2024_RNA-seq-analysis

# Contents

# RNA-seq analysis workshop

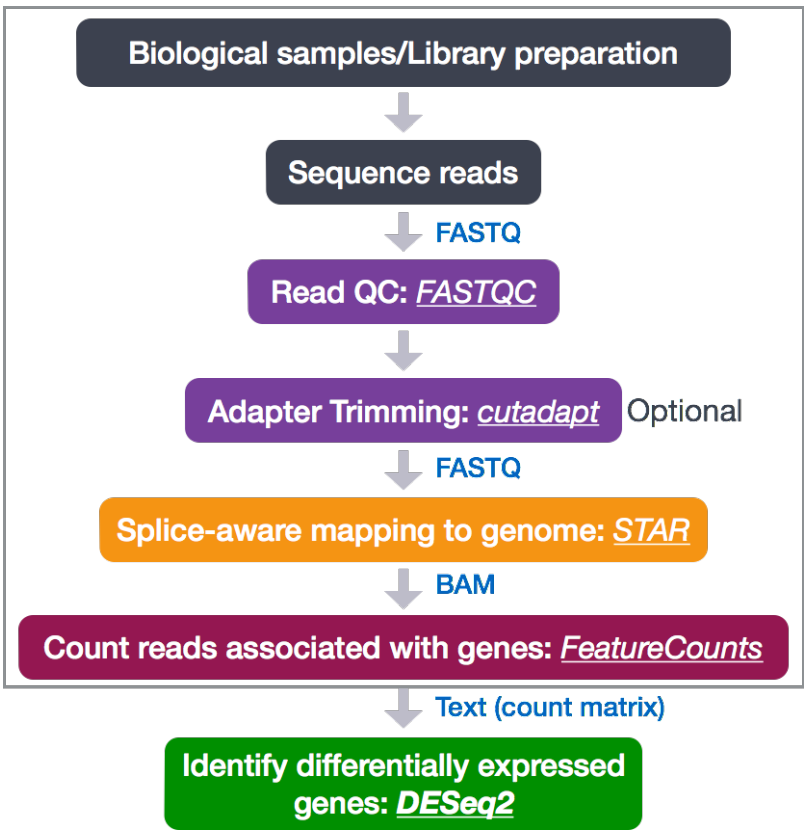
The published version for this module can be found on my bookdown site 2024_RNA-seq-analysis.

# Chapter 1

# Differential gene expression (DGE) analysis overview

The goal of RNA-seq is to perform differential expression testing to determine which genes are expressed at different levels between conditions. These genes can offer biological insight into the processes affected by the condition(s) of interest.

To determine the expression levels of genes, our RNA-seq workflow follows the steps detailed in the image below inside the box. All steps are performed on the command line (Linux/Unix) through the generation of read counts per gene as discussed in Corey's lectures. The differential expression analysis and any downstream functional analysis are generally performed in R using R packages specifically designed for the complex statistical analyses required to determine whether genes are differentially expressed starting from the count matrices.




In the next few lessons, we will walk you through an **end-to-end gene-level RNA-seq differential expression workflow** using various R packages. We will start with the count matrix, perform exploratory data analysis for quality assessment and to explore the relationship between samples, perform differential expression analysis, and visually explore the results prior to performing downstream functional analysis.

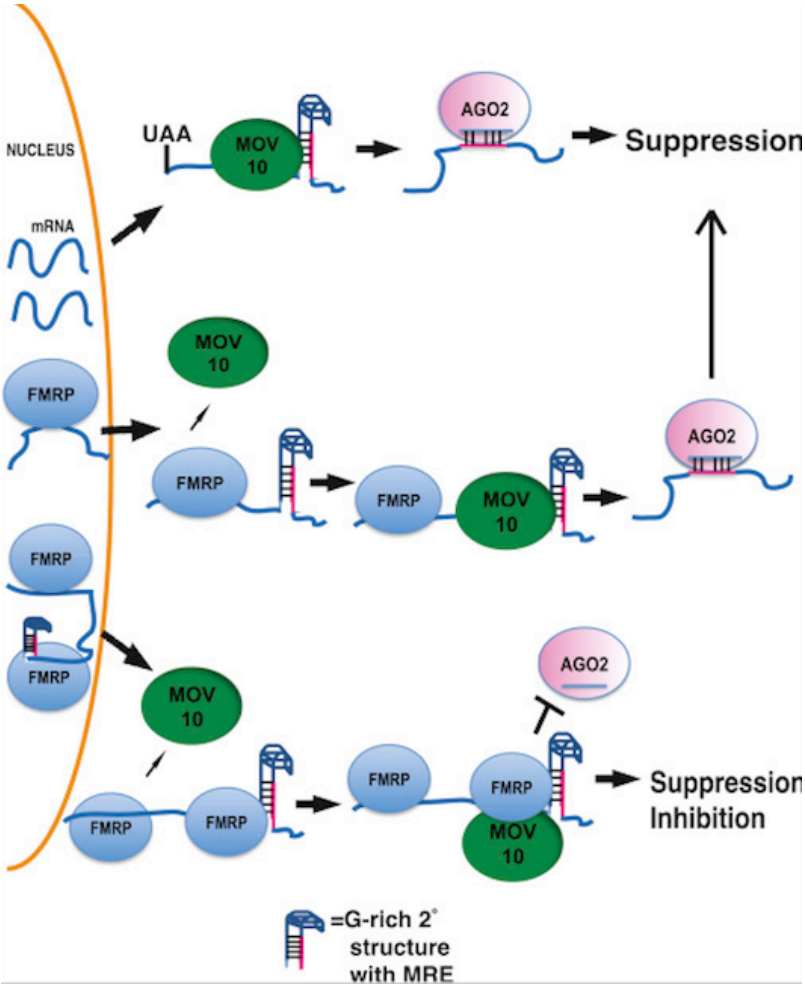
## 1.1 Review of the dataset

We will be using the full count matrix from the RNA-seq dataset that is part of a larger study described in Kenny PJ et al, Cell Rep 2014 investigating the interactions between genes potentially involved in Fragile X syndrome (FXS). FXS is a genetic disorder and the leading cause of inherited intellectual disabilities like autism. FXS is caused by aberrant production of a protein called Fragile X Messenger Ribonucleoprotein (**FMRP**) that is needed for brain development. People who have FXS do not make this protein. The authors demonstrated that **FMRP** associates with another RNA-binding protein **MOV10** (Mov10 RISC Complex RNA Helicase) and acts to regulate the translation of a subset of RNAs.

What is the function of **FMRP** and **MOV10**?

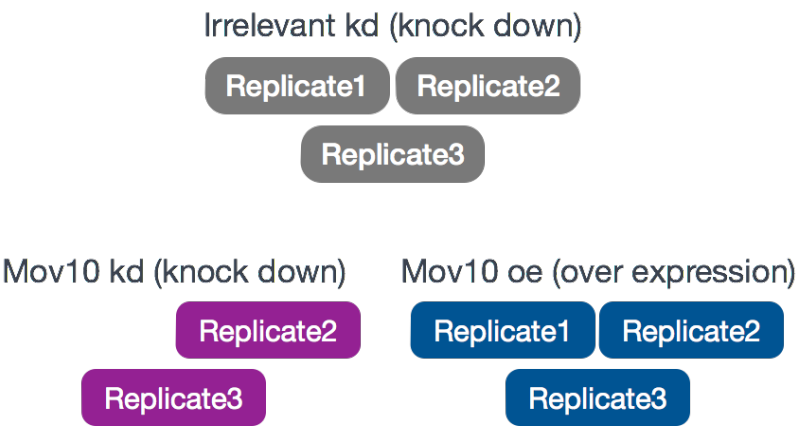
> **FMRP** is "most commonly found in the brain, is essential for normal cognitive development and female reproductive function. Mutations of this gene can lead to fragile X syndrome, mental retardation, premature ovarian failure, autism, Parkinson's disease, developmental delays and other cognitive deficits." - from wikipedia
>
> **MOV10** MOV10 has been shown to associate with the 3' UTR of mRNAs as part of its role in the RNA-induced silencing complex (RISC). MOV10 is an RNA helicase believed to assist in the unwinding of RNA duplexes. MOV10 is associated with **FMRP** in the context of the microRNA pathway. **MOV10** has a role in regulating translation: it facilitates microRNA-mediated translation of some RNAs, but it also increases expression of other RNAs.

The aim of the RNAseq part of the study was to characterize the transcription expression patterns of **FMRP** and **MOV10** to identify overlapping target genes which would suggest that these genes are regulated by the **MOV10-FMRP** complex.




**Model for** MOV10-FMRP** Association in Translation Regulation**. Top: fate of RNAs bound by** MOV10**.** MOV10** binds the 3 UTR-encoded G-rich structure to reveal MREs for subsequent AGO2 association. Middle: fate of RNAs bound by **FMRP**. **FMRP** binds RNAs in the nucleus. Upon export, **FMRP** recruits **MOV10**, which ultimately unwinds MREs for association with AGO2. Bottom: **FMRP** recruits **MOV10** to RNAs; however, binding of both **FMRP** and **MOV10** in proximity of MRE blocks association with AGO2. Red line indicates MRE.

RNA-seq was performed on HEK293F cells that were either transfected with a **MOV10** transgene, or siRNA to knock down Mov10 expression, or non-specific (irrelevant) siRNA. This resulted in 3 conditions **Mov10 oe** (over expression), **Mov10 kd** (knock down) and **Irrelevant kd**, respectively. The number of replicates is as shown below.




Using these data, we will evaluate transcriptional patterns associated with perturbation of **MOV10** expression. Please note that the irrelevant siRNA will be treated as our control condition.

**Links to raw data** [GEO]: https://www.ncbi.nlm.nih.gov/geo/query/acc.cgi?acc=GSE51443 "Gene Expression Omnibus" [SRA]: https://trace.ncbi.nlm.nih.gov/Traces/sra/?study=SRP031507 "Sequence Read Archive"

**Our questions:**

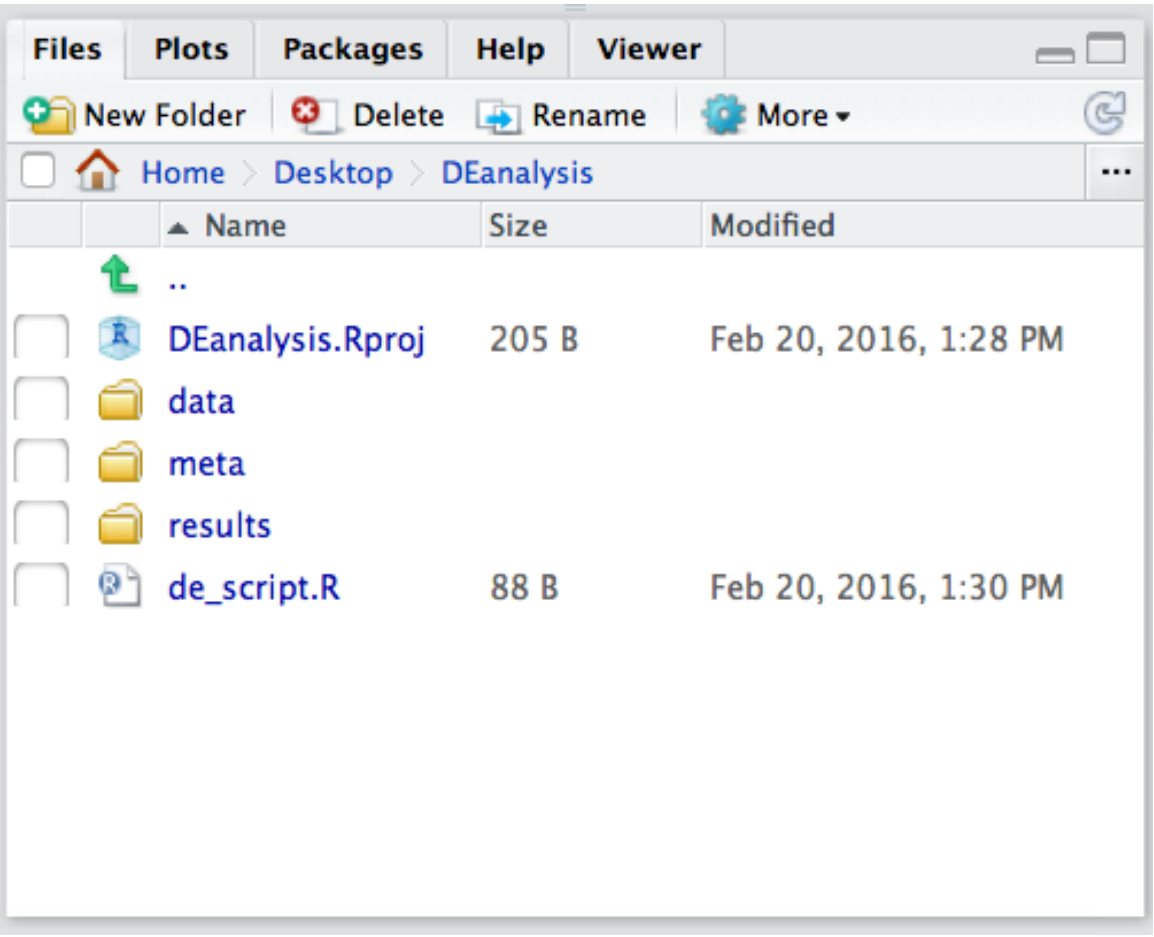
- What patterns of expression can we identify with the loss or gain of MOV10?

- Are there any genes that are differentially expressed between the two conditions?

- Are there any genes shared between the two conditions?

## 1.2 Setting up

Go to the `File` menu and to the `lessons` directory and open `09-DGE_codebook.Rmd`. This will open in the script editor in the top left hand corner. This is where we will be running all commands required for this analysis, and like last time this is where you will enter the code/answers for your homework assignment. Please save this as something with your name in it (e.g. `firstname_lastname_09-DGE_codebook.Rmd`).




### 1.2.1 Loading libraries

For this analysis we will be using several R packages, some which have been installed from CRAN and others from Bioconductor. To use these packages (and the functions contained within them), we need to **load the libraries**.

```
## Setup
### Bioconductor and CRAN libraries used
library(tidyverse)
library(RColorBrewer)
library(DESeq2)
library(pheatmap)
library(ggplot2)
library(ggrepel)
```

### 1.2.2 Loading data

To load the data into our current environment, we will be using the `read.table` function. We need to provide the path to each file and also specify arguments to let R know that we have a header (`header = T`) and the first column is our row names (`row.names = 1`). By default the function expects tab-delimited files, which is what we have.

```
## Load in data
data <- read.table("data/Mov10_full_counts.txt", header=T, row.names=1)

meta <- read.table("data/Mov10_full_meta.txt", header=T, row.names=1)
```

Use `class()` to inspect our data and make sure we are working with data frames:

```
### Check classes of the data we just brought in
class(meta)
```

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

```
class(data)
```

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

### 1.2.3 Viewing data

Make sure your datasets contain the expected samples / information before proceeding to perfom any type of analysis.

```
rownames(meta)
```

```
## [1] "Irrel_kd_1" "Irrel_kd_2" "Irrel_kd_3" "Mov10_kd_2" "Mov10_kd_3" "Mov10_oe_1"
## [7] "Mov10_oe_2" "Mov10_oe_3"
```

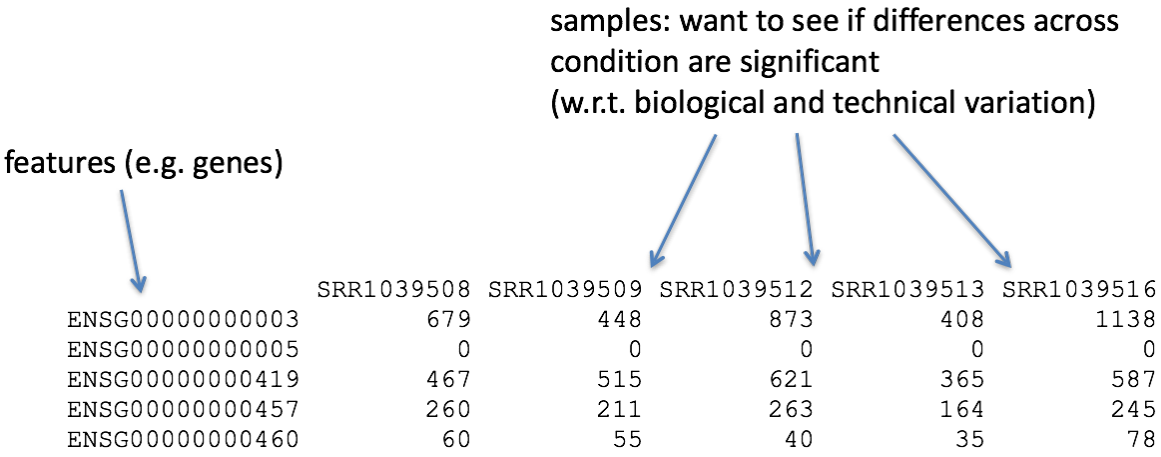
```
names(data)
```

```
## [1] "Mov10_kd_2" "Mov10_kd_3" "Mov10_oe_1" "Mov10_oe_2" "Mov10_oe_3" "Irrel_kd_1"
## [7] "Irrel_kd_2" "Irrel_kd_3"
```

You'll notice the colnames of the data are the sample names, and the rownames of the metadata are the sample names. This is important as we will be merging the metadata with the data based on the sample names.

## 1.3 Differential gene expression analysis overview

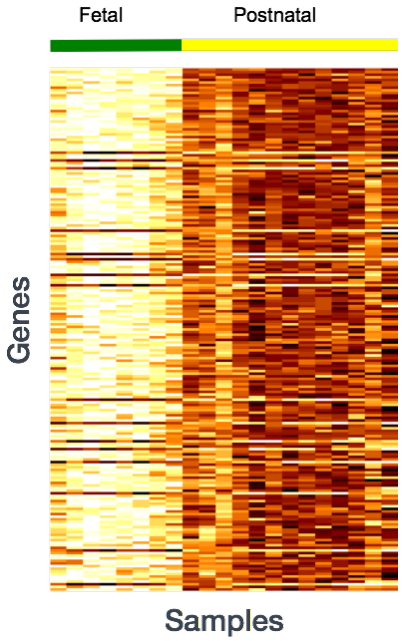
So what does this count data actually represent? The count data used for differential expression analysis represents the number of sequence reads that originated from a particular gene. The higher the number of counts, the more reads associated with that gene, and the assumption that there was a higher level of expression of that gene in the sample.



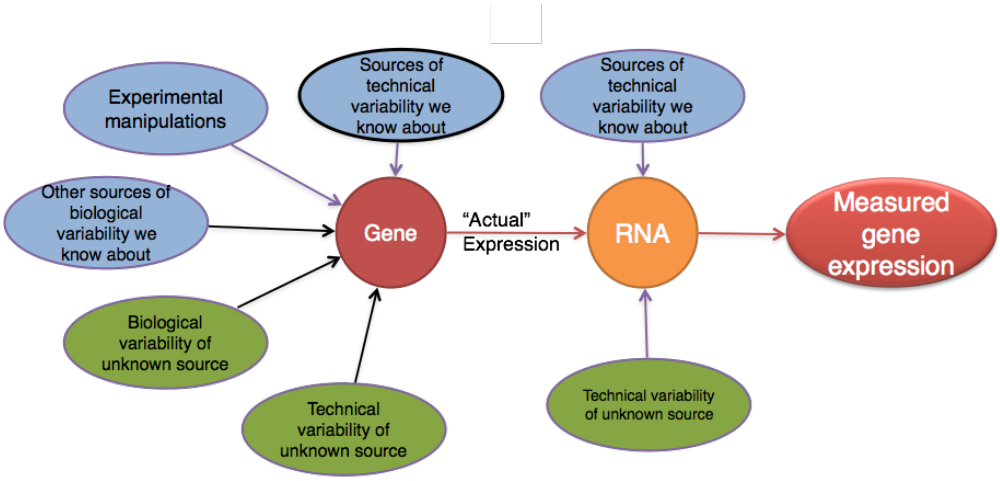

| | SRR1039508 | SRR1039509 | SRR1039512 | SRR1039513 | SRR1039516 |
|---|---|---|---|---|---|
| ENSG00000000003 | 679 | 448 | 873 | 408 | 1138 |
| ENSG00000000005 | 0 | 0 | 0 | 0 | 0 |
| ENSG00000000419 | 467 | 515 | 621 | 365 | 587 |
| ENSG00000000457 | 260 | 211 | 263 | 164 | 245 |
| ENSG00000000460 | 60 | 55 | 40 | 35 | 78 |

With differential expression analysis, we are looking for genes that change in expression between two or more groups (defined in the metadata) - case vs. control - correlation of expression with some variable or clinical outcome

**Why does it not work to identify differentially expressed gene by ranking the genes by how different they are between the two groups (based on fold change values)?**

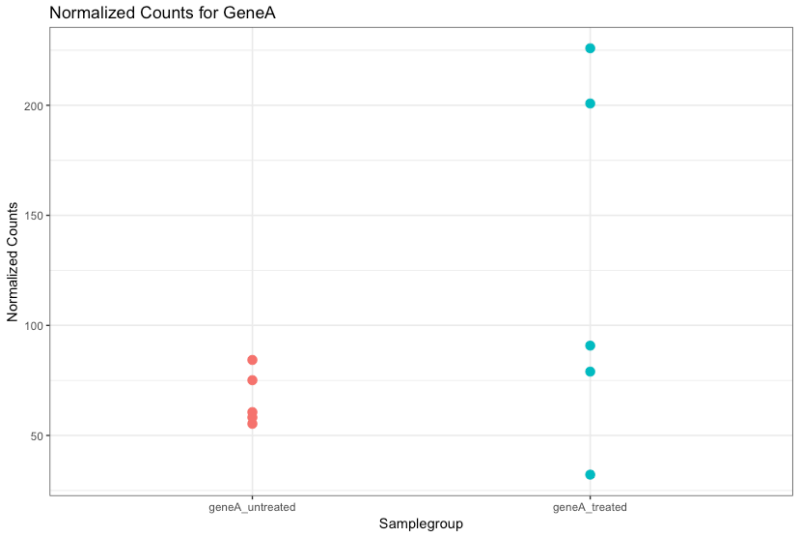



More often than not, there is much more going on with your data than what you are anticipating. Genes that vary in expression level between samples is a consequence of not only the experimental variables of interest but also due to extraneous sources. The goal of differential expression analysis to determine the relative role of these effects, and to separate the "interesting" from the "uninteresting".




Courtesy of Paul Pavlidis, UBC

The "uninteresting" presents as sources of variation in your data, and so even though the mean expression levels between sample groups may appear to be quite different, it is possible that the difference is not actually significant. This is illustrated for 'GeneA' expression between 'untreated' and 'treated' groups in the figure below. The mean expression level of geneA for the 'treated' group is twice as large as for the 'untreated' group, but the variation between replicates indicates that this may not be a significant difference. **We need to take into account the variation in the data (and where it might be coming from) when determining whether genes are differentially expressed.**

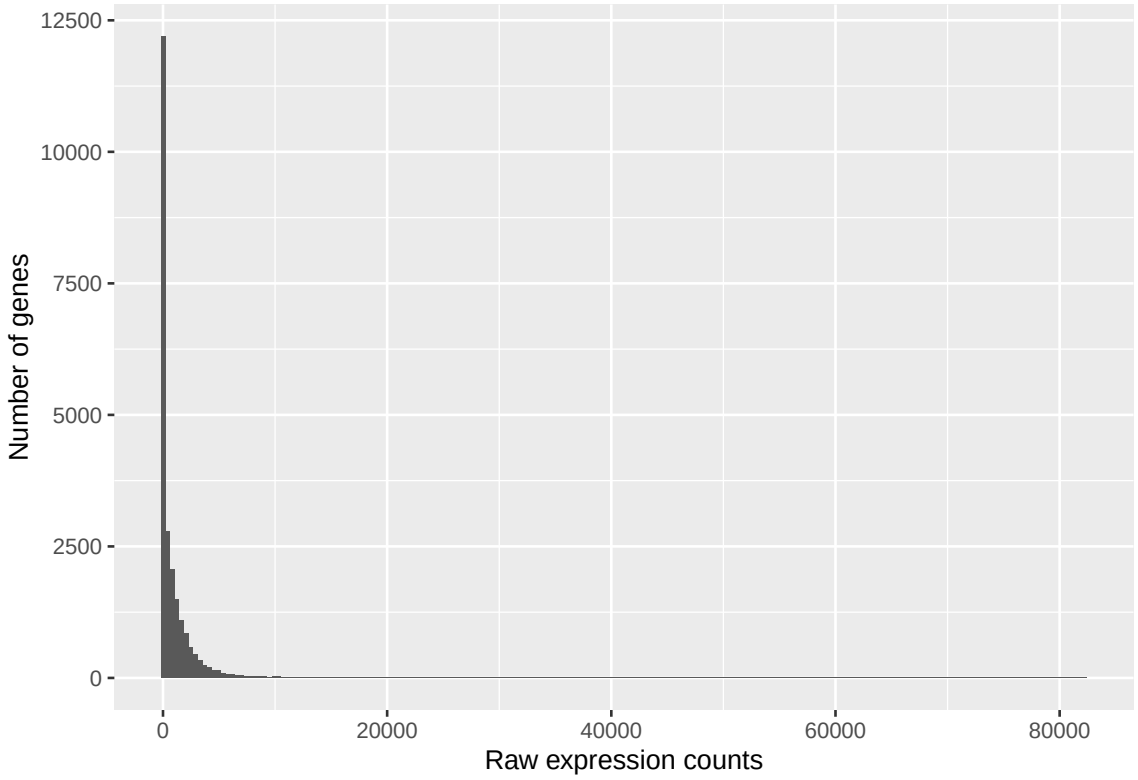



The goal of differential expression analysis is to determine, for each gene, whether the differences in expression (counts) **between groups** is significant given the amount of variation observed **within groups** (replicates). To test for significance, we need an appropriate statistical model that accurately performs normalization (to account for differences in sequencing depth, etc.) and variance modeling (to account for few numbers of replicates and large dynamic expression range).

