the

chromosomes or only some of them, and also if the chromosomes are depicted consecutively (useful for prokaryote organisms) or separately (one per line). If a $q$ cutoff is provided, then differentially expressed features are highlighted in a different color. The following code shows how to draw the Manhattan plot from the output object returned by `NOISeq` or `NOISeqBIO`. In this case, using Marioni's data, the expression (log-transformed) is represented for two chromosomes (see Fig. 16). Note that the chromosomes will be depicted in the same order that are given to "chromosomes" parameter.

Gene expression is represented in gray. Lines above 0 correspond to the first condition under comparison (kidney) and lines below 0 are for the second condition (liver). Genes up-regulated in the first condition are highlighted in red, while genes up-regulated in the second condition are highlighted in green. The blue lines on the horizontal axis (Y=0) correspond to the annotated genes. X scale shows the location in the chromosome.

```
> DE.plot(mynoiseq, chromosomes = c(1, 2), log.scale = TRUE, join = FALSE,
+       q = 0.8, graphic = "chrom")

[1] "REMEMBER. You are plotting these chromosomes and in this order:"
[1] 1 2
[1] "1289 differentially expressed features (up in first condition)"
[1] "325 differentially expressed features (down in first condition)"
```
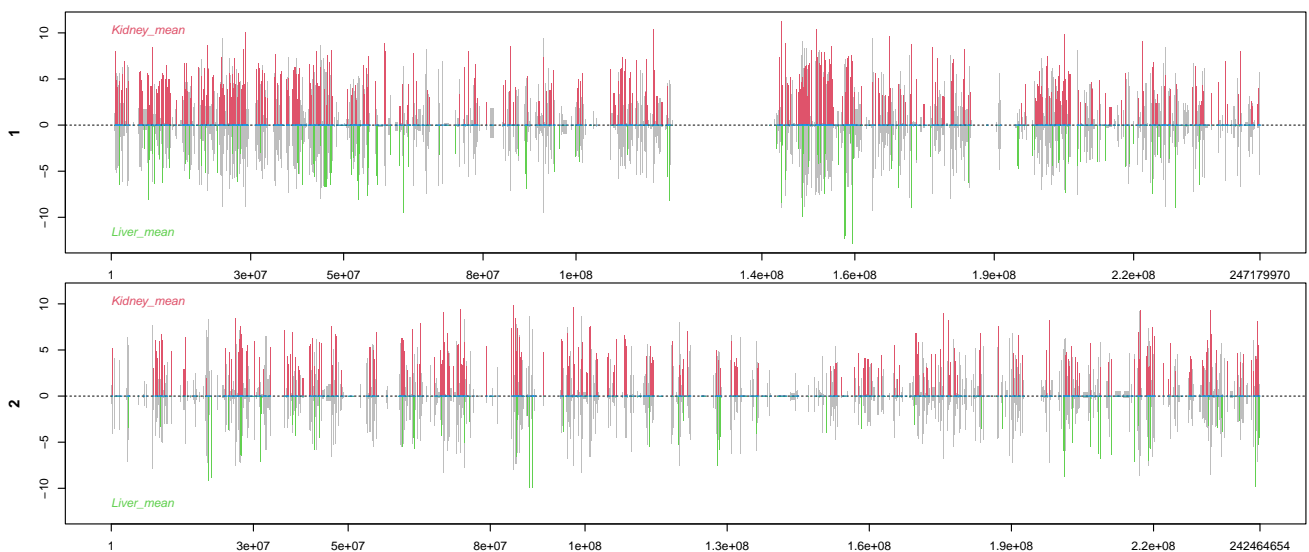


Figure 16: Manhattan plot for chromosomes 1 and 2

It is advisable, in this kind of plots, to save the figure in a file, for instance, a pdf file (as in the following code), in order to get a better visualization with the zoom.

```
pdf("manhattan.pdf", width = 12, height = 50)
DE.plot(mynoiseq, chromosomes = c(1,2), log.scale = TRUE,
        join = FALSE, q = 0.8)
dev.off()
```