### 1.3.1 RNA-seq count distribution

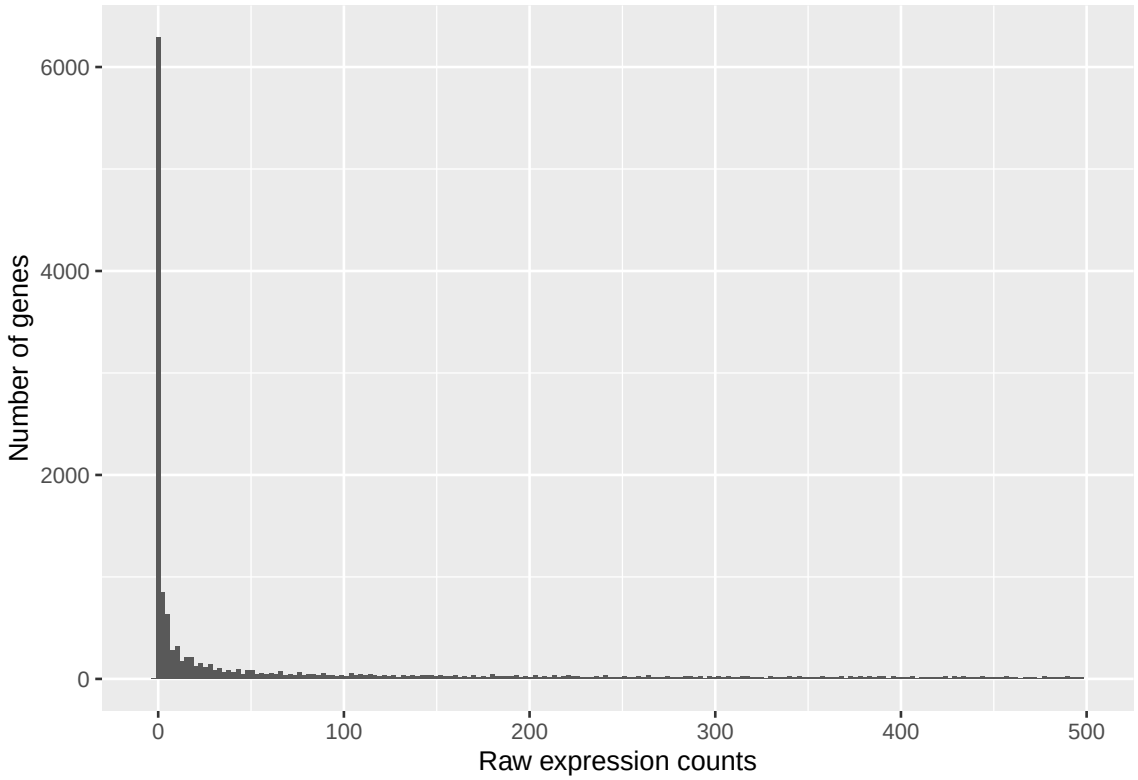
To determine the appropriate statistical model, we need information about the distribution of counts. To get an idea about how RNA-seq counts are distributed, let's plot the counts for a single sample, 'Mov10_oe_1':

```
ggplot(data) +
  geom_histogram(aes(x = Mov10_oe_1), stat = "bin", bins = 200) +
  xlab("Raw expression counts") +
  ylab("Number of genes")
```




If we zoom in close to zero, we can see a large number of genes with counts of zero:

```
ggplot(data) +
   geom_histogram(aes(x = Mov10_oe_1), stat = "bin", bins = 200) +
   xlim(-5, 500)  +
   xlab("Raw expression counts") +
   ylab("Number of genes")
```




These images illustrate some common features of RNA-seq count data, including a **low number of counts associated with a large proportion of genes**, and a long right tail due to the **lack of any upper limit for expression**.

### 1.3.2 Modeling count data

Count data is often modeled using the **binomial distribution**. The Binomial distribution is a common probability distribution that models the probability of obtaining one of two outcomes under a given number of parameters. It summarizes the number of successes in a fixed number of trials when each trial has two possible outcomes. For example, a coin toss is typically modeled using a binomial distribution, where the number of successes (e.g., heads) in a fixed number of trials (e.g., 10 coin tosses) when each trial has two possible outcomes (e.g., heads or tails) and the probability of success is constant. However, not all count data can be fit with the binomial distribution. The binomial is based on discrete events and used in situations when you have a certain number of cases.

The **binomial distribution** is used to model scenarios where there are two possible outcomes, such as "success" or "failure," over a fixed number of trials. For example, flipping a coin 10 times and counting the number of "heads" out of those 10 flips follows a binomial distribution with a probability of success ($p$) equal to 0.5.

When **the number of cases is very large (i.e. people who buy lottery tickets), but the probability of an event is very small (probability of winning)**, the **Poisson distribution** is used to model these types of count data.

**Poisson Distribution Formula**

$$P(X = x) = \frac{\lambda^x e^{-\lambda}}{x!}$$

where
$x$ = 0, 1, 2, 3, ...
$\lambda$ = mean number of occurrences in the interval
$e$ = Euler's constant ≈ 2.71828

**With RNA-seq data, for each sample we have millions of reads being sequenced and the probability of a read mapping to a gene is extremely low.** Thus, it would be an appropriate situation to use the Poisson distribution. However, a unique property of this distribution is that the mean == variance given by the single parameter $\lambda$.

For us to apply a poisson distribution to our data, we first need to find out whether our data fulfills the criteria to use the poisson distribution. To do that we can plot the *mean versus variance* for the 'Mov10 overexpression' replicates:

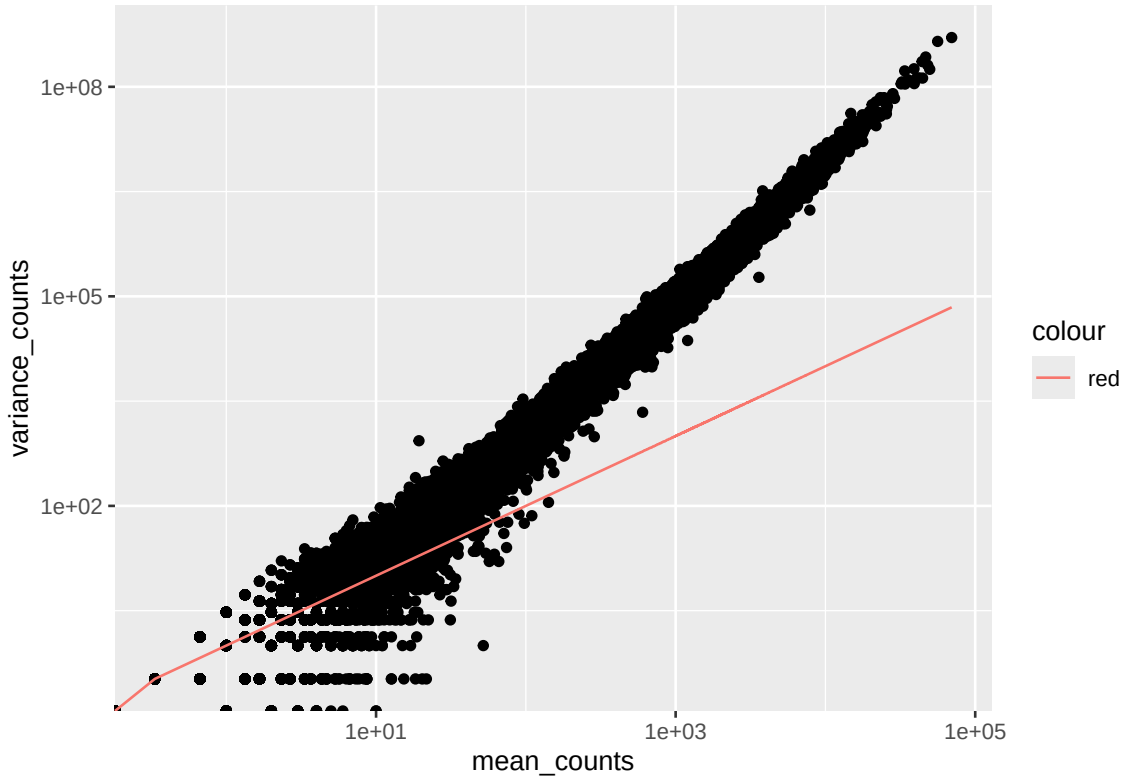
To calculate the mean and variance of our data, we will use the `apply` function. The `apply` function allows you to apply a function to the margins (rows or columns) of a matrix. The syntax for `apply` is as follows: `apply(X, MARGIN, FUN)`, where `X` is a matrix, `MARGIN` is the margin of the matrix to apply the function to (1 = rows, 2 = columns), and `FUN` is the function to apply.

We will use `MARGIN = 1` to apply the function mean and variance of the counts for each row (gene) across the 'Mov10 overexpression' replicates. We will then create a data frame with the mean and variance of the counts for each gene.

```
# calculate the mean and variance of the counts for each gene
mean_counts <- apply(data[, 3:5], 1, mean)

variance_counts <- apply(data[, 3:5], 1, var)

# for ggplot we need the data to be in a data.frame
df <- data.frame(mean_counts, variance_counts)
```

Run the following code to plot the *mean versus variance* for the 'Mov10 overexpression' replicates:

```
ggplot(df) +
  geom_point(aes(x = mean_counts, y = variance_counts)) +
  geom_line(aes(x = mean_counts, y = mean_counts, color="red")) +
  scale_y_log10() + scale_x_log10()
```



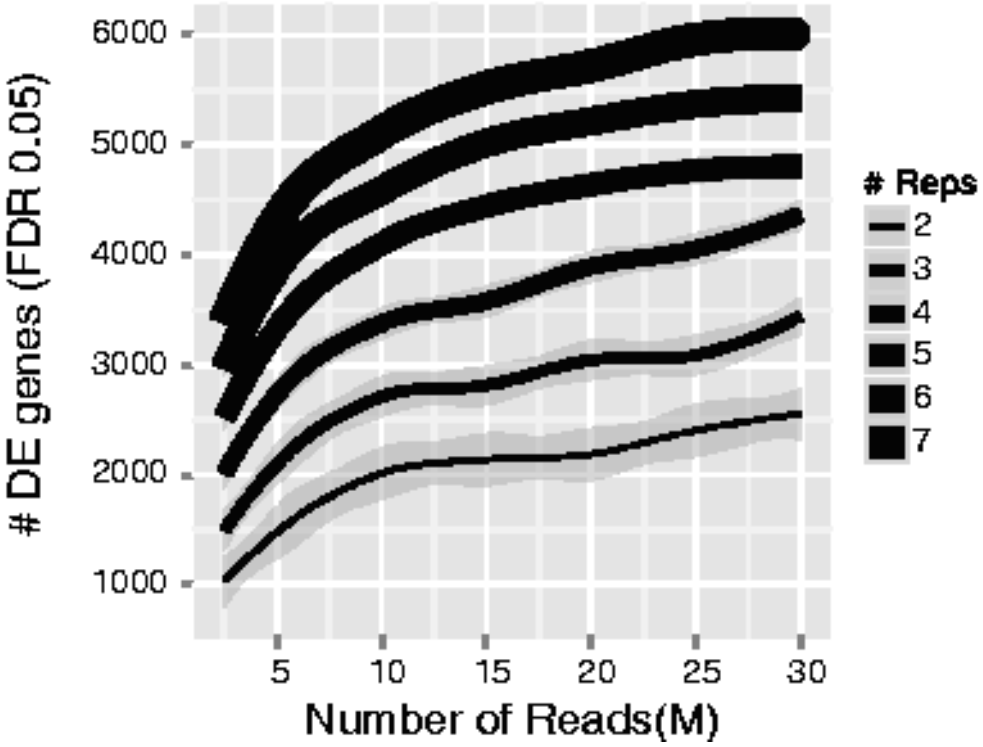

By plotting the *mean versus the variance* of our data we can easily see that the **variance > mean** and therefore it does not fit the Poisson distribution. Genes having higher mean counts have even higher variance. Also for gene having low mean counts, there is a scatter of points and we can see that there is variability even in the variance. To account for this extra variance we need a new model.

The model that fits best, given this type of variability between replicates, is the **Negative Binomial (NB) model**. Essentially, **the NB model** is a good approximation for data where the **variance > mean**, as is the case with RNA-seq count data.

### 1.3.3 Improving mean estimates (i.e. reducing variance) with biological replicates

The variance or scatter tends to reduce as we increase the number of biological replicates (*the distribution will approach the Poisson distribution with increasing numbers of replicates*). **The value of additional replicates is that as you add more data (replicates), you get increasingly precise estimates of group means, and ultimately greater confidence in the ability to distinguish differences between sample classes (i.e. more DE genes).**

The figure below shows how the number of differentially expressed (DE) genes identified is influenced by sequencing depth and number of replicates [1]. Notice that increasing the number of replicates identifies more DE genes compared to increasing sequencing depth. This is because most of the biological variability occurs between samples. As a result, having more replicates typically yields better results than increasing sequencing depth, as it captures natural biological variability more effectively and leads to more accurate estimates of within-group variation. While increasing sequencing depth reduces technical noise, it has less impact on detecting DE genes since it doesn't account for biological variability. Generally, a minimum sequencing depth of 20-30 million reads per sample is recommended, though successful RNA-seq experiments have been conducted with as few as 10 million reads when enough replicates are included.



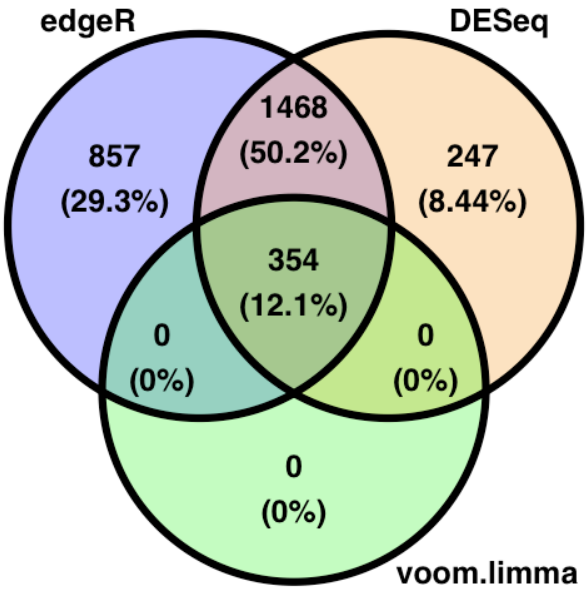

**NOTE**

- **Biological replicates** represent multiple samples (i.e. RNA from different mice) representing the same sample class
- **Technical replicates** represent the same sample (i.e. RNA from the same mouse) but with technical steps replicated
- Usually biological variance is much greater than technical variance, so we do not need to account for technical variance to identify biological differences in expression
- **Don't spend money on technical replicates - biological replicates are much more useful**

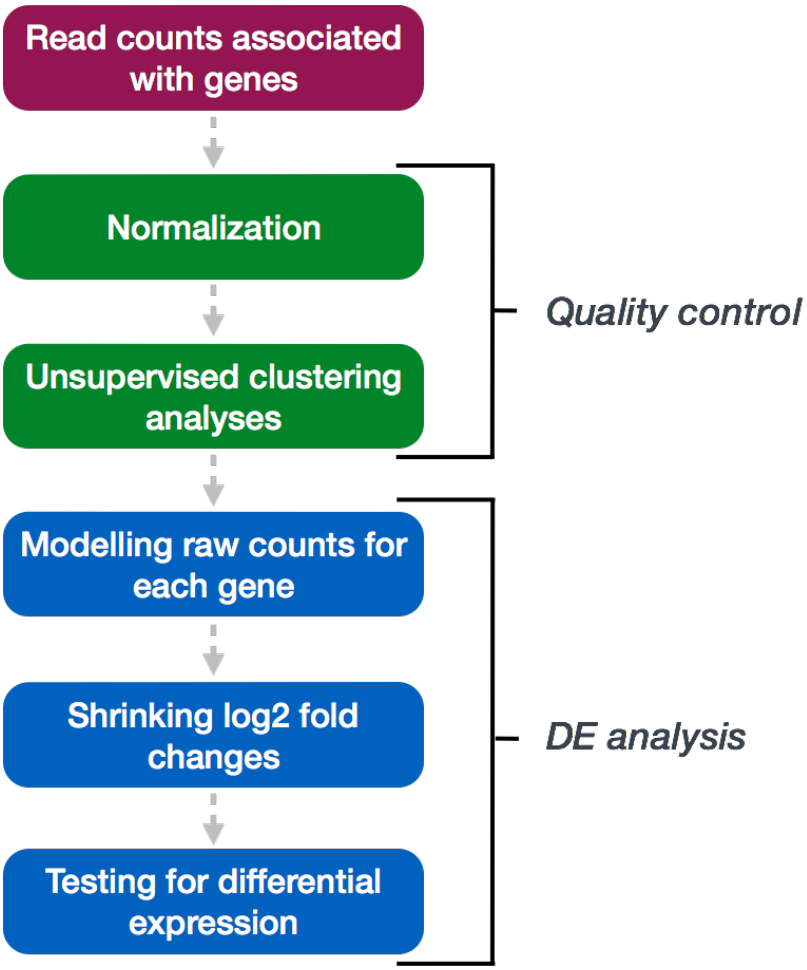
### 1.3.4 Differential expression analysis workflow

To model counts appropriately when performing a differential expression analysis, there are a number of software packages that have been developed for differential expression analysis of RNA-seq data. Even as new methods are continuously being developed a few tools are generally recommended as best practice, e.g. **DESeq2** and **EdgeR**. Both these tools use the negative binomial model, employ similar methods, and typically, yield similar results. They are pretty stringent, and have a good balance between sensitivity and specificity (reducing both false positives and false negatives).

Here is a comparison of the three most highy used software packages for differential expression analysis. No single method is optimal under all circumstances, for example, limma works best when sample number is high, and edgeR and DESeq2 perform well for small sample sizes. It is also difficult to compare analysis methods due to different procedures in calculating pvalues.




**We will be using DESeq2 for the DE analysis, and the analysis steps with DESeq2 are shown in the flowchart below**. DESeq2 first normalizes the count data to account for differences in library sizes and RNA composition between samples. Then, we will use the normalized counts to make some plots for QC at the gene and sample level. The final step is to use the appropriate functions from the DESeq2 package to perform the differential expression analysis.




We will go in-depth into each of these steps in the following lessons, but additional details and helpful suggestions regarding DESeq2 can be found in the DESeq2 vignette. As you go through this workflow and questions arise, you can reference the vignette from within RStudio:

```
vignette("DESeq2")
```

This is very convenient, as it provides a wealth of information at your fingertips! Be sure to use this as you need during the workshop.

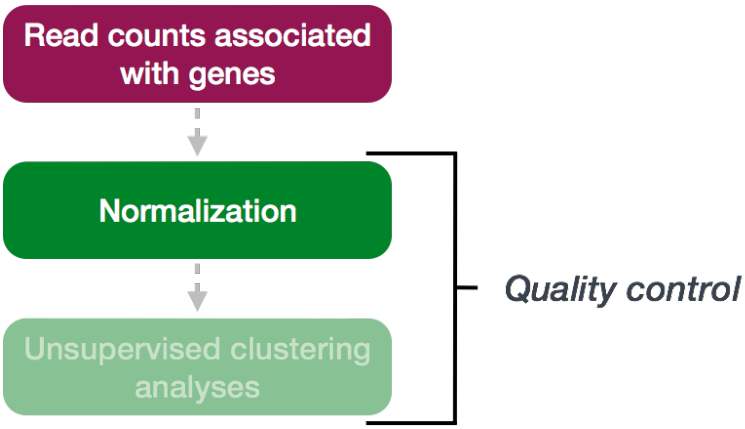
---

# Chapter 2

# Count normalization

## 2.1 Normalization

The first step in the DE analysis workflow is count normalization, which is necessary to make accurate comparisons of gene expression between samples.

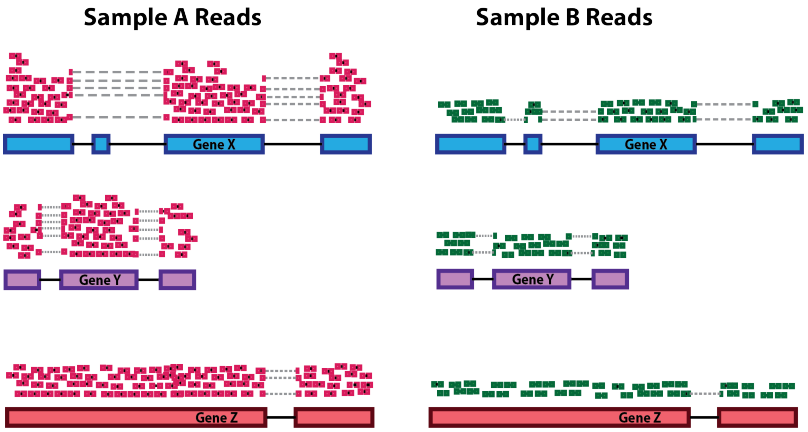



The counts of mapped reads for each gene is proportional to the expression of RNA ("interesting") in addition to many other factors ("uninteresting"). Normalization is the process of scaling raw count values to account for the "uninteresting" factors. In this way the expression levels are more comparable between and/or within samples.

The main factors often considered during normalization are:

- **Sequencing depth:**

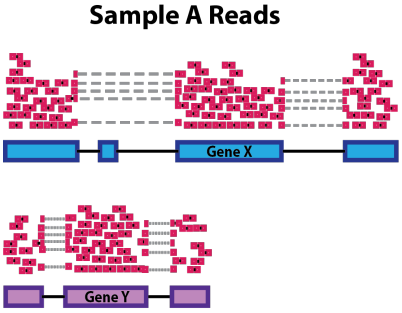
Accounting for sequencing depth is necessary for comparison of gene expression between samples. In the example below, each gene appears to have doubled in expression in *Sample A* relative to *Sample B*, however this is a consequence of *Sample A* having double the sequencing depth.




***NOTE:*** *In the figure above, each pink and green rectangle represents a read aligned to a gene. Reads connected by dashed lines connect a read spanning an intron.*

- **Gene length:**

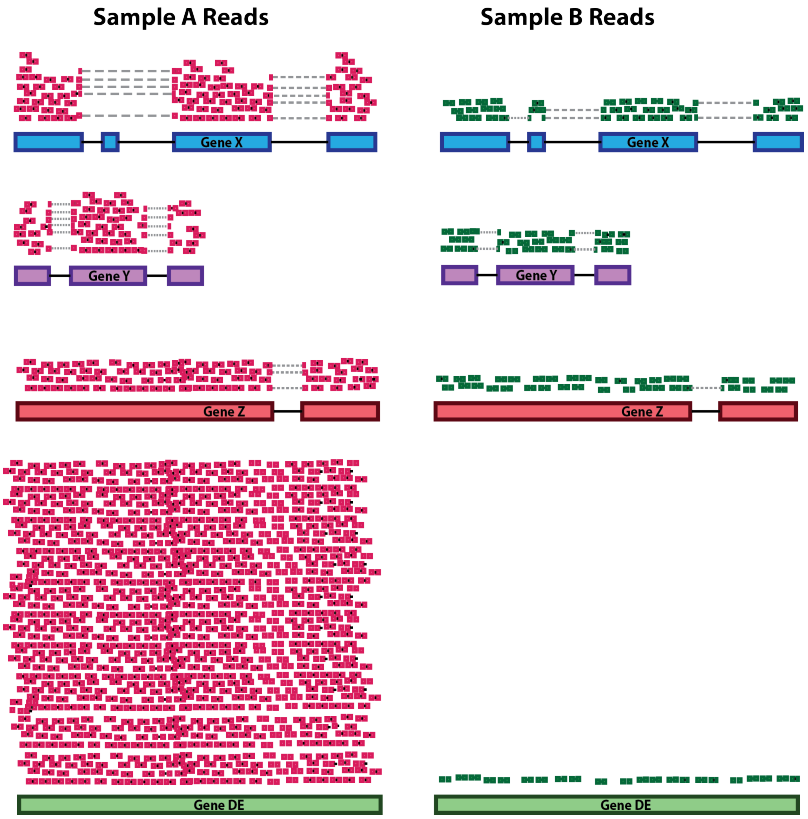
Accounting for gene length is necessary for comparing expression between different genes within the same sample. In the example, *Gene X* and *Gene Y* have similar levels of expression, but the number of reads mapped to *Gene X* would be many more than the number mapped to *Gene Y* because *Gene X* is longer.




- **RNA composition:**

RNA composition bias arises when a few highly expressed genes dominate the overall expression profile of a sample. Accounting for RNA composition is recommended for accurate comparison of expression between samples, and is particularly important when performing differential expression analyses [1].

In the example, if we were to divide each sample by the total number of counts to normalize, the counts would be greatly skewed by the DE gene, which takes up most of the counts for *Sample A*, but not *Sample B*. Most other genes for *Sample A* would be divided by the larger number of total counts and appear to be less expressed than those same genes in *Sample B*.




***While normalization is essential for differential expression analyses, it is also necessary for exploratory data analysis, visualization of data, and whenever you are exploring or comparing counts between or within samples.***

### 2.1.1 Common normalization methods

Several common normalization methods exist to account for these differences:

TPM is very similar to RPKM and FPKM. The only difference is the order of operations. Here's how you calculate TPM:

For example, here's how you would calculate TPM for a gene in a sample:

1. Divide the read counts by the length of each gene in kilobases. This gives you reads per kilobase (RPK).

$$RPK = \frac{reads}{genelength(kb)}$$

2. Count up all the RPK values in a sample and divide this number by 1,000,000. This is your "per million" scaling factor.

$$\text{Per million scaling factor} = \frac{\sum RPK}{1,000,000}$$

3. Divide the RPK values by the "per million" scaling factor. This gives you TPM.

$$TPM = \frac{RPK}{\text{Per million scaling factor}}$$

The key idea here is that **TPM** normalizes for both **gene length** and **sequencing depth**, making it easier to compare the **relative abundance** of genes across different samples, regardless of how many total reads were generated.

| Normalization method | Description | Accounted factors | Recommendations for use |
|---|---|---|---|
| **TPM** (transcripts per kilobase million) | counts per length of transcript (kb) per million reads mapped | sequencing depth and gene length | gene count comparisons within a sample or between samples of the same sample group; **NOT for DE analysis** |
| **RPKM/FPKM** (reads/fragments per kilobase of exon per million reads/fragments mapped) | similar to TPM | sequencing depth and gene length | gene count comparisons between genes within a sample; **NOT for between sample comparisons or DE analysis** |
| DESeq2's **median of ratios** [1] | counts divided by sample-specific size factors determined by median ratio of gene counts relative to geometric mean per gene | sequencing depth and RNA composition | gene count comparisons between samples and for **DE analysis**; **NOT for within sample comparisons** |
| EdgeR's **trimmed mean of M values (TMM)** [2] | uses a weighted trimmed mean of the log expression ratios between samples | sequencing depth, RNA composition | gene count comparisons between samples and for **DE analysis**; **NOT for within sample comparisons** |

### 2.1.2 TPM (Not Recommended)

**Step 1: Calculate Reads Per Kilobase (RPK)**

To calculate RPK for each gene in each sample, divide the read counts by the gene length in kilobases.

| Gene | Sample 1 (reads) | Sample 2 (reads | Length |
|---|---|---|---|
| Gene A | 300 | 600 | 1 kb |
| Gene B | 500 | 400 | 2 kb |

**Sample 1:** - Gene A RPK = 300 reads / 1 kb = 300 - Gene B RPK = 500 reads / 2 kb = 250

**Sample 2:** - Gene A RPK = 600 reads / 1 kb = 600 - Gene B RPK = 400 reads / 2 kb = 200

| Gene | Sample 1 (RPK) | Sample 2 (RPK) |
|---|---|---|
| Gene A | 300 | 600 |
| Gene B | 250 | 200 |

**Step 2: Calculate the Per-Million Scaling Factor**

Sum the RPK values in each sample and divide by 1,000,000.

**Sample 1:** - Total RPK = 300 + 250 = 550 - Scaling Factor = 550 / 1,000,000 = 0.00055

**Sample 2:** - Total RPK = 600 + 200 = 800 - Scaling Factor = 800 / 1,000,000 = 0.0008

**Step 3: Calculate TPM**

Now, divide each gene's RPK by the per-million scaling factor to get the TPM.

**Sample 1:** - Gene A TPM = 300 / 0.00055 = 545,455 - Gene B TPM = 250 / 0.00055 = 454,545

**Sample 2:** - Gene A TPM = 600 / 0.0008 = 750,000 - Gene B TPM = 200 / 0.0008 = 250,000

| Gene | Sample 1 (TPM) | Sample 2 (TPM) |
|---|---|---|
| Gene A | 545,455 | 750,000 |
| Gene B | 454,545 | 250,000 |

TPM normalizes read counts by the total reads in a sample, allowing comparisons between genes while accounting for gene length. However, it does not account for the variability inherent in count data or overdispersion, making statistical testing on these values unreliable.

### 2.1.3 RPKM/FPKM (Not Recommended)

Using RPKM/FPKM normalization means the total normalized counts will vary between samples, making direct comparisons between genes unreliable.

| Gene | Sample A | Sample B |
|---|---|---|
| XCR1 | 5.5 | 5.5 |
| WASHC1 | 73.4 | 21.8 |
| ... | ... | ... |
| Total RPKM | 1,000,000 | 1,500,000 |

For example, even though both samples show XCR1 as 5.5, they cannot be directly compared because their total counts differ.

**Conclusion:** Tools like DESeq2 or edgeR are better suited for differential expression analysis, as they:

- Use raw counts for more accurate modeling.
- Apply statistical modeling (e.g., Negative Binomial) to account for variance across replicates.

### 2.1.4 DESeq2 Normalization: Median of Ratios Method

For differential expression analysis, gene length need not be factored in; however, **sequencing depth** and **RNA composition** must be considered. The median of ratios method neutralizes these differences.

This method involves several steps, even though from the user-end it appears as a single function call in DESeq2.

**Step 1: Create a Pseudo-reference Sample (Row-wise Geometric Mean)**

The **geometric mean** of a set of $n$ positive numbers $x_1, x_2, \ldots, x_n$ is defined as:

$$\text{Geometric Mean} = \left( \prod_{i=1}^{n} x_i \right)^{\frac{1}{n}}$$

For each gene, a pseudo-reference sample is created that is equal to the geometric mean across all samples, reducing the impact of extreme values.

| Gene | Sample A | Sample B | Pseudo-reference Sample |
|---|---|---|---|
| EF2A | 1489 | 906 | sqrt(1489 * 906) = **1161.5** |
| ABCD1 | 22 | 13 | sqrt(22 * 13) = **17.7** |
| ... | ... | ... | ... |

**Step 2: Calculate the Ratio of Each Sample to the Reference**

For every gene in a sample, calculate the ratio of the sample count to the reference (sample/ref). This is performed for each sample in the dataset, resulting in ratios for most genes that are similar, as depicted below:

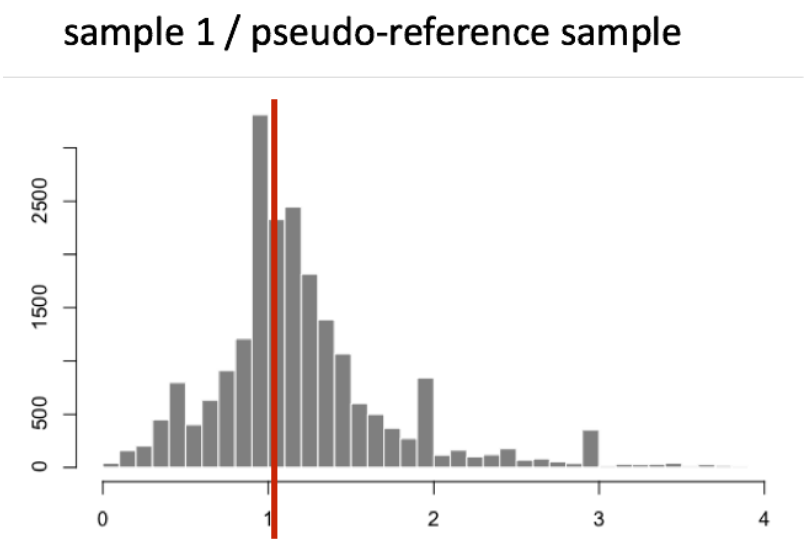
| Gene | Sample A | Sample B | Pseudo-reference Sample | Ratio of Sample A/Ref | Ratio of Sample B/Ref |
|---|---|---|---|---|---|
| EF2A | 1489 | 906 | 1161.5 | 1489/1161.5 = **1.28** | 906/1161.5 = **0.78** |
| ABCD1 | 22 | 13 | 17.7 | 22/17.7 = **1.24** | 13/17.7 = **0.73** |
| MEFV | 793 | 410 | 570.2 | 793/570.2 = **1.39** | 410/570.2 = **0.72** |
| ... | ... | ... | ... | ... | ... |

**Step 3: Calculate the Normalization Factor for Each Sample (Size Factor)**

The median value of all ratios for a given sample is calculated as the normalization factor (size factor):

```
normalization_factor_sampleA <- median(c(1.28, 1.24, 1.39, 1.35, 0.59))
normalization_factor_sampleB <- median(c(0.78, 0.73, 0.72, 0.74, 1.35))
```

The figure below illustrates the distribution of values for the ratios across all genes in a single sample.




This method is robust as it assumes that not all genes are differentially expressed, allowing normalization factors to account for both sequencing depth and RNA composition efficiently.

**Step 4: Calculate the Normalized Count Values Using the Normalization Factor**

Each raw count value in a given sample is divided by that sample's normalization factor to generate normalized count values across all genes:

| Gene | Sample A | Sample B |
|---|---|---|
| EF2A | 1489 / 1.3 = **1145.39** | 906 / 0.77 = **1176.62** |
| ABCD1 | 22 / 1.3 = **16.92** | 13 / 0.77 = **16.88** |
| ... | ... | ... |

Note that normalized count values may not always be whole numbers.

### 2.1.5 Summary of Key Reasons for Using DESeq2

- **Median of Ratios**: This method accounts for sequencing depth and RNA composition, making it more appropriate for RNA-Seq data analysis.
- **Geometric Mean**: It creates a pseudo-reference that reduces sensitivity to extreme values, affording more stable estimates.
- **Normalization Procedures**: It ensures reliable comparisons of gene expression across samples by mitigating bias from technical variations, leading to more accurate differential expression analysis.

This combination of methodologies allows DESeq2 to provide robust results, enhancing the identification of differentially expressed genes in complex biological datasets.

**Exercise**

Determine the normalized counts for your gene of interest, PD1, given the raw counts and size factors below.

NOTE: You will need to run the code below to generate the raw counts dataframe (PD1) and the size factor vector (size_factors), then use these objects to determine the normalized counts values:

```
# Raw counts for PD1
PD1 <- c(21, 58, 17, 97, 83, 10)
names(PD1) <- paste0("Sample", 1:6)
PD1 <- data.frame(PD1)
PD1 <- t(PD1)

# Size factors for each sample
size_factors <- c(1.32, 0.70, 1.04, 1.27, 1.11, 0.85)
```

```
# Your code here
```

---

## 2.2 Count normalization of Mov10

Now that we know the theory of count normalization, we will normalize the counts for the Mov10 dataset using DESeq2. This requires a few steps:

1. Ensure the row names of the metadata dataframe are present and in the same order as the column names of the counts dataframe.
2. Create a `DESeqDataSet` object
3. Generate the normalized counts

### 2.2.1 1. Match the metadata and counts data

We should always make sure that we have sample names that match between the two files, and that the samples are in the right order. DESeq2 will output an error if this is not the case.

```
### Check that sample names match in both files
all(colnames(data) %in% rownames(meta))
```

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

```
all(colnames(data) == rownames(meta))
```

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

The colnames of our data don't match the rownames of our metadata so we need to reorder them. We can use the `match` function:

```
idx <- match(rownames(meta),colnames(data))
data <- data[,idx]

all(colnames(data) == rownames(meta))
```

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

**Exercise** points = +2

Suppose we had sample names matching in the counts matrix and metadata file, but they were out of order. Write the line(s) of code required to create a new matrix with columns ordered such that they were identical to the row names of the metadata.

```
# write the code to reorder the columns of the 'counts' matrix to match the rownames of the metadata

# check that the rownames of the metadata match the colnames of the 'counts' matrix
```

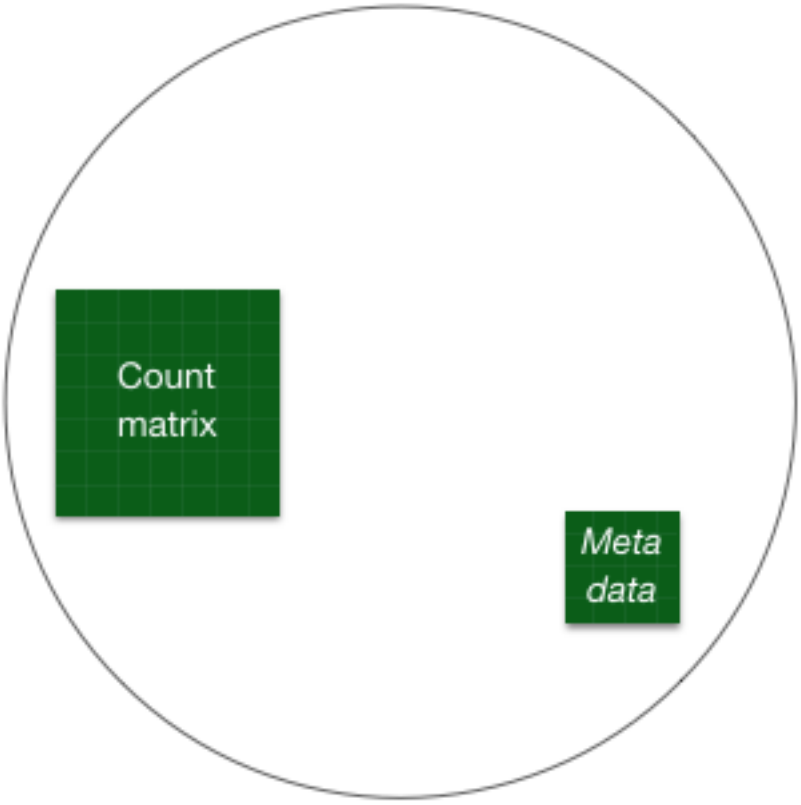
---

### 2.2.2 2. Create DESEq2 object

Bioconductor software packages often define and use a custom class within R for storing data (input data, intermediate data and also result data). These custom data structures are similar to `lists` in that they can contain multiple different data types/structures within them. But, unlike lists they have pre-specified `data slots`, which hold specific types/classes of data. The data stored in these pre-specified slots can be accessed by using specific package-defined functions.

Let's start by creating the `DESeqDataSet` object and then we can talk a bit more about what is stored inside it. To create the object we will need the **count matrix** and the **metadata** table as input. We will also need to specify a **design formula**. The design formula specifies the column(s) in the metadata table and how they should be used in the analysis. For our dataset we only have one column we are interested in, that is `~sampletype`. This column has three factor levels, which tells DESeq2 that for each gene we want to evaluate gene expression change with respect to these different levels.

```
## Create DESeq2Dataset object
dds <- DESeqDataSetFromMatrix(countData = data, colData = meta, design = ~ sampletype)
```




You can use DESeq-specific functions to access the different slots and retrieve information, if you wish. For example, suppose we wanted the original count matrix we would use `counts()`:

```
head(counts(dds[,1:5]))
```

```
##             Irrel_kd_1 Irrel_kd_2 Irrel_kd_3 Mov10_kd_2 Mov10_kd_3
## 1/2-SBSRNA4         45         31         39         57         41
## A1BG                77         58         40         71         40
## A1BG-AS1           213        172        126        256        177
## A1CF                 0          0          0          0          1
## A2LD1               91         80         50        146         81
## A2M                  9          8          4         10          9
```

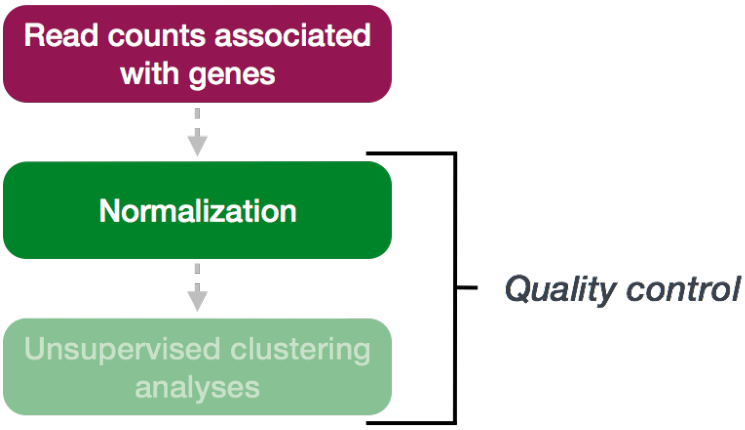
As we go through the workflow we will use the relevant functions to check what information gets stored inside our object. We can also run:

```
slotNames(dds)
```

```
## [1] "design"             "dispersionFunction" "rowRanges"
## [4] "colData"            "assays"             "NAMES"
## [7] "elementMetadata"    "metadata"
```

### 2.2.3 3. Generate the Mov10 normalized counts

The next step is to normalize the count data in order to be able to make fair gene comparisons between samples.




To perform the **median of ratios method** of normalization, DESeq2 has a single `estimateSizeFactors()` function that will generate size factors for us. We will use the function in the example below, but **in a typical RNA-seq analysis this step is automatically performed by the `DESeq()` function**, which we will see later.

```
dds <- estimateSizeFactors(dds)
```

By assigning the results back to the `dds` object we are filling in the slots of the `DESeqDataSet` object with the appropriate information. We can take a look at the normalization factor applied to each sample using:

```
sizeFactors(dds)
```

```
## Irrel_kd_1 Irrel_kd_2 Irrel_kd_3 Mov10_kd_2 Mov10_kd_3 Mov10_oe_1 Mov10_oe_2
##  1.1224020  0.9625632  0.7477715  1.5646728  0.9351760  1.2016082  1.1205912
## Mov10_oe_3
##  0.6534987
```

Now, to retrieve the normalized counts matrix from `dds`, we use the `counts()` function and add the argument `normalized=TRUE`.

```
normalized_counts <- counts(dds, normalized=TRUE)
```

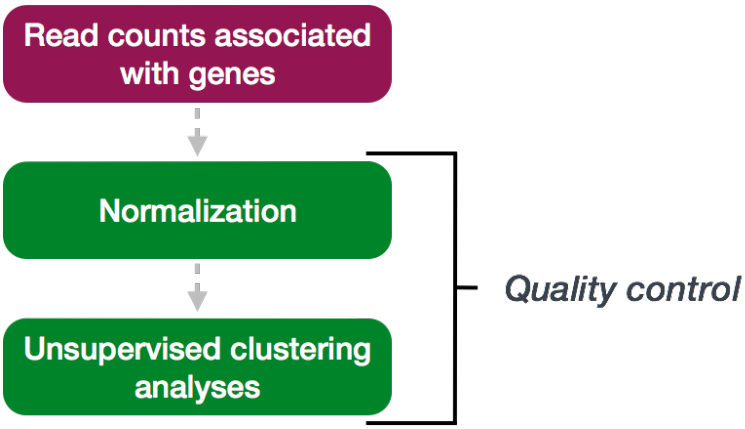
We can save this normalized data matrix to file for later use:

```
write.table(normalized_counts, file="data/normalized_counts.txt", sep="\t", quote=F, col.names=NA)
```

# Chapter 3

# Quality Control

The next step in the DESeq2 workflow is QC, which includes sample-level and gene-level steps to perform QC checks on the count data to help us ensure that the samples/replicates look good.

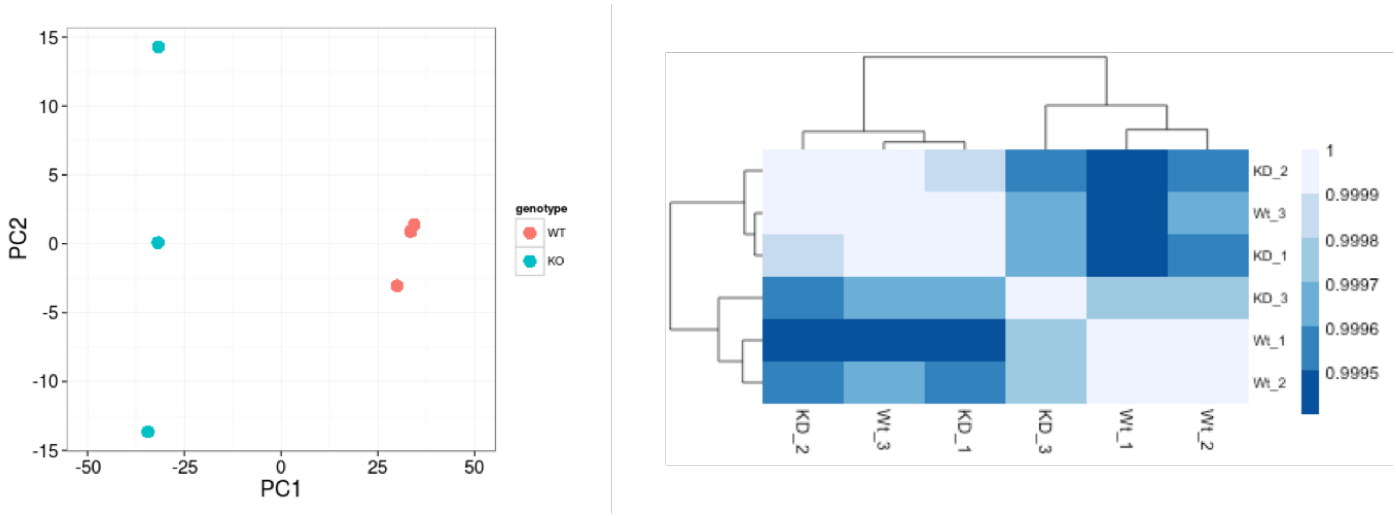



## 3.1 Sample-level QC

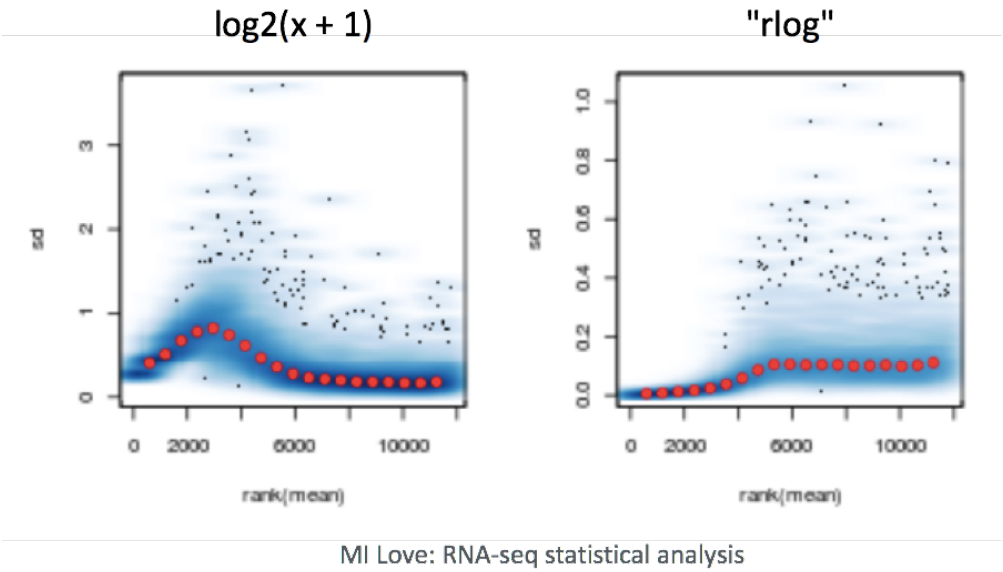
A useful initial step in an RNA-seq analysis is often to assess overall similarity between samples:

- Which samples are similar to each other, which are different?
- Does this fit to the expectation from the experiment's design?
- What are the major sources of variation in the dataset?

To explore the similarity of our samples, we will be performing sample-level QC using Principal Component Analysis (PCA) and hierarchical clustering methods. Our sample-level QC allows us to see how well our replicates cluster together, as well as, observe whether our experimental condition represents the major source of variation in the data. Performing sample-level QC can also identify any sample outliers, which may need to be explored further to determine whether they need to be removed prior to DE analysis.

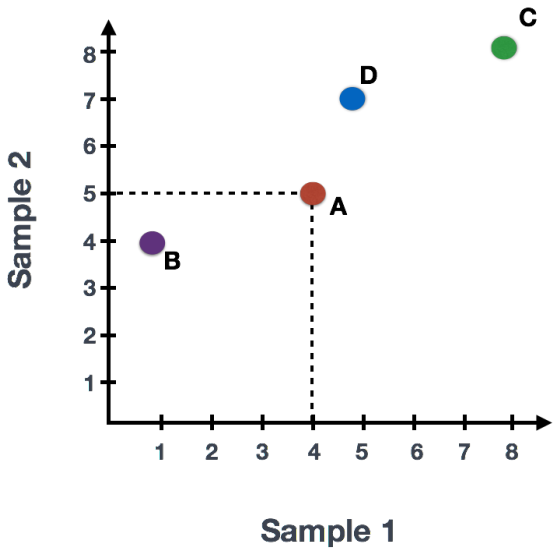



When using these unsupervised clustering methods, log2-transformation of the normalized counts improves the distances/-clustering for visualization. DESeq2 uses a **regularized log transform** (rlog) of the normalized counts for sample-level QC as it moderates the variance across the mean, improving the clustering.