**Distribution of differentially expressed features per chromosomes or biotypes**

This function creates a figure with two plots if both chromosomes and biotypes information is provided. Otherwise, only a plot is depicted with either the chromosomes or biotypes (if information of any of them is available). The $q$ cutoff must be provided. Both plots are analogous. The chromosomes plot shows the percentage of features in each chromosome, the proportion of them that are differentially expressed (DEG) and the percentage of differentially expressed features in each chromosome. Users may choose plotting all the chromosomes or only some of them. The chromosomes are depicted according to the number of features they contain (from the greatest to the lowest). The plot for biotypes can be described similarly. The only difference is that this plot has a left axis scale for the most abundant biotypes and a right axis scale for the rest of biotypes, which are separated by a green vertical line.

The following code shows how to draw the figure from the output object returned by `NOISeq` for the Marioni's example data.

```
> DE.plot(mynoiseq, chromosomes = NULL, q = 0.8, graphic = "distr")
```
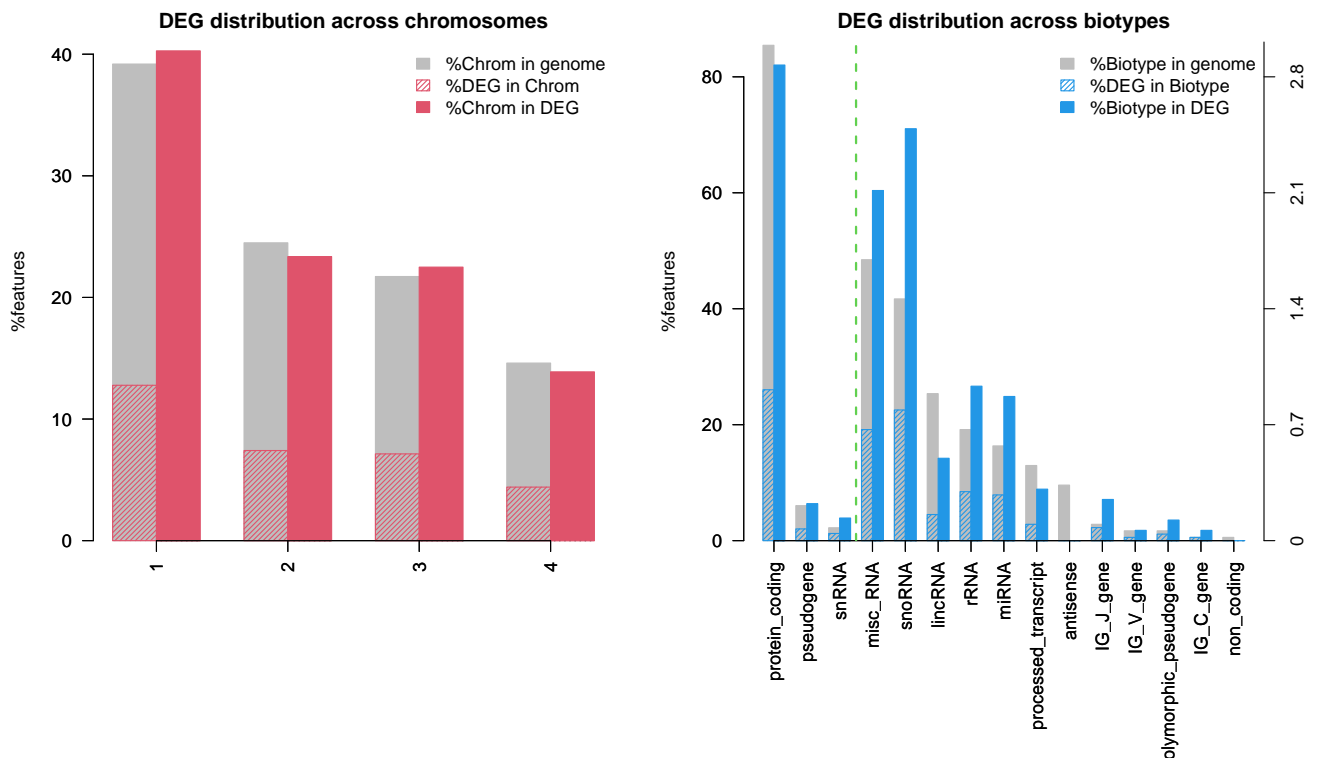
Figure 17: Distribution of DEG across chromosomes and biotypes for Marioni's example dataset.

```
[1] "1614 differentially expressed features"
```