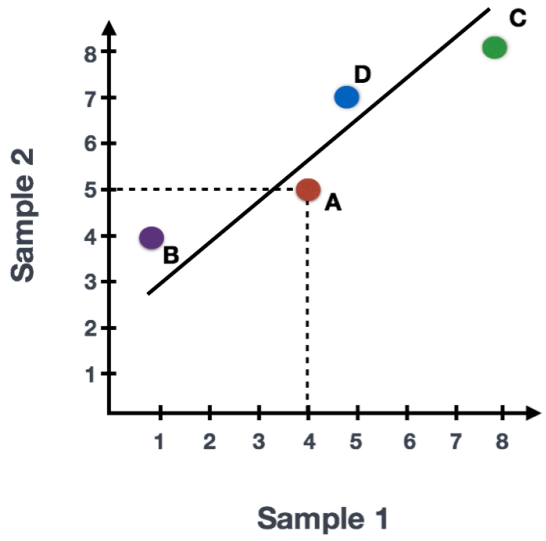

Principal Component Analysis (PCA) is a technique used to emphasize variation and highlight strong patterns in a dataset. It is particularly useful for dimensionality reduction, meaning it simplifies large datasets by identifying the directions (or principal components) that explain the greatest variation in the data. For a more detailed introduction to PCA, we recommend watching this StatQuest's video.

Suppose we have a dataset with two samples and four genes. To evaluate the relationship between these samples, we can plot the counts of one sample against the other, with Sample 1 on the x-axis and Sample 2 on the y-axis, as shown below:




| | Sample 1 | Sample 2 |
|---|---|---|
| Gene A | 4 | 5 |
| Gene B | 1 | 4 |
| Gene C | 8 | 8 |
| Gene D | 5 | 7 |

For PCA analysis, the first step is finding the **first principal component** (PC1). The **first principal component** is the direction that **maximizes the variance** (the sum of squared distances from the mean) and simultaneously **minimizes the squared distances between the data points** and their projections onto that line. Why? Because variance is essentially the spread (or squared distances) of the data from its mean, and PCA wants to capture as much of this spread as possible. In this example, the most variation is along the diagonal. That is, the **largest spread in the data** is between the two endpoints of this line. **This is called the first principal component, or PC1.** The genes at the endpoints of this line (Gene B and Gene C) have the **greatest influence** on the direction of this line.




After determining PC1, PCA assigns each sample a **PC1 score**, which represents how each sample aligns with this principal component. The score for each sample is calculated by taking the product of each gene's influence on PC1 and its normalized expression in the sample, and summing these products across all genes. A second line can then be drawn to represent the **second principal component (PC2)**, which captures the second-largest amount of variation in the data. This process can continue for additional principal components.

The formula to calculate a sample's PC1 score is as follows:

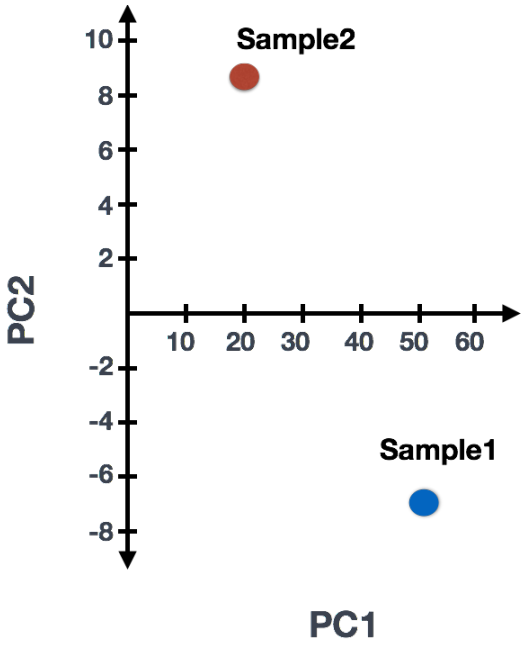
$$\text{Sample1 PC1 score} = \sum_{i=1}^{n} (\text{counts}_i \times \text{influence}_i)$$

Where:

- $\text{counts}_i$: Normalized expression of gene $i$ in the sample
- $\text{influence}_i$: Influence of gene $i$ on PC1

While calculating the **influence** of each gene on PC1 is complex, a key step involves calculating a z-score for each gene.

A z-score measures how far a gene's expression deviates from the mean, and genes with larger z-scores (like Gene B and Gene C in this example) have a greater influence on PC1 because they contribute more to the variation.




| | PC1 | PC2 |
|---|---|---|
| Sample1 | 51 | -7 |
| Sample2 | 21 | 8.5 |

Briefly, calculating PCA and the influence of each gene on PC1 involves the following steps:

#### 3.1.0.1 Steps to calculate PCA

1. **Z-Scores**:

- A z-score transforms the data points, indicating how far a value is from the mean in terms of standard deviations. This standardization adjusts for differences in scale between different genes, allowing for more accurate comparisons.
- The formula for calculating a z-score for a value $x$ of a gene is:

$$z = \frac{x - \mu}{\sigma}$$

Where:

- $\mu$ is the mean expression of the gene across samples.
- $\sigma$ is the standard deviation of the gene expression.

2. **Covariance matrix**:

- Covariance measures the degree to which two variables change together. In the context of PCA, we calculate the covariance matrix for standardized (z-scored) gene expressions to understand how genes vary together.
- A positive covariance indicates that higher values of one gene correspond to higher values of another, while a negative covariance implies that higher values of one gene correspond to lower values of another.
- After standardizing the data, calculate the covariance between the z-scores of different genes. This will produce a covariance matrix that reveals how gene expressions vary together.
- In the context of Principal Component Analysis (PCA), the covariance matrix is a crucial component. It captures how variables in the dataset change with respect to one another and is used to determine the principal components.

3. **Eigenvectors and Eigenvalues:** Find these from the covariance matrix. They reveal the directions (principal components) and their importance (variance explained) in the data. Think of PCA as creating new "axes" or dimensions that capture the most variation in the data. PCA will identify the direction of these axes based on the covariance of the standardized gene expressions that better highlight differences in your data.

The **take-home message** is that if **two samples have similar levels of expression for the genes that contribute significantly to PC1**, they will have similar PC1 scores and will be plotted close to each other on the PC1 axis. Therefore, we would expect **biological replicates** (samples from the same condition) to have similar PC1 scores and cluster together. On the other hand, samples from different treatment groups should have different PC1 scores, reflecting their distinct gene expression patterns. This is more easily understood through visualizing example PCA plots.

#### 3.1.0.2 Interpreting PCA plots

In real datasets with many samples and genes, PCA reduces the complex, high-dimensional data into a 2-dimensional space, where each sample is represented by its scores for the principal components. Typically, we plot the first two principal components (PC1 and PC2), as they explain the most variation in the data.

When interpreting PCA plots, **biological replicates** (samples from the same condition) should cluster together because they have similar expression patterns for the genes driving variation in PC1. **Samples from different treatment groups** will often separate along these axes, reflecting differences in their gene expression profiles.

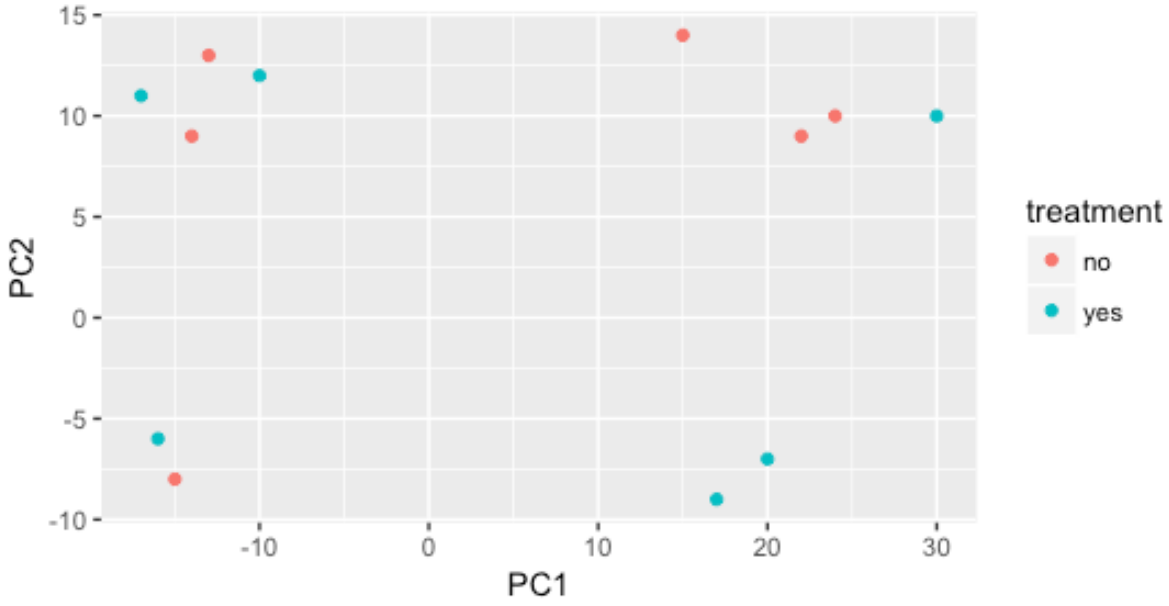
This is best understood by looking at example PCA plots, where you can visualize how biological replicates cluster and treatment groups separate based on their gene expression patterns.

#### 3.1.0.3 Interpreting PCA plots example

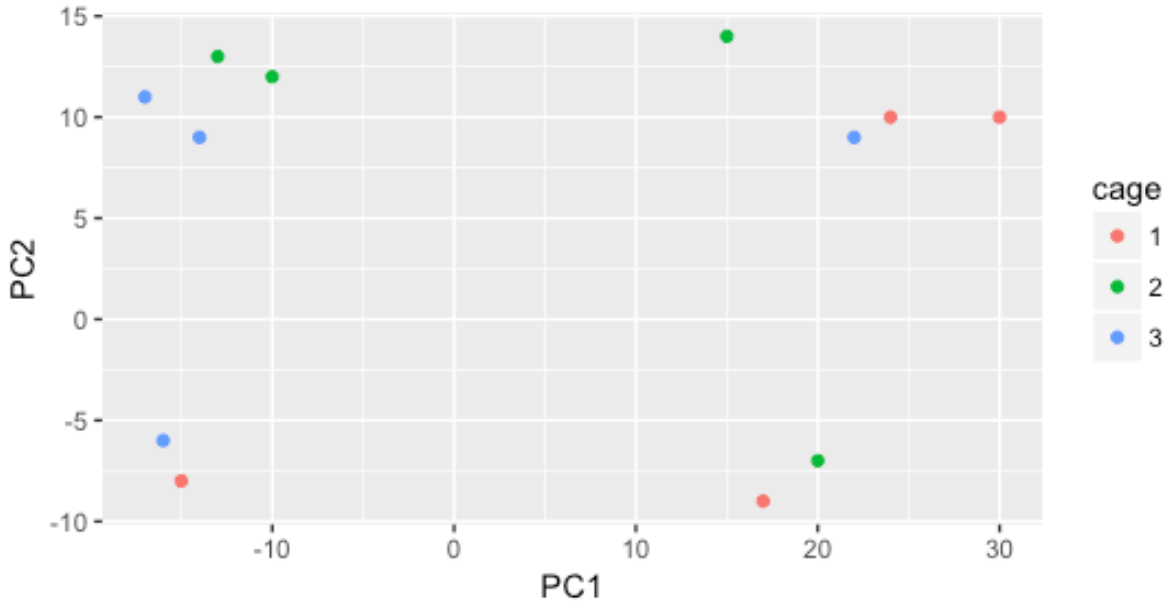
We have an example dataset and a few associated PCA plots below to get a feel for how to interpret them. The metadata for the experiment is displayed below. The main condition of interest is `treatment`.

| sample | strain | date | cage | treatment | replicate | sex |
|---|---|---|---|---|---|---|
| B1 | BALB/cJ | 20180515 | 1 | yes | 1 | M |
| B2 | C57BL/6J | 20180515 | 2 | yes | 1 | M |
| B3 | BALB/cJ | 20180515 | 3 | no | 1 | M |
| B4 | C57BL/6J | 20180515 | 1 | no | 1 | F |
| B5 | BALB/cJ | 20180515 | 2 | yes | 2 | F |
| B6 | C57BL/6J | 20180515 | 3 | yes | 2 | M |
| B7 | BALB/cJ | 20180515 | 1 | no | 2 | M |
| B8 | C57BL/6J | 20180515 | 2 | no | 2 | M |
| B9 | BALB/cJ | 20180515 | 3 | yes | 3 | F |
| B10 | C57BL/6J | 20180307 | 1 | yes | 3 | F |
| B11 | BALB/cJ | 20180307 | 2 | no | 3 | M |
| B12 | C57BL/6J | 20180307 | 3 | no | 3 | M |

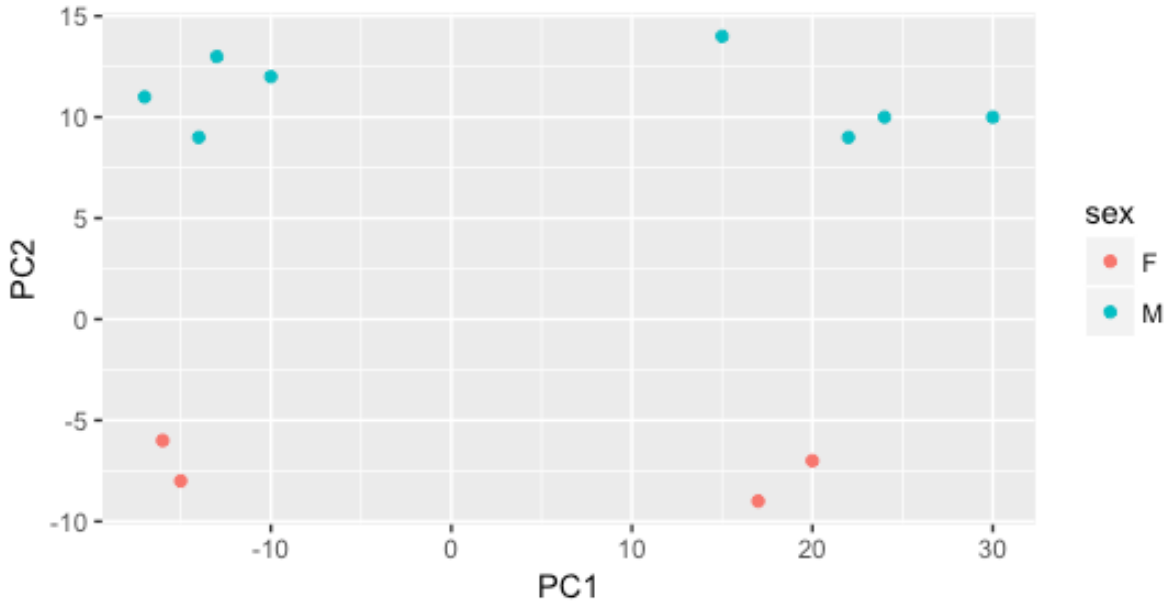
When visualizing the samples on PC1 and PC2, we can color the points based on different metadata columns to identify potential sources of variation. In the first plot, if we don't observe a clear separation of samples by `treatment`, we can explore other factors that may explain the variation. Ideally, our metadata table includes all known sources of variation (such as batch effects, age, or gender), and we can use these factors to color the PCA plot to see if they explain any underlying patterns.



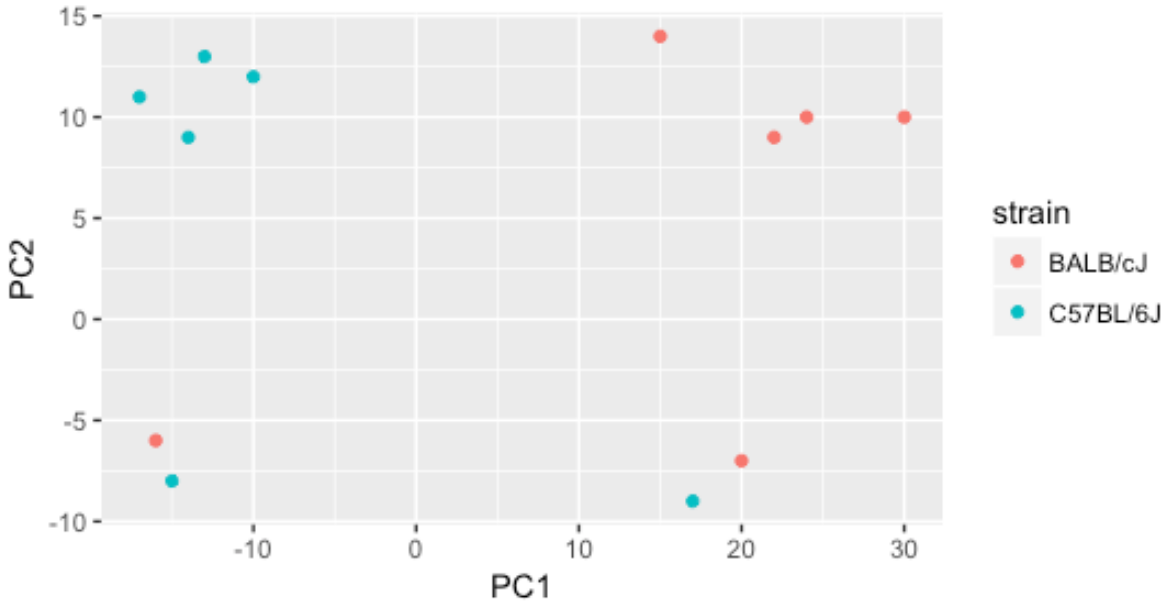

We begin by coloring the points in the PCA plot based on the `cage` factor, but this factor does not seem to explain the variation observed on PC1 or PC2.




Next, we color the points by the `sex` factor. Here, we observe that `sex` separates the samples along PC2, which is valuable information. We can use this in our model to account for variation due to `sex` and potentially regress it out to focus on other sources of variation.



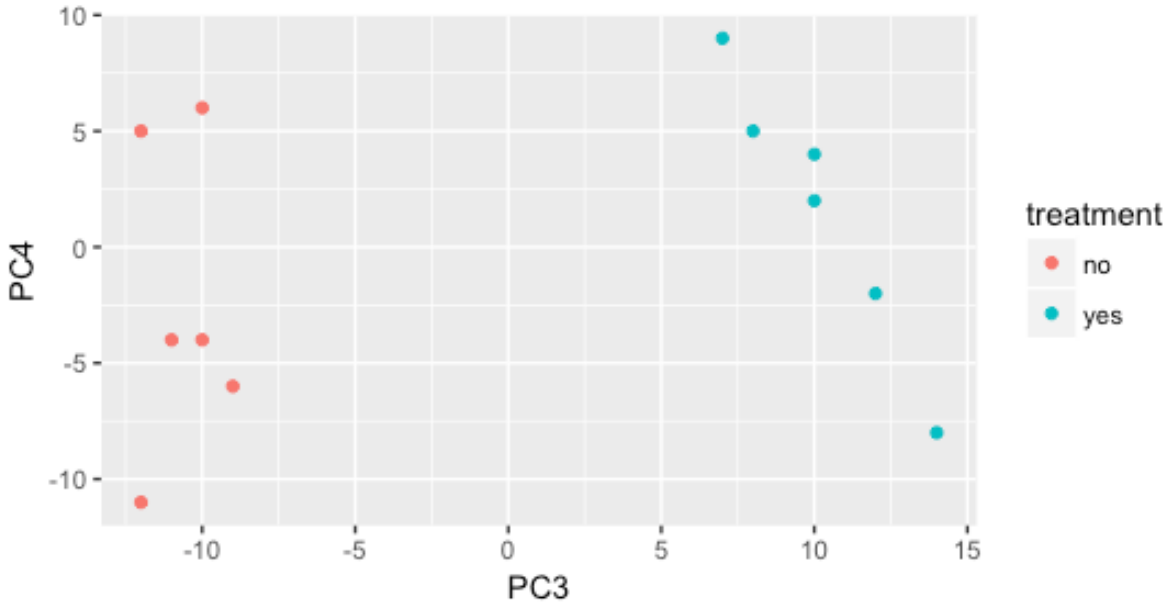

Next we explore the `strain` factor and find that it explains the variation on PC1.




It's great that we have been able to identify the sources of variation for both PC1 and PC2. By accounting for it in our model, we should be able to detect more genes differentially expressed due to `treatment`.

Worrisome about this plot is that we see two samples that do not cluster with the correct strain. This would indicate a likely **sample swap** and should be investigated to determine whether these samples are indeed the labeled strains. If we found there was a switch, we could swap the samples in the metadata. However, if we think they are labeled correctly or are unsure, we could just remove the samples from the dataset.

Still we haven't found if `treatment` is a major source of variation after `strain` and `sex`. So, we explore PC3 and PC4 to see if `treatment` is driving the variation represented by either of these PCs.

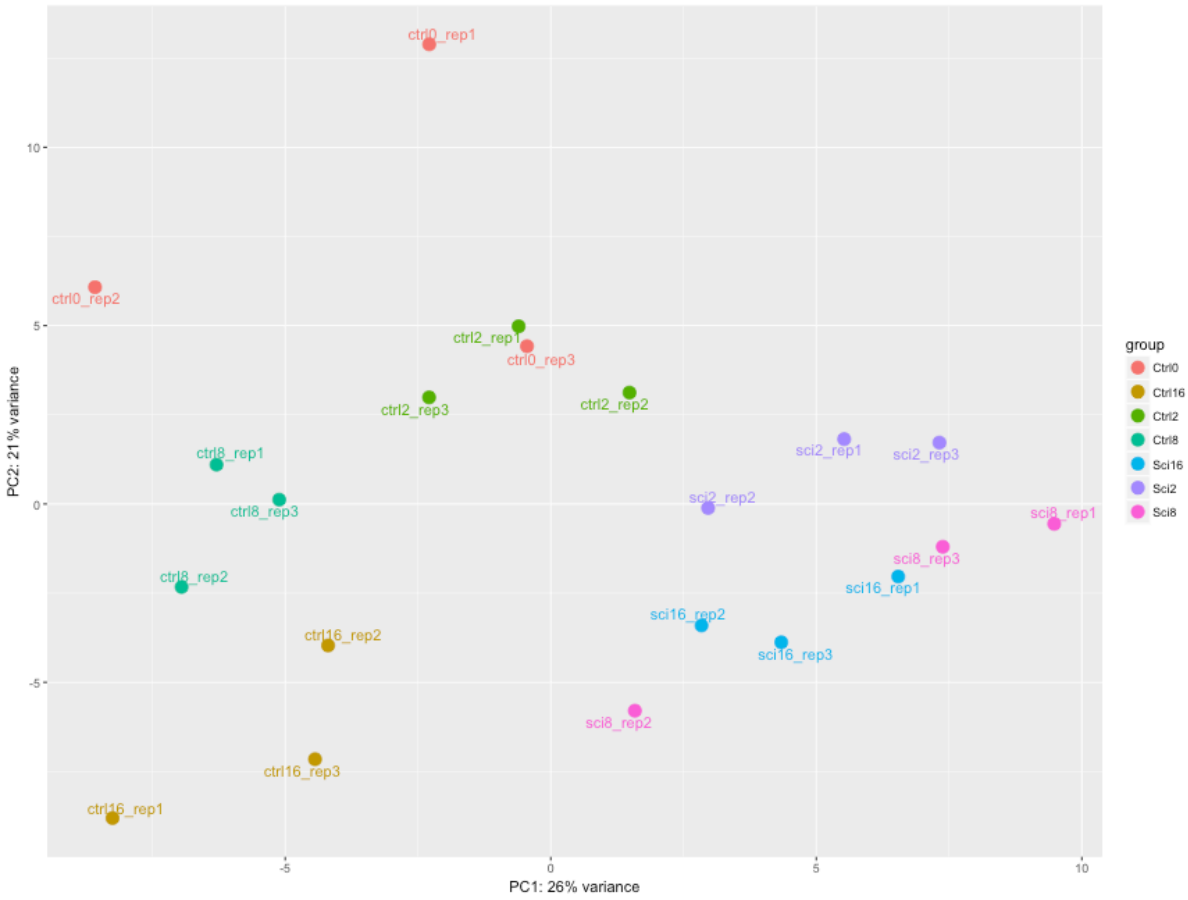



We find that the samples separate by `treatment` on PC3, and are optimistic about our DE analysis since our condition of interest, `treatment`, is separating on PC3 and we can regress out the variation driving PC1 and PC2. Treatment separating on PC3 indicates that it still contributes to variation, but its effect is smaller and is only captured after accounting for the

larger effects (`strain`, `sex`).. me stop

**Exercise** points = +5

The figure below was generated from a time course experiment with sample groups 'Ctrl' and 'Sci' and the following timepoints: 0h, 2h, 8h, and 16h.




**Determine the sources explaining the variation represented by PC1 and PC2.**

- Ans:

**Do the sample groups separate well?**

- Ans:

**Do the replicates cluster together for each sample group?**

- Ans:

**Are there any outliers in the data?**

- Ans:

**Should we have any other concerns regarding the samples in the dataset?**

- Ans:

### 3.1.1 Recommended Resources

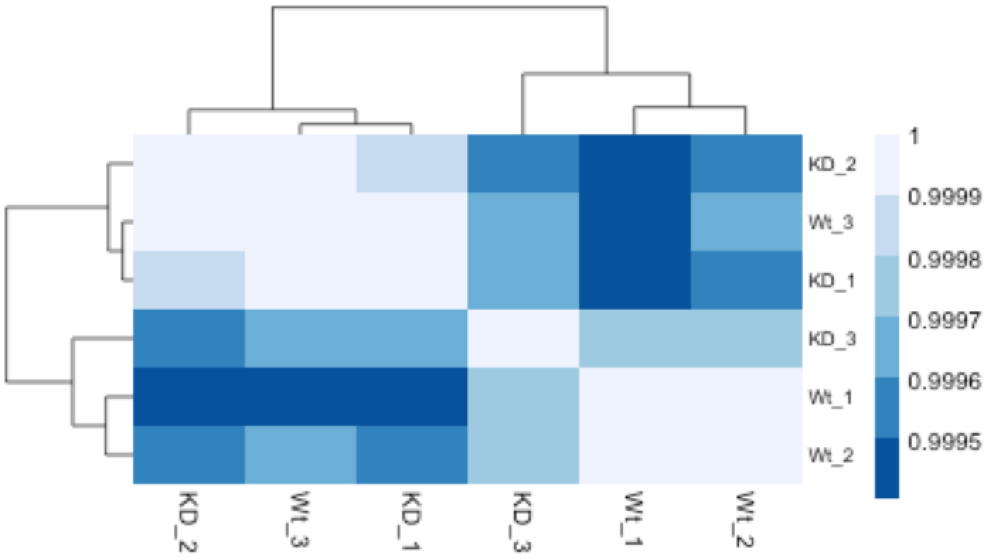
For further reading and visual examples of PCA, consider resources such as: - StatQuest's PCA video (Link) - Online tutorials on PCA in bioinformatics.

### 3.1.2 Hierarchical Clustering Heatmap

Similar to PCA, hierarchical clustering is another, complementary method for identifying strong patterns in a dataset and potential outliers. The heatmap displays **the correlation of gene expression for all pairwise combinations of samples** in the dataset. Since the majority of genes are not differentially expressed, samples generally have high correlations with each other (values higher than 0.80). Samples below 0.80 may indicate an outlier in your data and/or sample contamination.

The hierarchical tree can indicate which samples are more similar to each other based on the normalized gene expression values. The color blocks indicate substructure in the data, and you would expect to see your replicates cluster together as a block for each sample group. Additionally, we expect to see samples clustered similar to the groupings observed in a PCA plot.

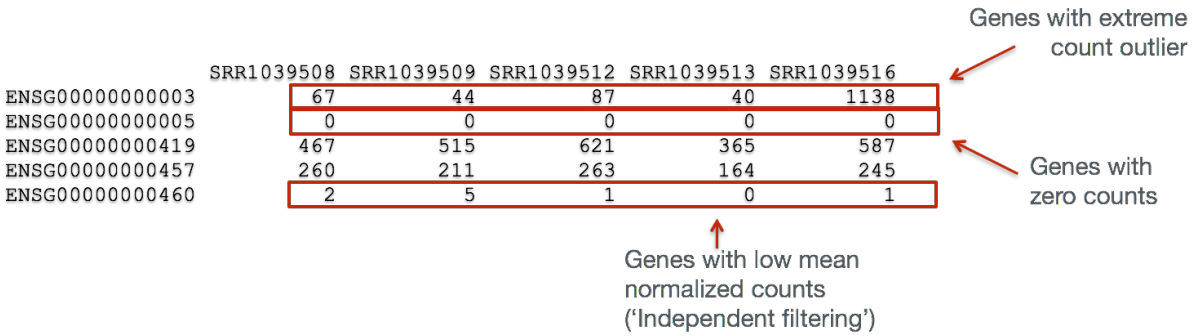
**In the plot below, we would be quite concerned about 'Wt_3' and 'KD_3' samples not clustering with the other replicates. We would want to explore the PCA to see if we see the same clustering of samples.**




## 3.2 Gene-level QC

In addition to examining how well the samples/replicates cluster together, there are a few more QC steps. Prior to differential expression analysis it is beneficial to omit genes that have little or no chance of being detected as differentially expressed. This will increase the power to detect differentially expressed genes. The genes omitted fall into three categories:

- Genes with zero counts in all samples
- Genes with an extreme count outlier
- Genes with a low mean normalized counts




**DESeq2 will perform this filtering by default; however other DE tools, such as EdgeR will not.** Filtering is a necessary step, especially when you are using methods other than DESeq2.

## 3.3 Mov10 quality assessment and exploratory analysis using DESeq2

Now that we have a good understanding of the QC steps normally employed for RNA-seq, let's implement them for the Mov10 dataset we are going to be working with.

### 3.3.1 Transform normalized counts using the rlog transformation

**To improve the distances/clustering for the PCA and heirarchical clustering visualization methods**, we need to apply the **regularized log transformation (rlog)** transformation to the normalized counts. **rlog** is used for visualizations in RNA-Seq data analysis, particularly with tools like DESeq2, because it stabilizes the variance across different levels of gene expression, making it easier to interpret patterns in the data.

**Example of Variance Stabilization** Raw RNA-Seq count data has different levels of variability:

Without stabilization: Lowly expressed genes show high variability between samples (often due to noise), and highly expressed genes show less variability. With stabilization (e.g., rlog or vst): Variance across genes is more evenly distributed, which makes the data easier to work with in analyses like clustering or PCA.

> The rlog transformation of the normalized counts is only necessary for these visualization methods during this quality assessment.

```
### Transform counts for data visualization
rld <- rlog(dds, blind=TRUE)
```

The `blind=TRUE` argument results in a transformation unbiased to sample condition information. When performing quality assessment, it is important to include this option. The DESeq2 vignette has more details.

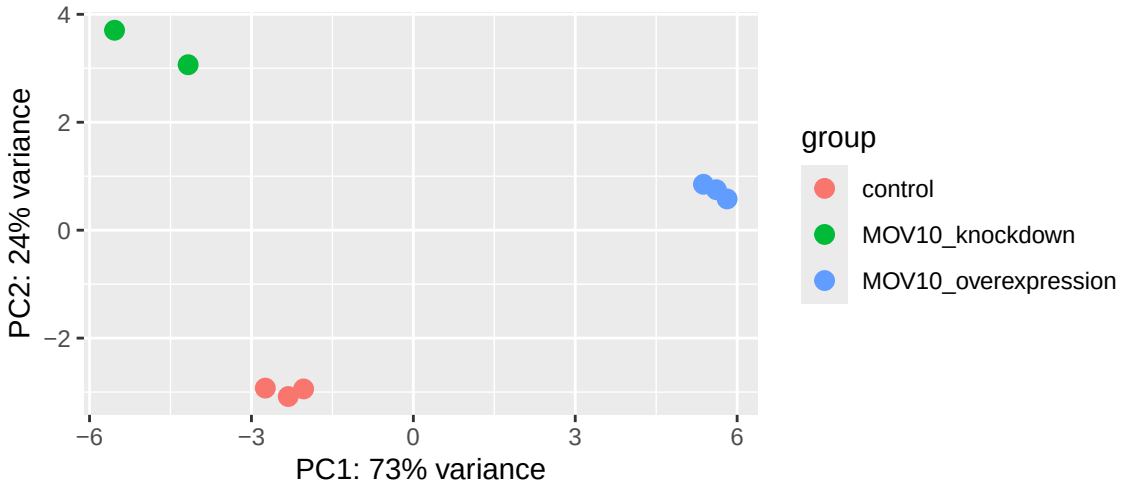
The `rlog` function returns a `DESeqTransform` object, another type of DESeq-specific object. The reason you don't just get a matrix of transformed values is because all of the parameters (i.e. size factors) that went into computing the rlog transform are stored in that object. We use this object to plot the PCA and heirarchical clustering figures for quality assessment.

### 3.3.2 Principal components analysis (PCA)

DESeq2 has a built-in function for plotting PCA plots, that uses `ggplot2` under the hood. This is great because it saves us having to type out lines of code and having to fiddle with the different ggplot2 layers. In addition, it takes the `rlog` object as an input directly, hence saving us the trouble of extracting the relevant information from it.

The function `plotPCA()` requires two arguments as input: an `rlog` object and the `intgroup` (the column in our metadata that we are interested in).

```
### Plot PCA
plotPCA(rld, intgroup="sampletype")
```




**What does this plot tell you about the similarity of samples? Does it fit the expectation from the experimental design?** By default the function uses the *top 500 most variable genes*. You can change this by adding the `ntop` argument and specifying how many genes you want to use to draw the plot.

Resources are available to learn how to do more complex inquiries using the PCs.

### 3.3.3 Hierarchical Clustering

Since there is no built-in function for heatmaps in DESeq2 we will be using the `pheatmap()` function from the `pheatmap` package. This function requires a matrix/dataframe of numeric values as input, and so the first thing we need to is retrieve that information from the `rld` object:

```
### Extract the rlog matrix from the object
rld_mat <- assay(rld) ## assay() is function from the "SummarizedExperiment" package that was loaded when you
```

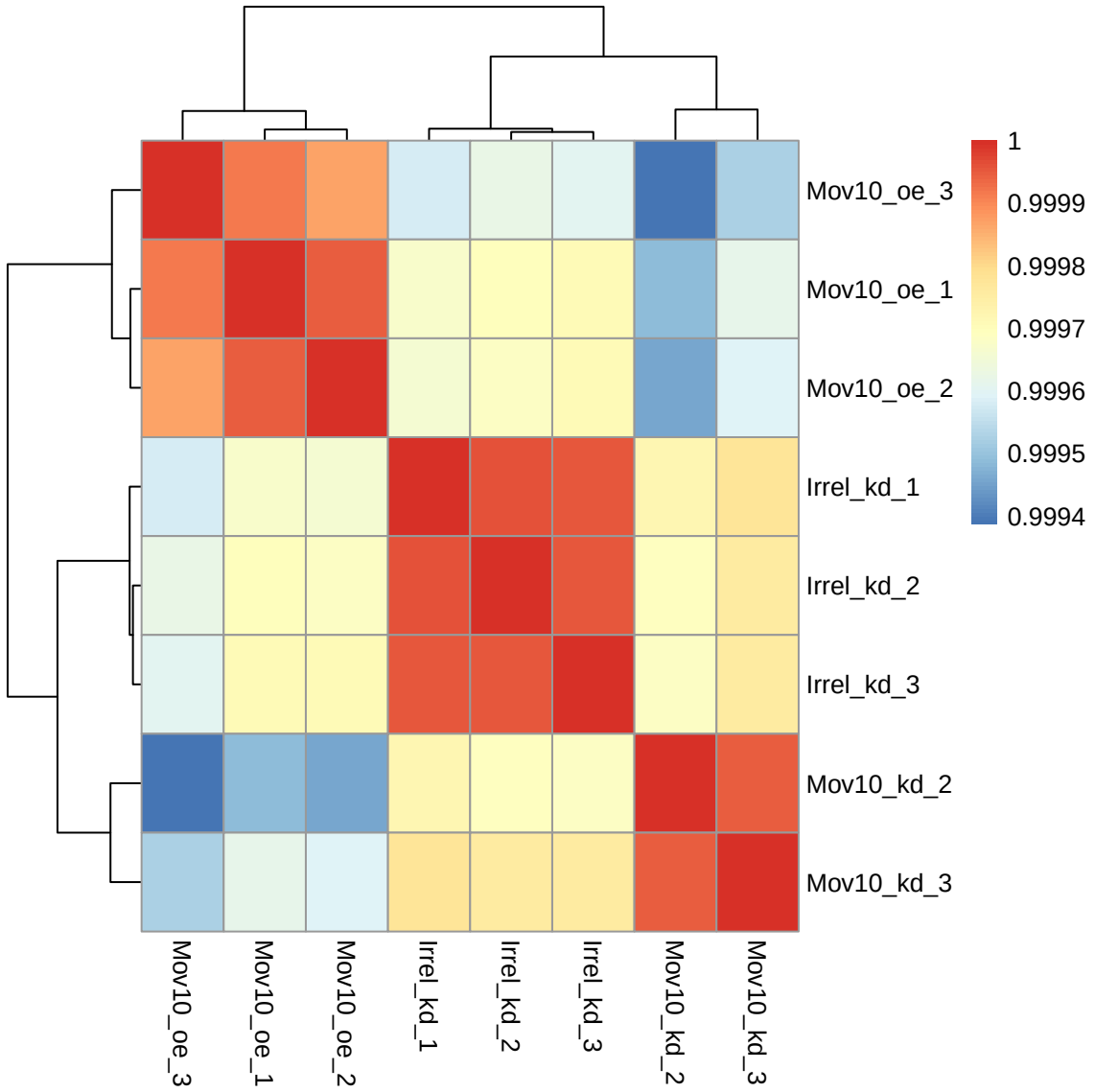
Then we need to compute the pairwise correlation values for samples. We can do this using the `cor()` function:

```
### Compute pairwise correlation values
rld_cor <- cor(rld_mat) ## cor() is a base R function

head(rld_cor)   ## check the output of cor(), make note of the rownames and colnames
```

```
##            Irrel_kd_1 Irrel_kd_2 Irrel_kd_3 Mov10_kd_2 Mov10_kd_3 Mov10_oe_1
## Irrel_kd_1  1.0000000  0.9999614  0.9999532  0.9997202  0.9997748  0.9996700
## Irrel_kd_2  0.9999614  1.0000000  0.9999544  0.9996918  0.9997568  0.9996984
## Irrel_kd_3  0.9999532  0.9999544  1.0000000  0.9996816  0.9997574  0.9997067
## Mov10_kd_2  0.9997202  0.9996918  0.9996816  1.0000000  0.9999492  0.9994868
## Mov10_kd_3  0.9997748  0.9997568  0.9997574  0.9999492  1.0000000  0.9996154
## Mov10_oe_1  0.9996700  0.9996984  0.9997067  0.9994868  0.9996154  1.0000000
##            Mov10_oe_2 Mov10_oe_3
## Irrel_kd_1  0.9996599  0.9995804
## Irrel_kd_2  0.9996825  0.9996227
## Irrel_kd_3  0.9997090  0.9996026
## Mov10_kd_2  0.9994565  0.9993869
## Mov10_kd_3  0.9995905  0.9995235
## Mov10_oe_1  0.9999505  0.9999196
```

And now to plot the correlation values as a heatmap:

```
### Plot heatmap
pheatmap(rld_cor)
```




Overall, we observe pretty high correlations across the board ( $> 0.999$) suggesting no outlying sample(s). Also, similar to the PCA plot you see the samples clustering together by sample group. Together, these plots suggest to us that the data are of good quality and we have the green light to proceed to differential expression analysis.

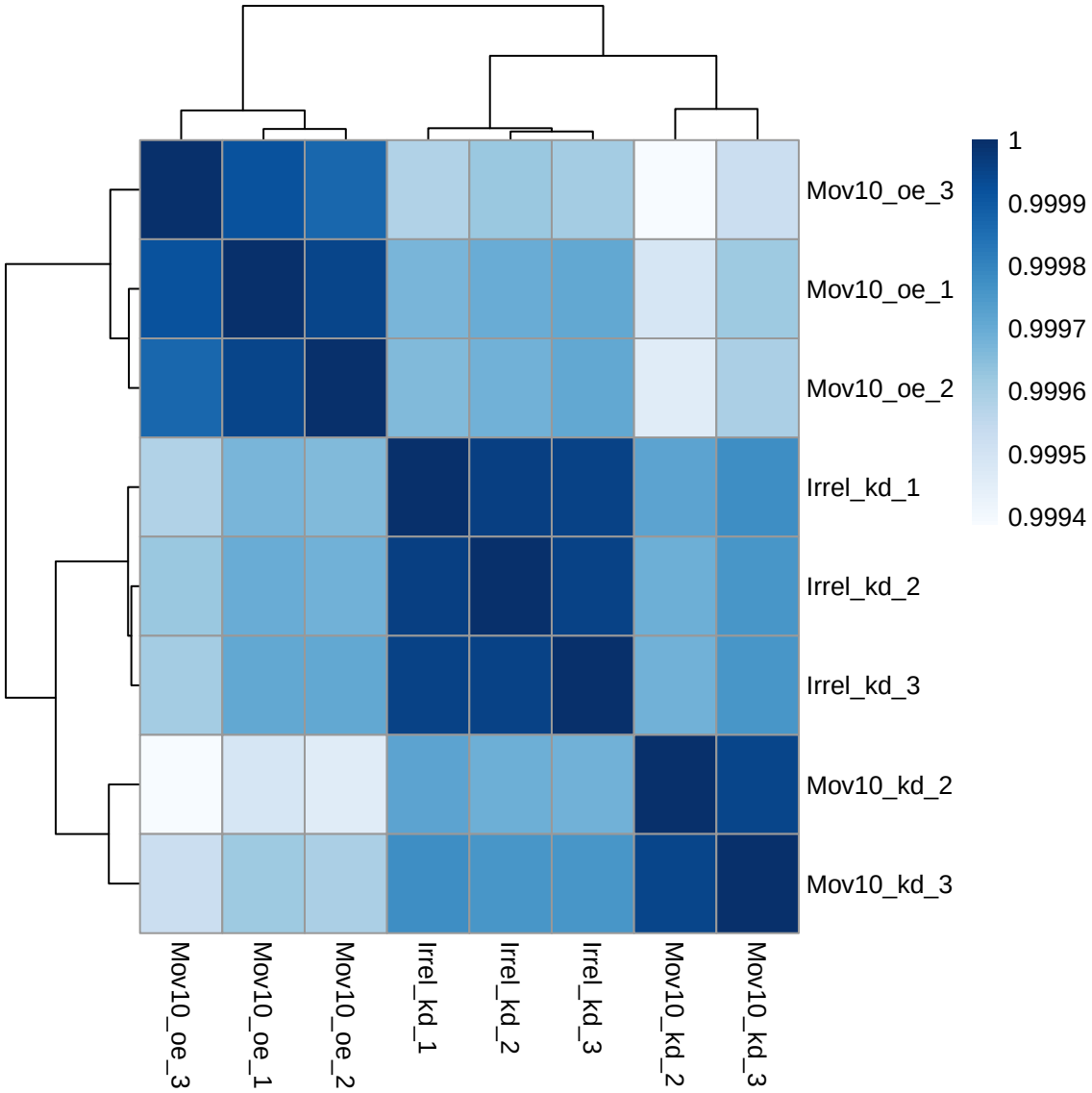
**Exercise** points = +3

The pheatmap function has a number of different arguments that we can alter from default values to enhance the aesthetics of the plot. Try adding the arguments `color`, `border_color`, `fontsize_row`, `fontsize_col`, `show_rownames` and `show_colnames` to your pheatmap. How does your plot change (plot shown below)? Take a look through the help pages (?pheatmap) and identify what each of the added arguments is contributing to the plot.

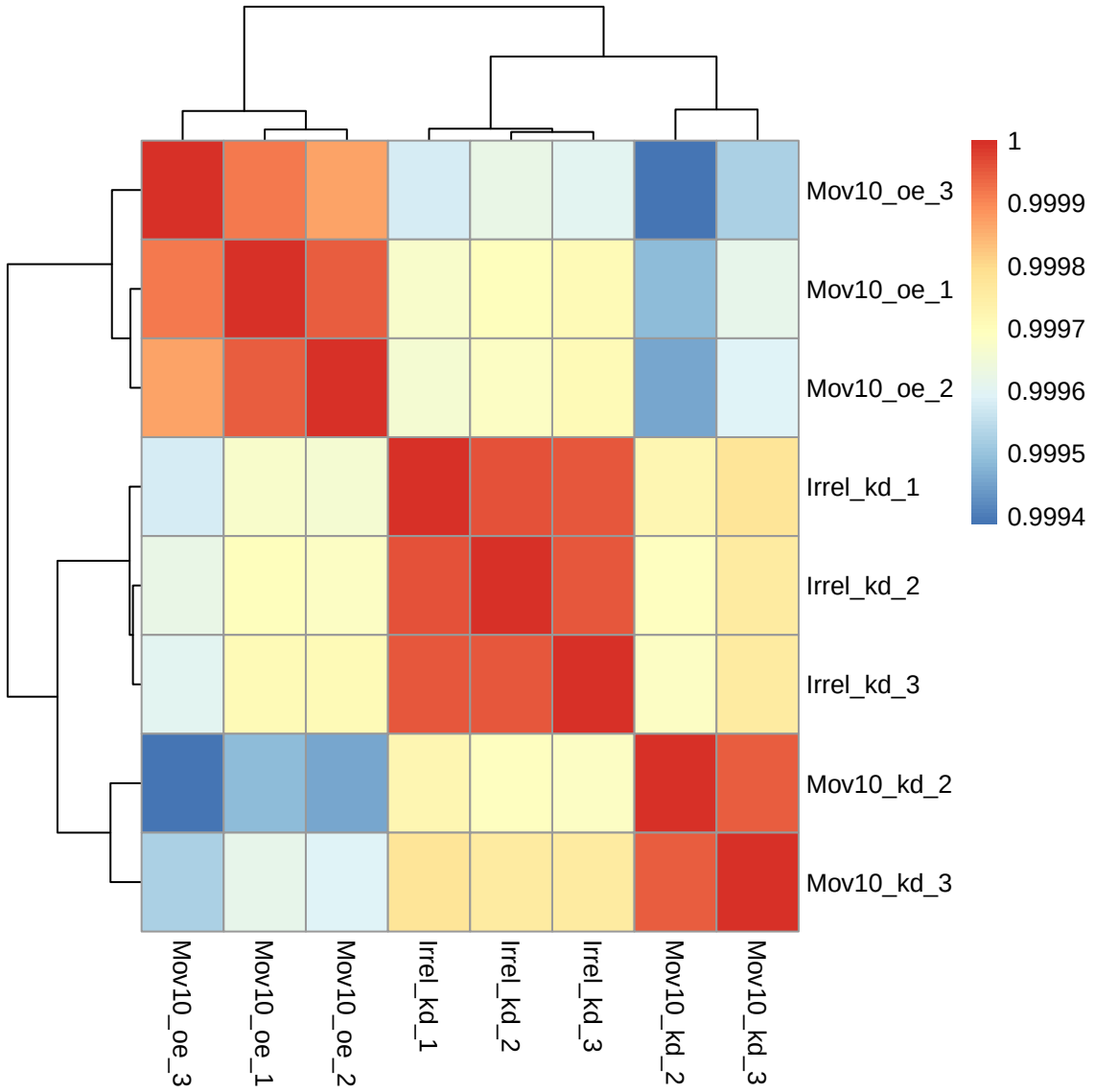
Pheatmap for exercise:

```
library(pheatmap)
library(RColorBrewer)
heat.colors <- colorRampPalette(brewer.pal(9, "Blues"))(100)
```

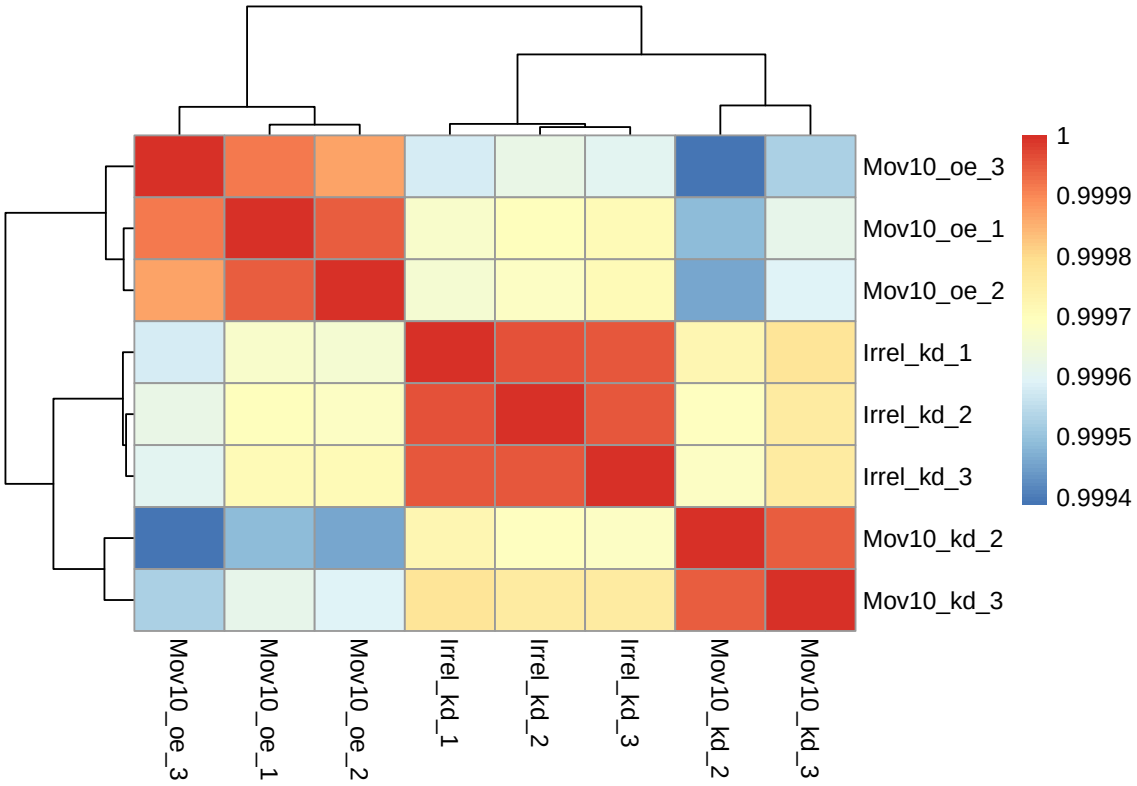
```
pheatmap(rld_cor, color = colorRampPalette(brewer.pal(9, "Blues"))(100))
```