# 6 Setup

This vignette was built on:

```
> sessionInfo()

R version 4.4.0 beta (2024-04-15 r86425)
Platform: x86_64-pc-linux-gnu
Running under: Ubuntu 22.04.4 LTS

Matrix products: default
BLAS:   /home/biocbuild/bbs-3.19-bioc/R/lib/libRblas.so
LAPACK: /usr/lib/x86_64-linux-gnu/lapack/liblapack.so.3.10.0

locale:
 [1] LC_CTYPE=en_US.UTF-8       LC_NUMERIC=C               LC_TIME=en_US.UTF-8
 [4] LC_COLLATE=en_US.UTF-8     LC_MONETARY=en_US.UTF-8    LC_MESSAGES=en_US.UTF-8
 [7] LC_PAPER=en_US.UTF-8       LC_NAME=C                  LC_ADDRESS=C
[10] LC_TELEPHONE=C             LC_MEASUREMENT=en_US.UTF-8 LC_IDENTIFICATION=C

time zone: America/New_York
tzcode source: system (glibc)

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

other attached packages:
```

```
[1] NOISeq_2.48.0        Matrix_1.7-0        Biobase_2.64.0        BiocGenerics_0.50.0

loaded via a namespace (and not attached):
[1] compiler_4.4.0 tools_4.4.0     grid_4.4.0        lattice_0.22-6
```

# References

[1] S. Tarazona, F. García-Alcalde, J. Dopazo, A. Ferrer, and A. Conesa. Differential expression in RNA-seq: A matter of depth. *Genome Research*, 21: 2213 - 2223, 2011.

[2] S. Tarazona, P. Furió-Tarí, D. Turrá, A. Di Pietro, M.J. Nueda, A. Ferrer, and A. Conesa. Data quality aware analysis of differential expression in RNA-seq with NOISeq R/Bioc package. *Nucleic Acids Research*, 43(21):e140, 2015.

[3] J.C. Marioni, C.E. Mason, S.M. Mane, M. Stephens, and Y. Gilad. RNA-seq: an assessment of technical reproducibility and comparison with gene expression arrays. *Genome Research*, 18: 1509 - 517, 2008.

[4] A. Mortazavi, B.A. Williams, K. McCue, L. Schaeffer, and B. Wold. Mapping and quantifying mammalian transcriptomes by RNA-Seq. *Nature Methods*, 5: 621 - 628, 2008.

[5] J.H. Bullard, E. Purdom, K.D. Hansen, and S. Dudoit. Evaluation of statistical methods for normalization and differential expression in mRNA-Seq experiments. *BMC bioinformatics*, 11(1):94, 2010.

[6] M.D. Robinson, and A. Oshlack. A scaling normalization method for differential expression analysis of RNA-Seq data. *Genome Biology*, 11: R25, 2010.

[7] M. Nueda, A. Conesa, and A. Ferrer. ARSyN: a method for the identification and removal of systematic noise in multifactorial time-course microarray experiments. *Biostatistics*, 13(3):553–566, 2012.

[8] B. Efron, R. Tibshirani, J.D. Storey, V. Tusher. Empirical Bayes Analysis of a Microarray Experiment. *Journal of the American Statistical Association*, 2001.