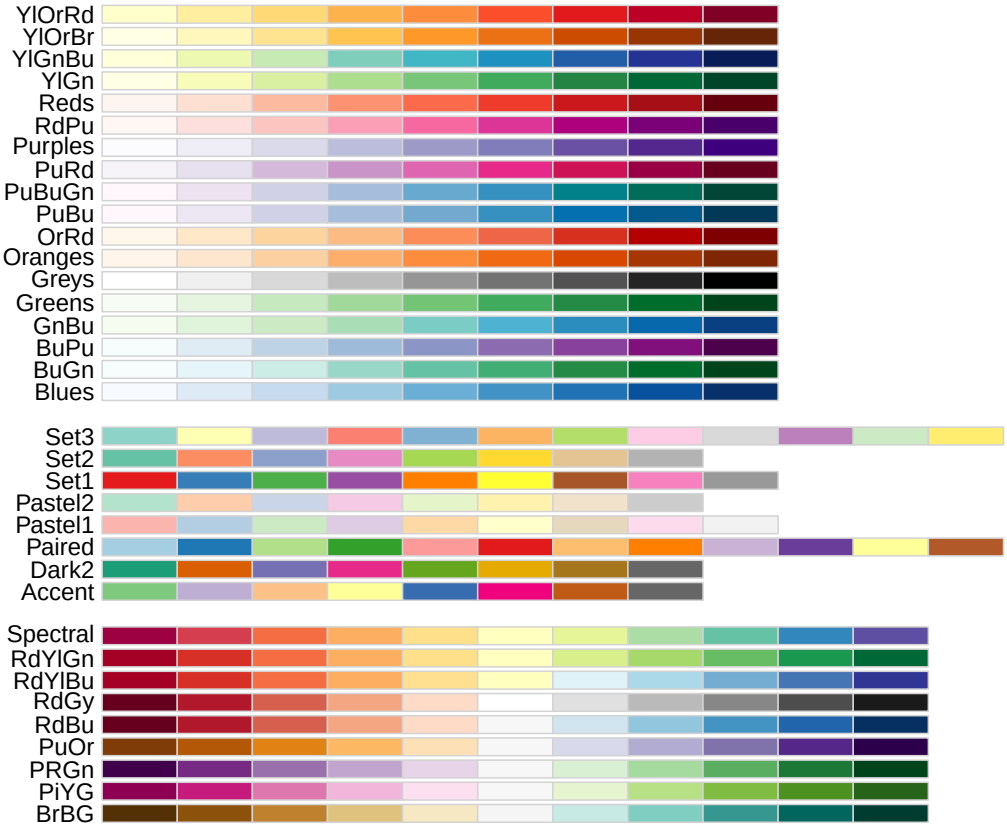



```
library(pheatmap)
library(RColorBrewer)

pheatmap(rld_cor)
```




```
library(pheatmap)
pheatmap(rld_cor)
```


Mov10_oe_3
Mov10_oe_1
Mov10_oe_2
Irrel_kd_1
Irrel_kd_2
Irrel_kd_3
Mov10_kd_2
Mov10_kd_3
1
0.9999
0.9998
0.9997
0.9996
0.9995
0.9994

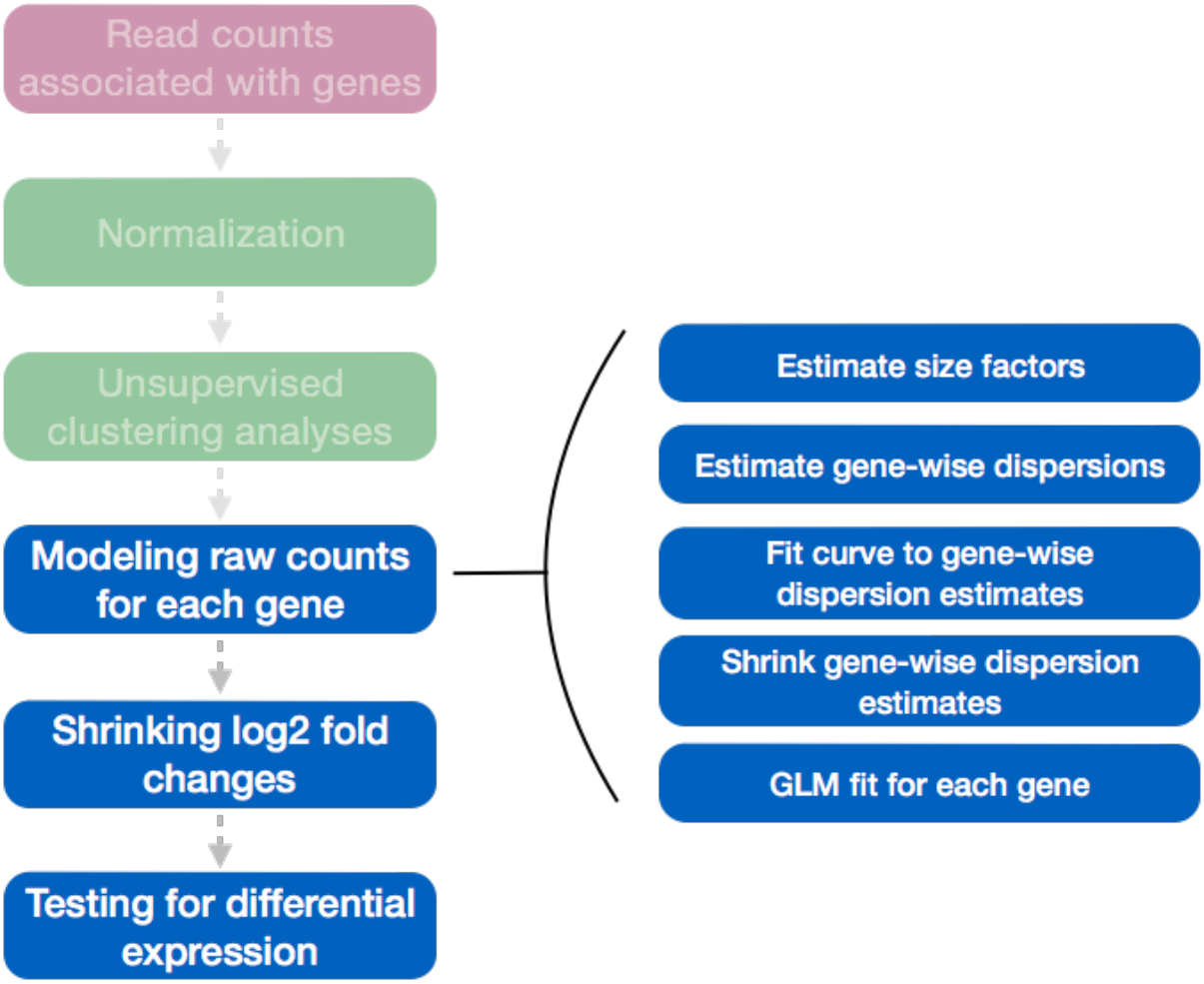



```
# The color palette "Blues" is a good choice for this heatmap
heat.colors <- brewer.pal(9, "Blues")

# Your code here
```

# Chapter 4

# DGE analysis workflow

Differential expression analysis with DESeq2 involves multiple steps as displayed in the flowchart below in blue.




Here's a more succinct version of your workflow:

**DESeq2 Workflow Overview:**

1. **Load Data:** Start with raw count data and sample metadata.

2. **Normalize Counts:** Estimate size factors to adjust for sequencing depth or library size differences.

3. **Estimate Dispersion:** Calculate gene-wise dispersion, measuring variability in counts not explained by mean expression.

4. **Fit Dispersion Trend:** Model the relationship between mean counts and dispersion, either parametrically (default) or non-parametrically.

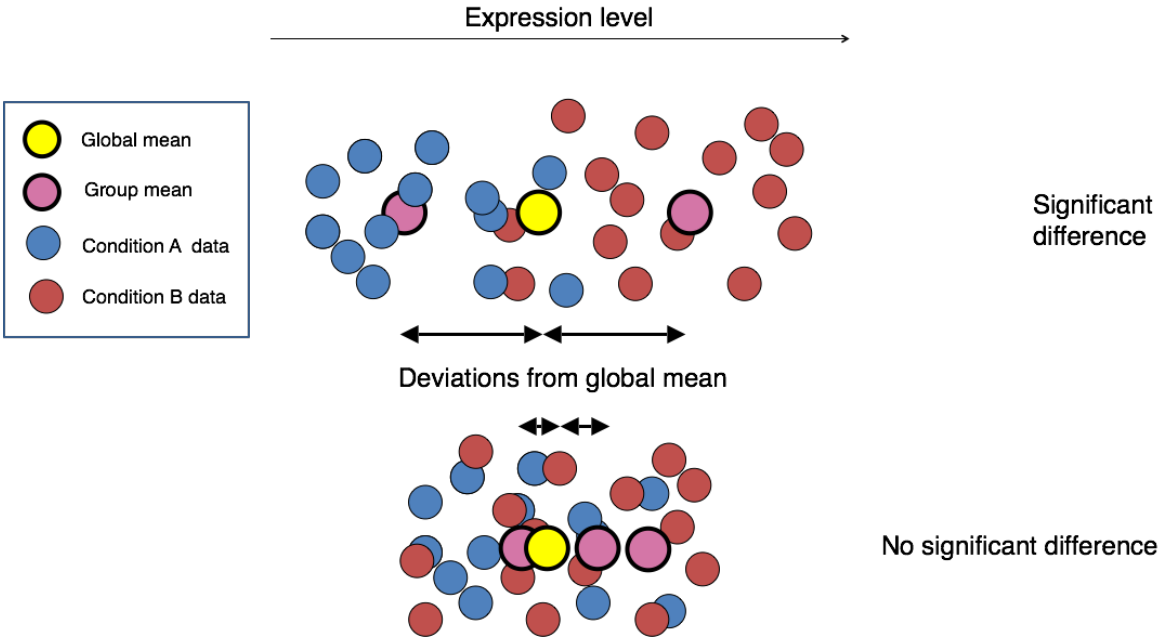
5. **Shrink Dispersions:** Use empirical Bayes shrinkage to stabilize dispersion estimates, particularly for low-count genes. Empirical Bayes is a version of Bayesian statistics where the prior distribution is estimated from the data itself, rather than being set externally. In empirical Bayes shrinkage, the idea is to estimate a common pattern or distribution (the prior) across all observations (e.g., all genes) and then use this estimated distribution to influence or "shrink" each individual observation.

6. **Fit NB Model:** Fit a negative binomial model using the shrunken dispersion estimates, capturing variability and estimating log-fold changes.

7. **Shrink Log-Fold Changes:** Optionally stabilize log-fold changes with regularization, especially for low-expression genes.

8. **Statistical Testing:** Perform Wald tests to identify differentially expressed genes, providing adjusted p-values for multiple testing.

This final step in the differential expression analysis workflow of fitting the raw counts to the **NB model** and performing the statistical test for differentially expressed genes, is the step we care about. This is the step that performs statistical tests to identify significant differences in gene expression between sample groups.




Although the DESeq2 paper was published in 2014, but the package is continually updated and available for use in R through Bioconductor.

## 4.1 Running DESeq2

Prior to performing the differential expression analysis, it is a good idea to know what **sources of variation** are present in your data, either by exploration during the QC and/or prior knowledge. **Sources of variation** refer to anything that causes differences in gene expression across your samples, including **biological factors** (**treatment**, **sex**, **age**) and **technical factors** (**batch effects**, **sequencing depth**). Ideally, these are all factors in your metadata. These factors lead to variability in the data and can affect your ability to detect meaningful biological signals (like differential expression between **treated** and **control** samples).Once you know the major sources of variation, you can remove them prior to analysis or control for them in the statistical model by including them in your **design formula**.

### 4.1.1 Set the Design Formula

The main purpose of the **design formula** in DESeq2 is to specify the factors that are influencing gene expression so that their effects can be **accounted for** or **controlled** during the analysis. This allows DESeq2 to isolate the effect of the variable you're primarily interested in while adjusting for other known sources of variation. The design formula should have all of the factors in your metadata that account for major sources of variation in your data. The last factor entered in the formula should be the condition of interest.

For example:

```
design(dds)
```

```
## ~sampletype
```

This shows that **sampletype** is the condition of interest and as such is the only covariate. DESeq2 will test for differential expression between the two sample types, adjusting for any other variation.

**Example**

If you want to examine the expression differences between `treatments` as shown below, and you know that major sources of variation include `sex` and `age`, then your design formula would be:

```
design <- ~ sex + age + treatment
```

| | sex | age | litter | treatment |
|---|---|---|---|---|
| sample1 | M | 11 | 1 | Ctrl |
| sample2 | M | 13 | 2 | Ctrl |
| sample3 | M | 11 | 1 | Treat |
| sample4 | M | 13 | 1 | Treat |
| sample5 | F | 11 | 1 | Ctrl |
| sample6 | F | 13 | 1 | Ctrl |
| sample7 | F | 11 | 1 | Treat |
| sample8 | F | 13 | 2 | Treat |

This allows DESeq2 to adjust for **sex** and **age** while testing for differences in **treatment**.

**Why is this important?**

Including known sources of variation (like **sex** and **age**) ensures any gene expression differences are due to the condition of interest (e.g., **treatment**), not confounded by other factors.

**Steps to set the design formula:** 1. Identify your factor of interest (e.g., **treatment**). 2. Determine any confounders (e.g., **sex**, **age**). 3. Write the formula, placing the factor of interest last. 4. Use **PCA** to ensure all relevant factors are included.

**Exercise** points = +3

1. Suppose you wanted to study the expression differences between the two age groups in the metadata shown above, and major sources of variation were **sex** and **treatment**, how would the design formula be written?

```
# Your code here
```

2. Based on our Mov10 metadata dataframe, which factors could we include in our design formula?

- Ans:

3. What would you do if you wanted to include a factor in your design formula that is not in your metadata?

- Ans:

### 4.1.2 MOV10 Differential Expression Analysis

Now that we understand how to specify the model in DESeq2, we can proceed with running the differential expression pipeline on the raw count data.

**Running Differential Expression in Two Lines of Code**

To obtain differential expression results from our raw count data, we only need to run two lines of code!

First, we create a DESeqDataSet, as we did in the 'Count normalization' lesson, specifying the location of our raw counts and metadata, and applying our design formula:

```
## Create DESeq object
dds <- DESeqDataSetFromMatrix(countData = data, colData = meta, design = ~ sampletype)
```

Next, we run the actual differential expression analysis with a single call to the `DESeq()` function. This function handles everything—from **normalization** to **linear** modeling—all in one step. During execution, `DESeq2` will print messages detailing the steps being performed: estimating size factors, estimating dispersions, gene-wise dispersion estimates, modeling the mean-dispersion relationship, and statistical testing for differential expression.

```
## Run analysis
dds <- DESeq(dds)
```

```
## estimating size factors
```

```
## estimating dispersions
```

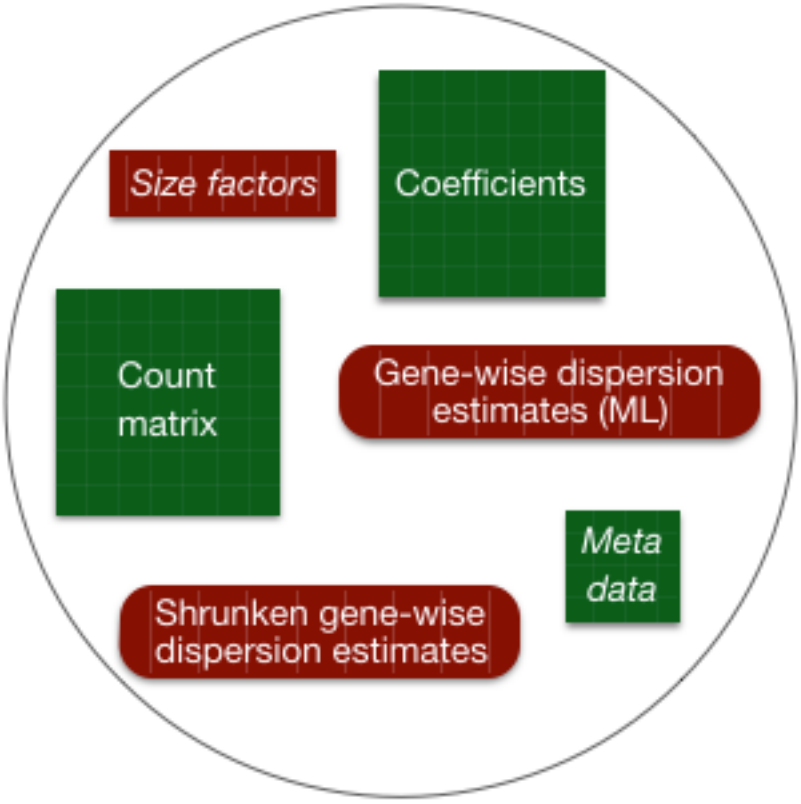
```
## gene-wise dispersion estimates
```

```
## mean-dispersion relationship
```

```
## final dispersion estimates
```

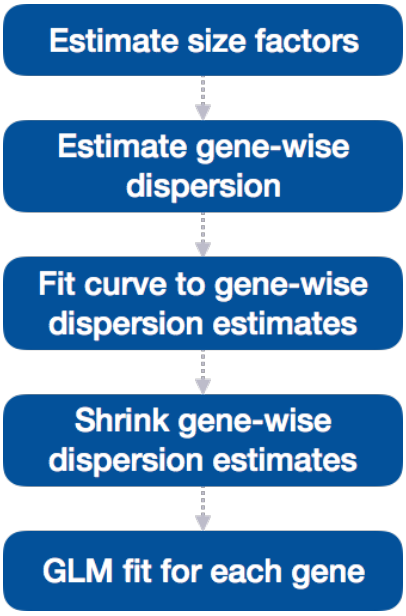
```
## fitting model and testing
```

By re-assigning the result to back to the same variable name (`dds`), we update our `DESeqDataSet` object, which will now contain the results of each step in the analysis, effectively filling in the `slots` of our `DESeqDataSet` object.




## 4.2 DESeq2 differential gene expression analysis workflow

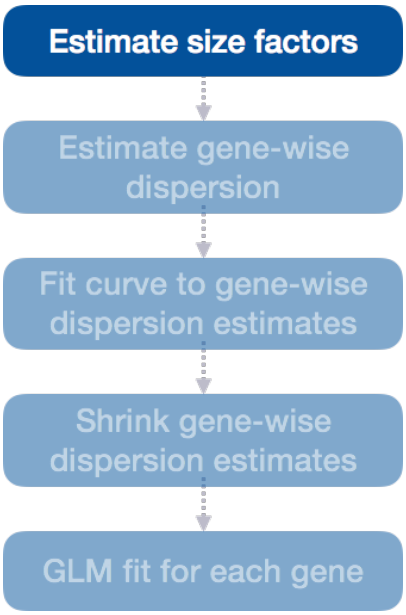
With these two lines of code, we have completed the core steps in the DESeq2 differential gene expression analysis. The key steps in this workflow are summarized below:




In the following sections, we will explore each step in detail to better understand how DESeq2 performs the statistical analysis and what metrics we should focus on to evaluate the quality of the results.

### 4.2.1 Step 1: Estimate size factors

The first step in the differential expression analysis is to estimate the size factors, which is exactly what we already did to normalize the raw counts.




DESeq2 will automatically estimate the size factors when performing the differential expression analysis. However, if you have already generated the size factors using `estimateSizeFactors()`, as we did earlier, then `DESeq2` will use these values.

To normalize the count data, `DESeq2` calculates size factors for each sample using the *median of ratios method* discussed previously in the Count normalization (Chapter 2) lesson.

#### 4.2.1.1 MOV10 DE analysis: examining the size factors

Let's take a quick look at size factor values we have for each sample:

```
## Check the size factors
sizeFactors(dds)
```

```
## Irrel_kd_1 Irrel_kd_2 Irrel_kd_3 Mov10_kd_2 Mov10_kd_3 Mov10_oe_1 Mov10_oe_2
##  1.1224020  0.9625632  0.7477715  1.5646728  0.9351760  1.2016082  1.1205912
## Mov10_oe_3
##  0.6534987
```

Take a look at the total number of reads for each sample:

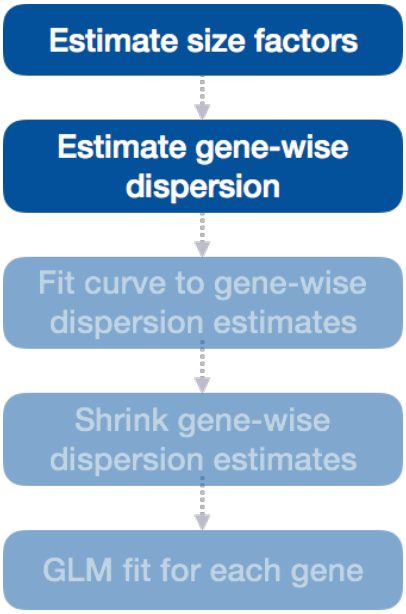
```
## Total number of raw counts per sample
colSums(counts(dds))
```

```
## Irrel_kd_1 Irrel_kd_2 Irrel_kd_3 Mov10_kd_2 Mov10_kd_3 Mov10_oe_1 Mov10_oe_2
##   22687366   19381680   14962754   32826936   19360003   23447317   21713289
## Mov10_oe_3
##   12737889
```

**How do the numbers correlate with the size factor?**

```
# Your code here
```

### 4.2.2 Step 2: Estimate Gene-wise Dispersion




The next step in differential expression analysis is estimating gene-wise dispersions. Understanding dispersion is crucial in RNA-Seq analysis.

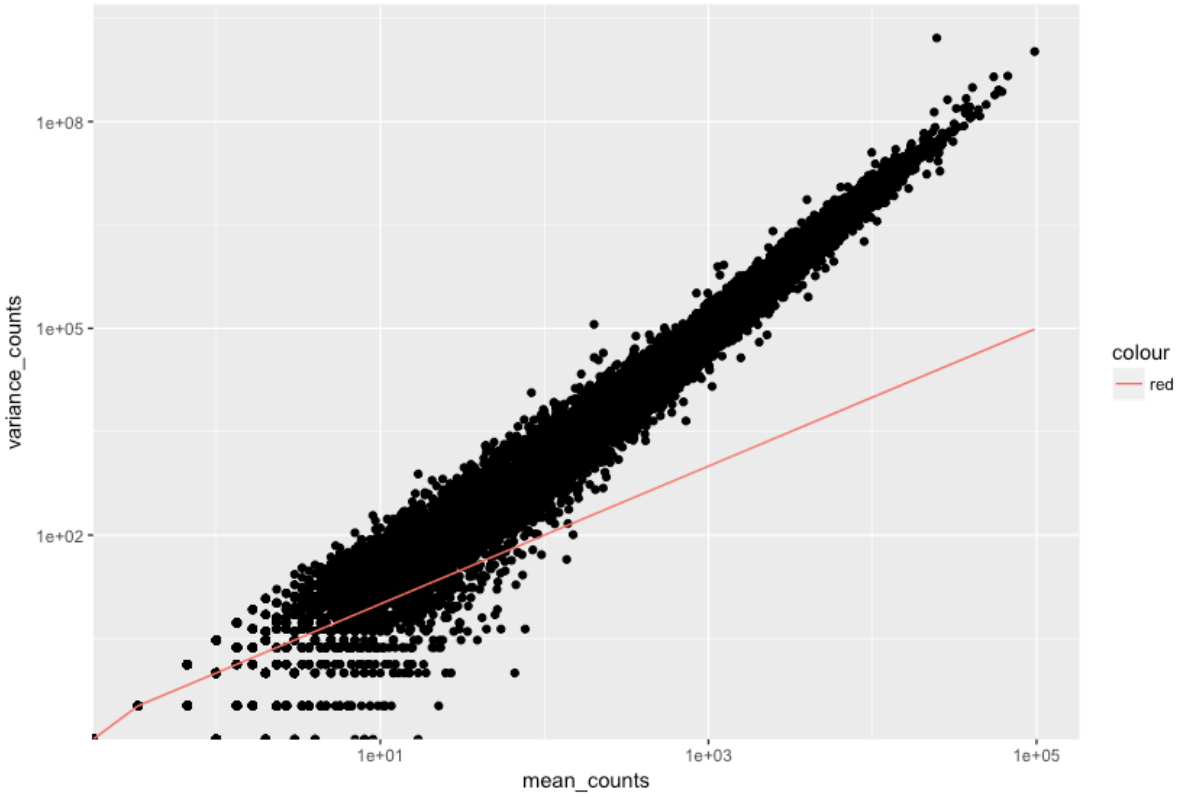
**What is Dispersion?**

Dispersion measures the variability in gene expression that cannot be explained by the mean expression levels alone. It captures the extra-Poisson variability observed in RNA-Seq data, where the variance tends to exceed the mean due to biological and technical factors.

In DESeq2, dispersion reflects the variability in gene expression for a given mean count and is characterized by:

- **Lower mean counts**: Higher dispersion (more variability).
- **Higher mean counts**: Lower dispersion (less variability).

The plot below illustrates the relationship between mean expression and variance: each black dot represents a gene, showing that variance can be predicted more reliably for genes with higher mean counts, while lower mean counts exhibit greater variability.




**Modeling Dispersion Using the Negative Binomial Distribution**

DESeq2 utilizes the Negative Binomial (NB) distribution to model RNA-Seq count data, effectively managing overdispersion, which is common in biological data. The two key parameters of the NB model are the mean expression level and the dispersion.

For each gene $i$, the observed count $Y_{ij}$ in sample $j$ is modeled as:

$$Y_{ij} \sim \text{NB}(\mu_{ij}, \alpha_i)$$

Where: - $Y_{ij}$ is the observed count for gene $i$ in sample $j$. - $\mu_{ij}$ is the expected normalized count for gene $i$. - $\alpha_i$ is the gene-specific dispersion parameter.

### 4.2.3 Step 3: Fit Curve to Gene-wise Dispersion Estimates

After estimating gene-wise dispersions using maximum likelihood, DESeq2 fits a global trend to model how dispersion changes as a function of mean expression across all genes. This process stabilizes the dispersion estimates, especially for low-count genes.

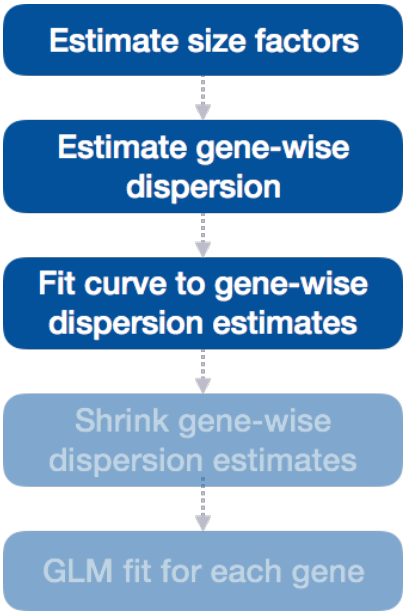
- **Low-count genes** with more uncertain dispersion estimates are pulled closer to the global trend.
- **Genes with higher counts** or more reliable estimates will be shrunk less.

#### Why is Shrinkage Important?

Shrinkage improves the reliability of dispersion estimates, making them more stable for genes with low or moderate counts, which typically have noisier estimates. This reduces the influence of random fluctuations and results in more accurate estimates for downstream differential expression testing.

Additionally, shrinkage helps to reduce false positives in differential expression analysis, ensuring that estimates for these genes are more reliable for subsequent testing.

Genes with extremely high dispersion values (outliers) may demonstrate variability that is unexpected for their mean expression levels and are typically not shrunk. These genes are often biological outliers or affected by technical issues.




### 4.2.4 Step 4: Assessing Model Fit with Dispersion Plots

After shrinkage, it's essential to evaluate how well the model fits the data through dispersion plots. Ideally, most genes should scatter around the fitted curve, with dispersions decreasing as mean expression increases. Unusual patterns, such as points far from the curve, may indicate data quality issues, like outliers or contamination.

Using DESeq2's `plotDispEsts()` function visualizes the fit of the model and the extent of shrinkage across genes.

#### 4.2.4.1 Examples of Problematic Dispersion Plots

Certainly! Problematic dispersion plots in RNA-Seq analysis, particularly when using tools like DESeq2, can arise from various issues. Here are some reasons that could lead to visual discrepancies in dispersion plots:

**Reasons for Problematic Dispersion Plots**

- **Low Count Data**:
- **High Variability**: Genes with low expression levels often exhibit greater variability (higher dispersion) across replicates, leading to noisy estimates.
- **Unstable Estimates**: With few counts, it's difficult to obtain reliable dispersion estimates, resulting in significant scattering in the plot.
- **Outliers in the Data**:
  - **Biological Outliers**: Certain genes may genuinely exhibit high variability due to biological conditions (e.g., activation in response to stimuli), leading to higher dispersion estimates.
  - **Technical Outliers**: Poor quality samples or technical errors in sequencing (e.g., contamination or failed PCR) can result in erroneous high counts for certain genes, affecting overall variability.
- **Batch Effects**:
  - Differences in sequencing runs or sample processing can introduce variability independent of biological differences. If not normalized, batch effects can distort dispersion estimates, making it seem like certain genes are more variable than they truly are.
- **Unequal Group Sizes**:
  - Uneven distribution of samples across experimental conditions can affect dispersion estimates, particularly if one group has fewer replicates or more variability.
- **Unaccounted Biological Variation**:
  - Biological variability among samples or genes that is not captured by the experimental design can affect estimates. For instance, genes involved in complex pathways may show higher dispersion due to interactions.
- **Data Quality Issues**:
  - Problems with sample handling, library construction, or sequencing can introduce discrepancies in data that visually manifest as unexpected dispersion.

### 4.2.5 Assessing and Addressing Issues

To tackle issues identified in problematic dispersion plots:

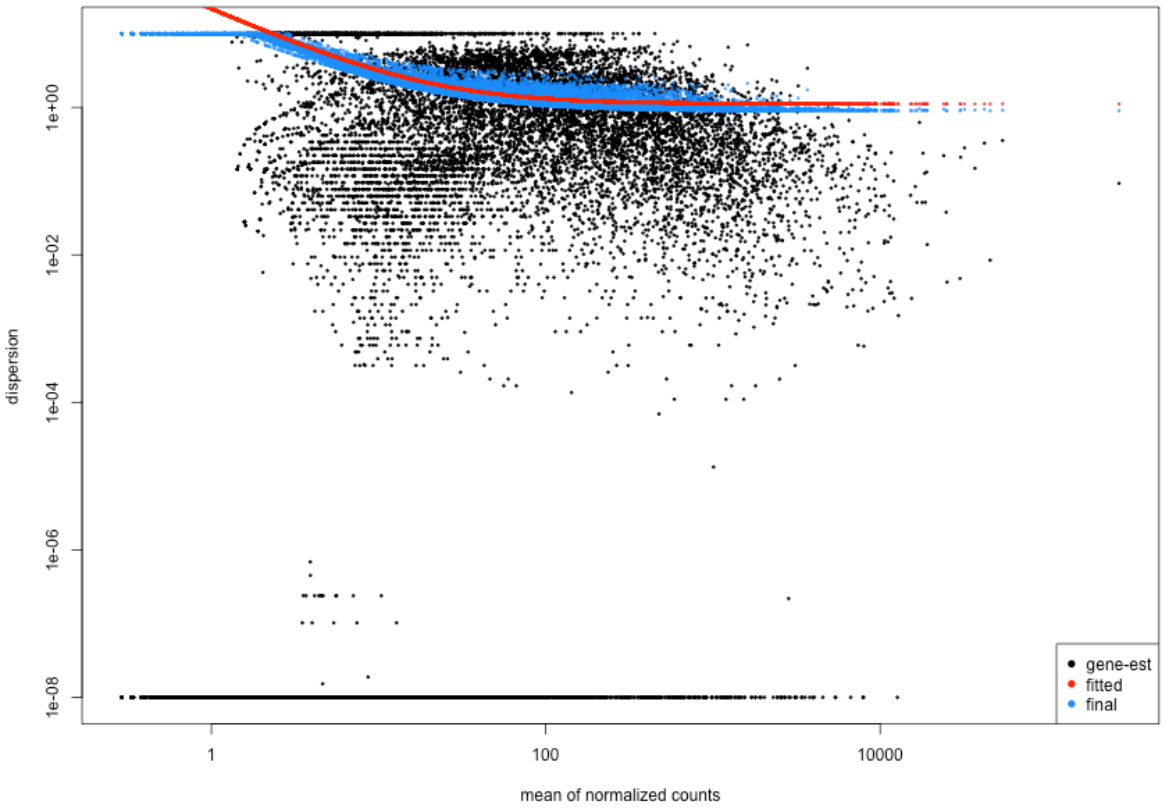
- **Quality Control**: Always perform thorough quality control checks on your raw sequencing data to identify outliers and poorly performing samples.
- **Filtering**: Remove low-count genes or outliers that might skew your data before running the analysis.
- **Increasing Replicates**: Where possible, aim for more replicates to stabilize dispersion estimates, particularly for low-expressed genes.

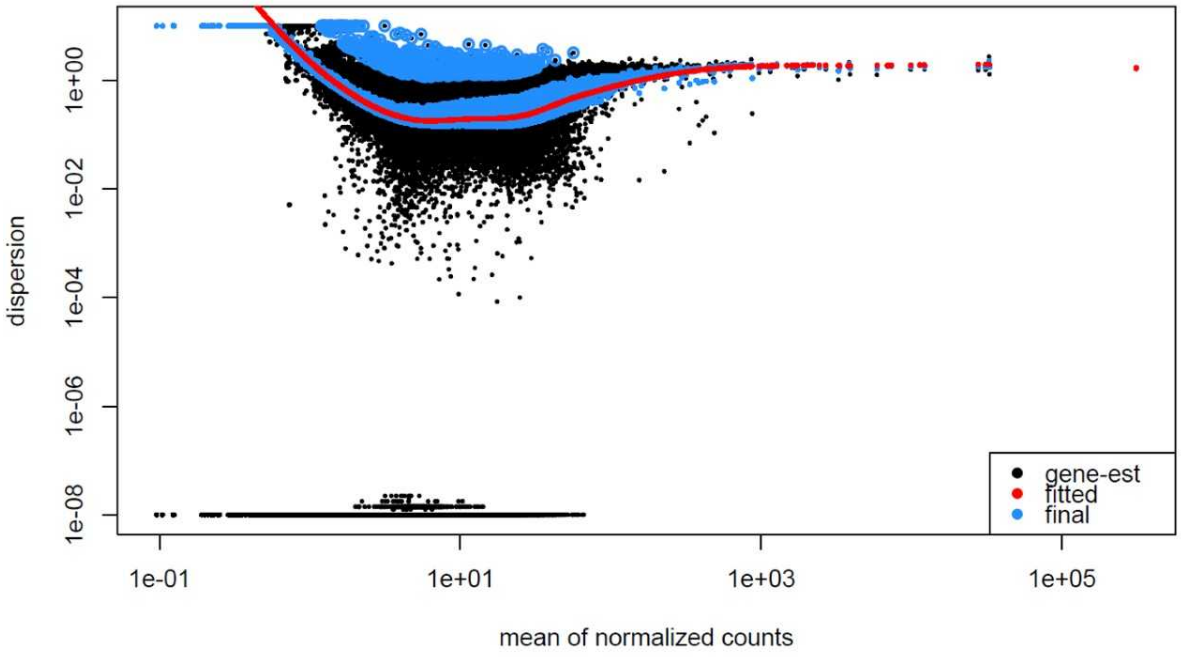
- **Model Evaluation**: Consider exploring different statistical models if patterns suggest that the negative binomial model is not a good fit.

- **Visual Inspection**: Regularly visualize data through various methods (e.g., PCA, heatmaps, dispersion plots) to check for quality and consistency.

By understanding the potential causes for problematic dispersion plots, you can troubleshoot and improve your RNA-Seq analysis to yield more reliable results. If you have specific questions or scenarios you want to discuss, feel free to ask!

Here are examples of dispersion plots suggesting potential data quality or fitting issues:

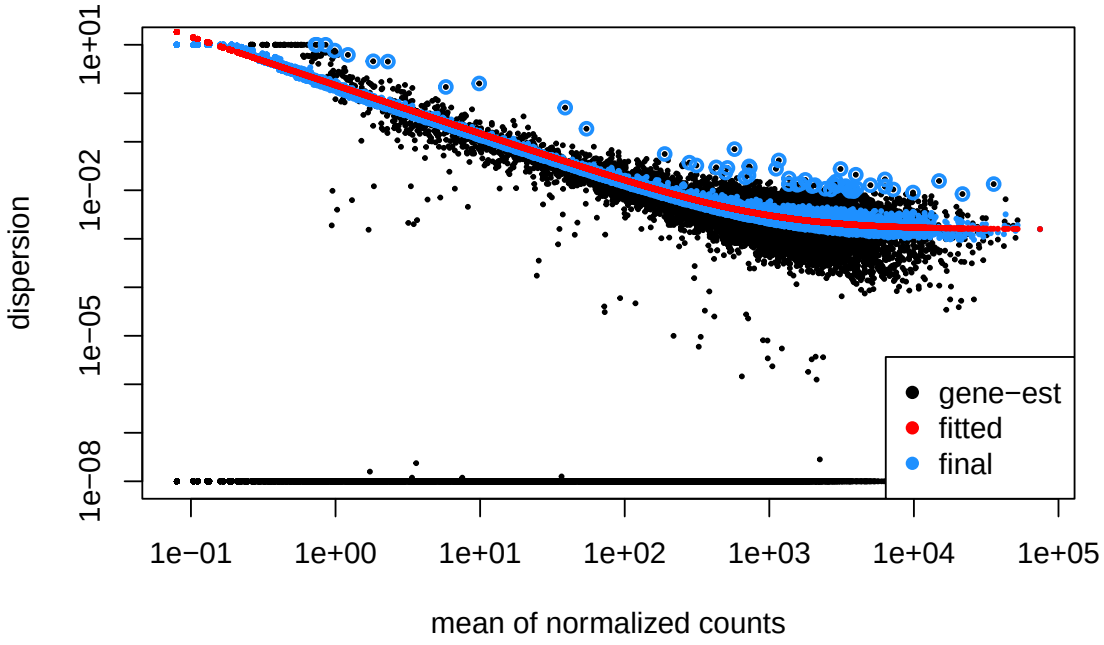





### 4.2.6 MOV10 Differential Expression Analysis: Exploring Dispersion Estimates

Now, let's explore the dispersion estimates for the MOV10 dataset:

```
## Plot dispersion estimates
plotDispEsts(dds)
```




**Since we have a small sample size, for many genes we see quite a bit of shrinkage. Do you think our data are a good fit for the model?**

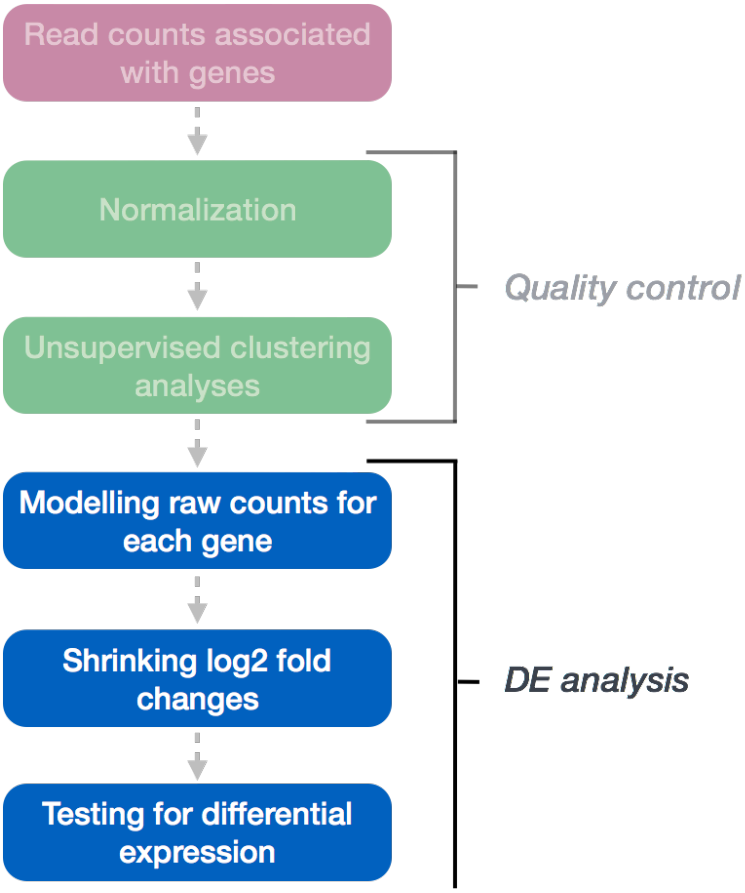
---

# Chapter 5

# Negative Binomial model fitting

## 5.1 Generalized Linear Model fit for each gene

The final step in the DESeq2 workflow is fitting the **Negative Binomial (NB) model** for each gene and performing differential expression testing. This step is crucial for identifying genes that are significantly differentially expressed between experimental conditions.




DESeq2 uses a **Negative Binomial Generalized Linear Model (GLM)** to estimate the counts for each gene. A GLM

is a statistical method that models relationships between variables and is an extension of linear regression. It is suitable for handling non-normally distributed data, such as RNA-seq counts, which often exhibit overdispersion (variance > mean).

To fit the NB GLM, DESeq2 requires two key parameters:

- **Size factor**, which accounts for differences in sequencing depth across samples.
- **Dispersion estimate**, which measures variability in gene expression across replicates.

The model incorporates the estimated dispersion and the design matrix specifying the experimental conditions and covariates.

### 5.1.1 Negative Binomial Model Formula

DESeq2 models RNA-seq counts as follows:

$$Y_{ij} \sim \text{NB}\left(\mu_{ij}, \alpha_i\right)$$

Where:

- $Y_{ij}$ is the observed counts for gene $i$ in sample $j$
- $\mu_{ij}$ is the expected normalized counts for gene $i$ in sample $j$
- $\alpha_i$ is the dispersion parameter for gene $i$, which has been estimated

The expected mean $\mu_{ij}$ is modeled as:

$$\mu_{ij} = sizeFactor_j \times q_{ij}$$

Where:

- $sizeFactor_j$ normalizes for differences in sequencing depth across samples
- $q_{ij}$ represents the true underlying expression level of gene $i$ in sample $j$, typically modeled as a function of covariates (such as experimental conditions).

### 5.1.2 Estimating Beta Coefficients Using the Design Matrix

In DESeq2, **beta coefficients** ($\beta$) are estimated using the **design matrix**, which captures the experimental conditions for each sample (e.g., control, treatment). These coefficients represent the **log2 fold changes** in gene expression between different conditions. The estimation process involves fitting a **negative binomial generalized linear model (GLM)** to the observed count data for each gene, where the expected mean expression $\mu_{ij}$ is modeled as a function of the design matrix $X_j$ and the beta coefficients $\beta_i$.

### 5.1.3 Step-by-Step Process:

1. **Define the Design Matrix $X_j$:** The design matrix $X$ includes the covariates (experimental conditions) for each sample. Each row of the matrix corresponds to a sample, and each column corresponds to a covariate (e.g., intercept, control, treatment). This matrix is generated using the **`model.matrix()`** function in R.

```
X <- model.matrix(~ sampletype, data = meta)

# OR

X <- model.matrix(design(dds), data = colData(dds))
```

2. **Link Mean Expression to the Design Matrix:** The expected mean expression for **gene** $i$ in **sample** $j$ is modeled as:

$$\log_2(\mu_{ij}) = \beta_i X_j$$

Or equivalently:

$$\mu_{ij} = 2^{\beta_i X_j}$$

Here:

- $X_j$ is the row of the design matrix for sample $j$ (it contains the covariates for that sample)
- $\beta_i$ is the vector of log2 fold changes (beta coefficients) for gene $i$.

3. **Estimate the Beta Coefficients:** "DESeq2 estimates the **beta coefficients** by fitting a **negative binomial GLM** to the data using **Maximum Likelihood Estimation (MLE)**. DESeq2 substitutes the mean ($\mu_{ij}$) with $2^{\beta_i X_j}$ in the negative binomial model, using MLE to predict expected counts based on experimental covariates and estimate gene expression changes while controlling for design factors.

The model is fit using the **`DESeq()`** function in R, which carries out the entire estimation process:

```
dds <- DESeq(dds)
```

```
## using pre-existing size factors

## estimating dispersions

## found already estimated dispersions, replacing these

## gene-wise dispersion estimates

## mean-dispersion relationship
```

```
## final dispersion estimates


## fitting model and testing
```

4. **Extract the Beta Coefficients:** Once the model is fitted, the estimated beta coefficients (log2 fold changes) for each gene can be extracted using the `coef()` function:

```
beta_coefficients <- coef(dds)
```

This gives a matrix of estimated beta coefficients, where each row corresponds to a gene and each column corresponds to a covariate in the design matrix.

### 5.1.4 Example:

For example, if the design matrix includes an intercept (control) and two conditions (MOV10 knockdown and MOV10 overexpression), the beta coefficients $\beta_i$ for gene $i$ would represent:

- $\beta_{i1}$: the log2 fold change for the control group (intercept)
- $\beta_{i2}$: the log2 fold change for the MOV10 knockdown group
- $\beta_{i3}$: the log2 fold change for the MOV10 overexpression group.

### 5.1.5 Calculation:

To calculate the fitted (expected) log2 counts for **gene** $i$ in **sample** $j$, you can take the **dot product** of the row of the design matrix $X_j$ for that sample and the beta coefficients $\beta_i$ for that gene:

$$\log_2(\mu_{ij}) = \beta_i X_j = \beta_{i1} X_{j1} + \beta_{i2} X_{j2} + \cdots + \beta_{iP} X_{jP}$$

Where:

- $P$ is the number of covariates in the design matrix

The expected mean $\mu_{ij}$ on the original count scale is obtained by exponentiating the log2-scale fitted values:

$$\mu_{ij} = 2^{\beta_i X_j}$$

**Exercise** points = +1

In the DESeq2 workflow, what is the purpose of the design matrix in the context of fitting the Negative Binomial model for each gene?

- Ans:

### 5.1.6 Matrix Multiplication Form:

If you want to calculate the fitted values for all genes and samples at once, you can express the model as a **matrix multiplication**:

$$\log_2(\mu) = \beta X^T$$

This computes the expected log2 counts for all genes across all samples.

In R, you can calculate the fitted log2 counts for all genes and samples using:

```
log2_fitted <- beta_coefficients %*% t(X)
```

This step multiplies the beta coefficients matrix by the transpose of the design matrix to compute the fitted log2 counts.

### 5.1.7 Log2 Fold Change and Adjustments

The $\beta$ coefficents are the estimates for the **log2 fold changes** for each sample group. However, **log2 fold changes** are inherently noisier when counts are low due to the large dispersion we observe with low read counts. To avoid this, the **log2 fold changes calculated by the model need to be adjusted**.

## 5.2 Shrunken log2 foldchanges (LFC)

To generate more accurate LFC estimates, DESeq2 allows for the **shrinkage of the LFC estimates toward zero** when the information for a gene is low, which could include:

- Low counts
- High dispersion values

As with the shrinkage of dispersion estimates, LFC shrinkage uses **information from all genes** to generate more accurate estimates. Specifically, the distribution of LFC estimates for all genes is used (as a prior) to shrink the LFC estimates of genes with little information or high dispersion toward more likely (lower) LFC estimates.

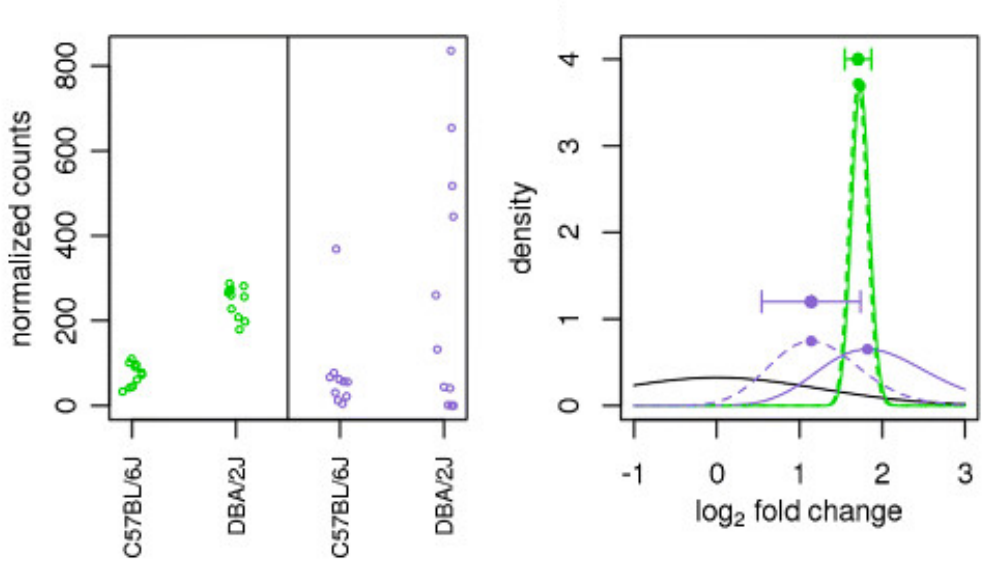



*Illustration taken from the DESeq2 paper.*

For example, in the figure above, the green gene and purple gene have the same mean values for the two sample groups (C57BL/6J and DBA/2J), but the green gene has little variation while the purple gene has high levels of variation. For the green gene with low variation, the **unshrunken LFC estimate** (vertex of the green **solid line**) is very similar to the shrunken LFC estimate (vertex of the green dotted line), but the LFC estimates for the purple gene are quite different due to the high dispersion. So even though two genes can have similar normalized count values, they can have differing degrees of LFC shrinkage. Notice the **LFC estimates are shrunken toward the prior (black solid line)**.

**Generating Shrunken LFC Estimates** To generate the shrunken log2 fold change estimates, you have to run an additional step on your results object (that we will create below) with the function `lfcShrink()`

### 5.2.1 Hypothesis Testing

In DESeq2, hypothesis testing evaluates whether changes in gene expression between conditions are statistically significant. The **null hypothesis** assumes no differential expression between groups, meaning the log2 fold change is zero ($LFC = 0$).

## 5.3 Statistical test for LFC estimates: Wald test

In DESeq2, the Wald test is the default used for hypothesis testing when comparing two groups. The Wald test is a test usually performed on the LFC estimates.

DESeq2 implements the Wald test by:

- Taking the LFC and dividing it by its standard error, resulting in a z-statistic
- The z-statistic is compared to a standard normal distribution, and a p-value is computed reporting the probability that a z-statistic at least as extreme as the observed value would be selected at random
- If the p-value is small we reject the null hypothesis (LFC = 0) and state that there is evidence against the null (i.e. the gene is differentially expressed).

## 5.4 MOV10 Differential Expression Analysis: Control versus Overexpression

We have three sample classes so we can make three possible pairwise comparisons:

1. Control vs. Mov10 overexpression
2. Control vs. Mov10 knockdown
3. Mov10 knockdown vs. Mov10 overexpression

**We are really only interested in #1 and #2 from above**. Using the design formula we provided `~ sampletype`, indicating that this is our main factor of interest.

### 5.4.1 Creating Contrasts for Hypothesis Testing

In DESeq2, **contrasts** specify which groups to compare for differential expression testing. You can define contrasts in two ways:

1. **Default Comparison:** DESeq2 automatically uses the alphabetically first level of the factor as the **baseline**.
2. **Manual Specification:** You can manually specify the comparison using the `contrast` argument in the `results()` function. The `contrast` argument takes a vector of three elements. The first element is the name of the factor (design). The second and third elements listed in the contrast are the names of the numerator and denominator level for the fold change, respectively.

#### Building the Results Table

To build the results table, we use the `results()` function. You can specify the `contrast` to be tested using the contrast argument. In this example, we'll save the unshrunken and shrunken results of **Control vs. Mov10 overexpression** to different variables. We'll also set the `alpha` to 0.05, which is more stringent than the default value of 0.1.

```
# define contrasts
contrast_oe <- c("sampletype",
                 "MOV10_overexpression",
                 "control")

# extract results table
res_tableOE_unshrunken <- results(dds,
                                  contrast = contrast_oe,
                                  alpha = 0.05)

resultsNames(dds)
```

```
## [1] "Intercept"
## [2] "sampletype_MOV10_knockdown_vs_control"
## [3] "sampletype_MOV10_overexpression_vs_control"
```

```
# shrink log2 fold changes
res_tableOE <- lfcShrink(dds = dds,
                coef = "sampletype_MOV10_overexpression_vs_control",
                res = res_tableOE_unshrunken)
```

```
## using 'apeglm' for LFC shrinkage. If used in published research, please cite:
##     Zhu, A., Ibrahim, J.G., Love, M.I. (2018) Heavy-tailed prior distributions for
##     sequence count data: removing the noise and preserving large differences.
##     Bioinformatics. https://doi.org/10.1093/bioinformatics/bty895
```

```
# save the results for future use
saveRDS(res_tableOE, file = "data/res_tableOE.RDS")
```

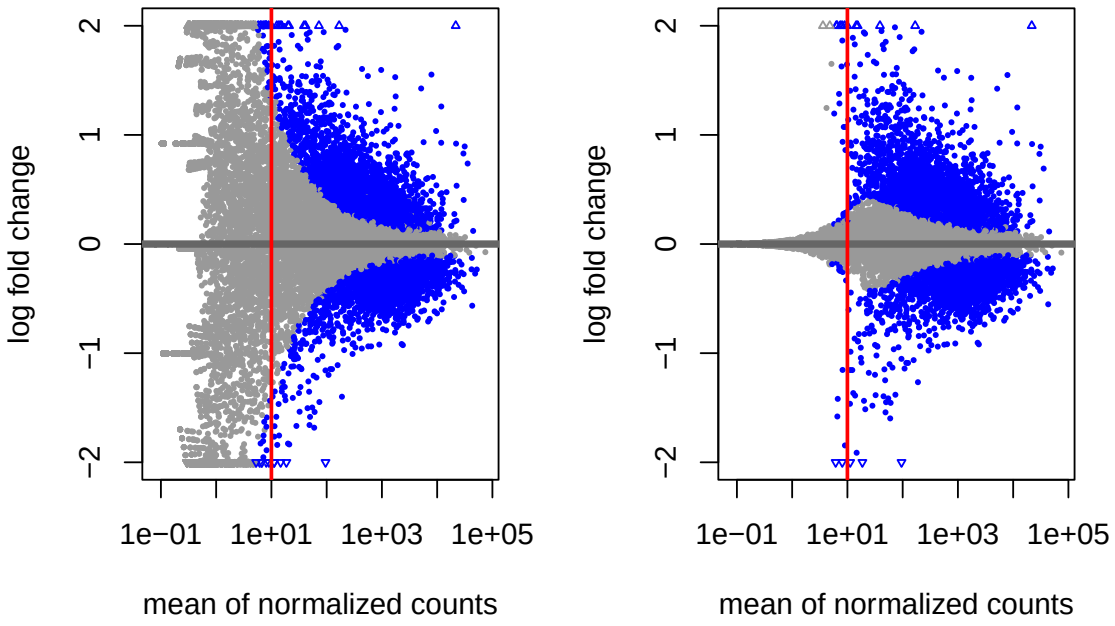
**The order of the names determines the direction of fold change that is reported.** The name provided in the second element is the level that is used as baseline. So for example, if we observe a log2 fold change of -2 this would mean the gene expression is lower in Mov10_oe relative to the control.

### 5.4.2 MA Plot

A plot that can be useful to exploring our results is the MA plot. The MA plot shows the mean of the normalized counts versus the log2 foldchanges for all genes tested. The genes that are significantly DE are colored to be easily identified. This is also a great way to illustrate the effect of LFC shrinkage. The DESeq2 package offers a simple function to generate an MA plot.

**Shrunken & Unshrunken Results:**

```
par(mfrow = c(1,2))

plotMA(res_tableOE_unshrunken, ylim=c(-2,2))

abline(v = 10,col="red",lwd = 2)

# Shrunken results:
plotMA(res_tableOE, ylim=c(-2,2))

abline(v = 10,col="red",lwd = 2)
```




**MOV10 DE Analysis: Exploring the Results**

The results table in `DESeq2` looks similar to a data.frame and can be treated like one for accessing or subsetting data. However, it is stored as a `DESeqResults` object, which is important to keep in mind when working with visualization tools.

```
class(res_tableOE)
```

```
## [1] "DESeqResults"
## attr(,"package")
## [1] "DESeq2"
```

Let's go through some of the columns in the results table to get a better idea of what we are looking at. To extract information regarding the meaning of each column we can use `mcols()`:

```
mcols(res_tableOE, use.names = T)
```

```
## DataFrame with 5 rows and 2 columns
##                        type             description
##                 <character>             <character>
## baseMean       intermediate mean of normalized c..
## log2FoldChange      results log2 fold change (MA..
## lfcSE               results posterior SD: sample..
## pvalue              results Wald test p-value: s..
## padj                results   BH adjusted p-values
```

Now let's take a look at what information is stored in the results:

```
head(res_tableOE)
```

```
## log2 fold change (MAP): sampletype MOV10_overexpression vs control
## Wald test p-value: sampletype MOV10 overexpression vs control
## DataFrame with 6 rows and 5 columns
##                baseMean log2FoldChange     lfcSE    pvalue      padj
##               <numeric>      <numeric> <numeric> <numeric> <numeric>
## 1/2-SBSRNA4  45.652040     0.16858121  0.209275 0.1610752 0.2655661
## A1BG         61.093102     0.13623770  0.176713 0.2401909 0.3603094
## A1BG-AS1    175.665807    -0.04016953  0.116916 0.6789312 0.7748342
## A1CF          0.237692     0.00626568  0.210057 0.7945932        NA
## A2LD1        89.617985     0.29007260  0.195380 0.0333343 0.0741149
## A2M           5.860084    -0.09623512  0.233587 0.1057823 0.1900466
```

```
names(res_tableOE)
```

```
## [1] "baseMean"       "log2FoldChange" "lfcSE"          "pvalue"
## [5] "padj"
```

**Interpreting p-values Set to NA**

In some cases, p-values or adjusted p-values may be set to NA for a gene. This happens in three scenarios:

1. **Zero counts:** If all samples have zero counts for a gene, its baseMean will be zero, and the log2 fold change, p-value, and adjusted p-value will all be set to NA.
2. **Outliers:** If a gene has an extreme count outlier, its p-values will be set to NA. These outliers are detected using Cook's distance.
3. **Low counts:** If a gene is filtered out by independent filtering for having a low mean normalized count, only the adjusted p-value will be set to NA.

### 5.4.3 Multiple Testing Correction

When testing many genes, using raw **p-values** increases the chance of false positives (genes appearing significant by chance). If we used the `p-value` directly from the Wald test with a significance cut-off of $p < 0.05$, that means there is a 5% chance it is a false positives. Each p-value is the result of a single test (single gene). The more genes we test, the more we inflate the false positive rate. This issue is the known as the **multiple testing problem.** FFor example, if we test 20,000 genes for differential expression, at $p < 0.05$ we would expect to find 1,000 genes by chance. If we found 3000 genes to be differentially expressed total, 150 of our genes are false positives. We would not want to sift through our "significant" genes to identify which ones are true positives.

DESeq2 addresses this by removing genes with low counts or outliers before testing and by applying the **Benjamini-Hochberg (BH)** method to control the **false discovery rate (FDR)**.

### 5.4.4 Benjamini-Hochberg Adjustment

The **BH-adjusted p-value** is calculated as:

1. Sort p-values from smallest to largest.
2. Assign ranks to each p-value $p_i$.
3. Compute the adjusted p-value:

$$p_i^{adj} = p_i \times \frac{m}{i}$$

Where:

- $p_i$ is the raw p-value
- $m$ is the total number of genes tested
- $i$ is the rank of the p-value.

**Adjusted p-values** are compared to a significance threshold (e.g., 0.05) to identify significant genes. The **Bonferroni method**, another correction, is more conservative and less commonly used due to a higher risk of false negatives.

In most cases, we should use the **adjusted p-values** (BH-corrected) to identify significant genes.

### 5.4.5 MOV10 DE Analysis: Control vs. Knockdown

After examining the overexpression results, let's move on to the comparison between **Control vs. Knockdown**. We'll use contrasts in the `results()` function to extract the results table and store it in the `res_tableKD` variable. You can also use `coef` to specify the contrast directly in the `lfcShrink()` function.

```
# define contrast
contrast_kd <-  c("sampletype",
                  "MOV10_knockdown",
                  "control")

# extract results table
res_tableKD_unshrunken <- results(dds,
                                  contrast = contrast_kd,
                                  alpha = 0.05)

resultsNames(dds)
```

```
## [1] "Intercept"
## [2] "sampletype_MOV10_knockdown_vs_control"
## [3] "sampletype_MOV10_overexpression_vs_control"
```

```
# use the `coef` argument to specify the contrast directly
res_tableKD <- lfcShrink(dds, coef = 2,
                         res = res_tableKD_unshrunken)
```

```
## using 'apeglm' for LFC shrinkage. If used in published research, please cite:
##     Zhu, A., Ibrahim, J.G., Love, M.I. (2018) Heavy-tailed prior distributions for
##     sequence count data: removing the noise and preserving large differences.
##     Bioinformatics. https://doi.org/10.1093/bioinformatics/bty895
```

```
## Save results for future use
saveRDS(res_tableKD, file = "data/res_tableKD.RDS")
```

## 5.5 Summarizing Results

To summarize the results, DESeq2 offers the `summary()` function, which conveniently reports the number of genes that are significantly differentially expressed at a specified threshold (`default FDR < 0.05`). Note that, even though the output refers to p-values, it actually summarizes the results using **adjusted p-values (padj/FDR)**.

Let's start by summarizing the results for the OE vs. control comparison:

```
## Summarize results
summary(res_tableOE)
```

```
##
## out of 19748 with nonzero total read count
## adjusted p-value < 0.05
## LFC > 0 (up)       : 3103, 16%
## LFC < 0 (down)     : 3408, 17%
## outliers [1]       : 0, 0%
## low counts [2]     : 4171, 21%
## (mean count < 5)
## [1] see 'cooksCutoff' argument of ?results
## [2] see 'independentFiltering' argument of ?results
```

In addition to reporting the number of up- and down-regulated genes at the default significance threshold, this function also provides information on:

- **Number of genes tested** (genes with non-zero total read count)
- **Number of genes excluded** from multiple test correction due to low mean counts

### 5.5.1 Extracting Significant Differentially Expressed Genes

In some cases, using only the FDR threshold doesn't sufficiently reduce the number of significant genes, making it difficult to extract biologically meaningful results. To increase stringency, we can apply an additional fold change threshold.

Although the summary() function doesn't include an argument for fold change thresholds, we can define our own criteria.

Let's start by setting the thresholds for both adjusted p-value (FDR $< 0.05$) and log2 fold change ($|\log2FC| > 0.58$, corresponding to a 1.5-fold change):

```
### Set thresholds
padj.cutoff <- 0.05
lfc.cutoff <- 0.58
```

Next, we'll convert the results table to a tibble for easier subsetting:

```
res_tableOE_tb <- res_tableOE %>%
  data.frame() %>%
  rownames_to_column(var = "gene") %>%
  as_tibble()
```

Now, we can filter the table to retain only the genes that meet the significance and fold change criteria:

```
sigOE <- res_tableOE_tb %>%
  dplyr::filter(padj < padj.cutoff &
                abs(log2FoldChange) > lfc.cutoff)

# Save the results for future use
saveRDS(sigOE,"data/sigOE.RDS")
```

**Exercise** points = +3

How many genes are differentially expressed in the **Overexpression vs. Control** comparison based on the criteria we just defined? Does this reduce the number of significant genes compared to using only the FDR threshold?

```
# Your code here
```

**Does this reduce our results?**

```
# Your code here
```

### 5.5.2 MOV10 Knockdown Analysis: Control vs. Knockdown

Next, let's perform the same analysis for the **Control vs. Mov10 knockdown** comparison. We'll use the same thresholds for adjusted p-value (FDR $< 0.05$) and log2 fold change ($|\text{log2FC}| > 0.58$).

```
res_tableKD_tb <- res_tableKD %>%
  data.frame() %>%
  rownames_to_column(var = "gene") %>%
  as_tibble()

sigKD <- res_tableKD_tb %>%
         dplyr::filter(padj < padj.cutoff &
         abs(log2FoldChange) > lfc.cutoff)

# We'll save this object for use in the homework
saveRDS(sigKD,"data/sigKD.RDS")
```

**How many genes are differentially expressed in the Knockdown compared to Control?**

```
# Your code here
```

With our subset of significant genes identified, we can now proceed to visualize the results and explore patterns of differential expression.

```
## Save all objects for later
save.image()
```

# Chapter 6

# Visualizing RNA-seq results

During this lesson, we will get you started with some basic and more advanced visualization techniques to explore and interpret the results of your differential expression analysis.

Let's start by loading a few libraries:

```
# load libraries
library(tidyverse)
library(ggplot2)
library(ggrepel)
library(RColorBrewer)
library(DESeq2)
library(pheatmap)
library(dplyr)
```

We will be working with three different data objects we have already created in earlier lessons:

- Metadata for our samples (a dataframe): `meta`
- Normalized expression data for every gene in each of our samples (a matrix): `normalized_counts`
- Tibble versions of the DESeq2 results we generated in the last lesson: `res_tableOE_tb` and `res_tableKD_tb`

Let's create tibble objects from the `meta` and `normalized_counts` data frames before we start plotting. This will enable us to use the `tidyverse` functionality more easily.

Check to make sure the column names of the `normalized_counts` matrix are the same as the row names of the `meta` data frame. If they are not, you will need to reorder the columns of `normalized_counts` to match the row names of `meta`.

```
# Read in the metadata
meta <- read.table("data/Mov10_full_meta.txt",
                   header=T, row.names=1)

# Create a tibble for meta data
mov10_meta <- meta %>%
  rownames_to_column(var="samplename") %>%
  as_tibble()
```

```
# read in the normalized counts
normalized_counts <- read.delim("data/normalized_counts.txt", row.names=1)

all(mov10_meta$samplename == colnames(normalized_counts))
```

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

```
# Create a tibble for normalized_counts
normalized_counts <- normalized_counts %>%
  data.frame() %>%
  rownames_to_column(var="gene") %>%
  as_tibble()
```

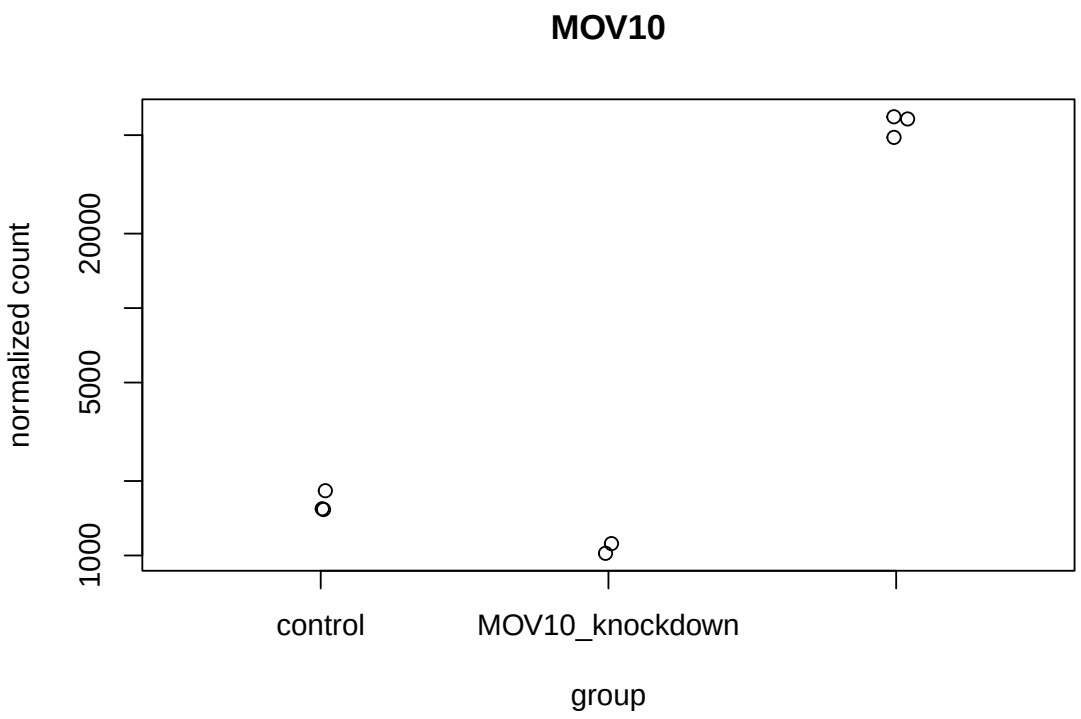
### 6.0.1 Plotting signicant DE genes

One way to visualize results would be to simply plot the expression data for a handful of genes. We could do that by picking out specific genes of interest or selecting a range of genes.

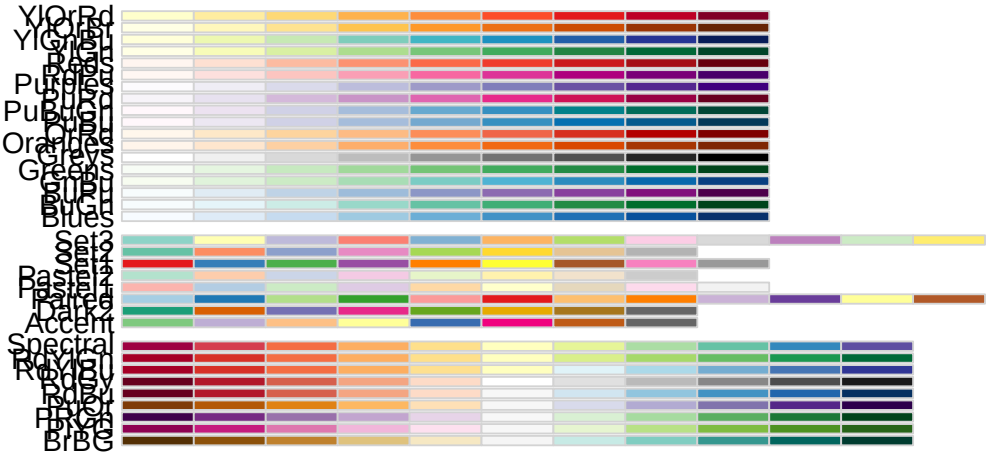
#### 6.0.1.1 Using DESeq2 `plotCounts()` to plot expression of a single gene

To pick out a specific gene of interest to plot, for example Mov10, we can use the `plotCounts()` from DESeq2. `plotCounts()` is a function that allows us to plot the normalized counts for a single gene across all samples.

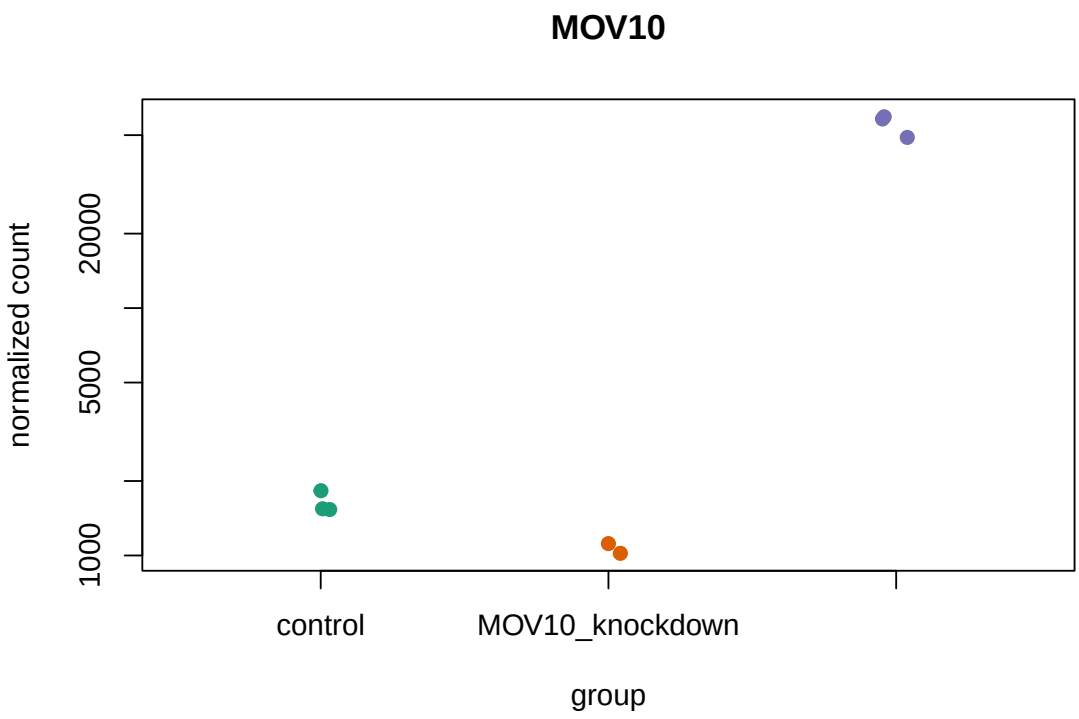
```
plotCounts(dds,
           gene="MOV10",
           intgroup="sampletype")
```




```
# We can also color the points by sample type
sampletype = as.factor(mov10_meta$sampletype)

library(RColorBrewer)

display.brewer.all()
```




```
col = brewer.pal(8,"Dark2")
palette(col)

plotCounts(dds, gene="MOV10",
           intgroup="sampletype",
           col = as.numeric(sampletype),
           pch = 19)
```

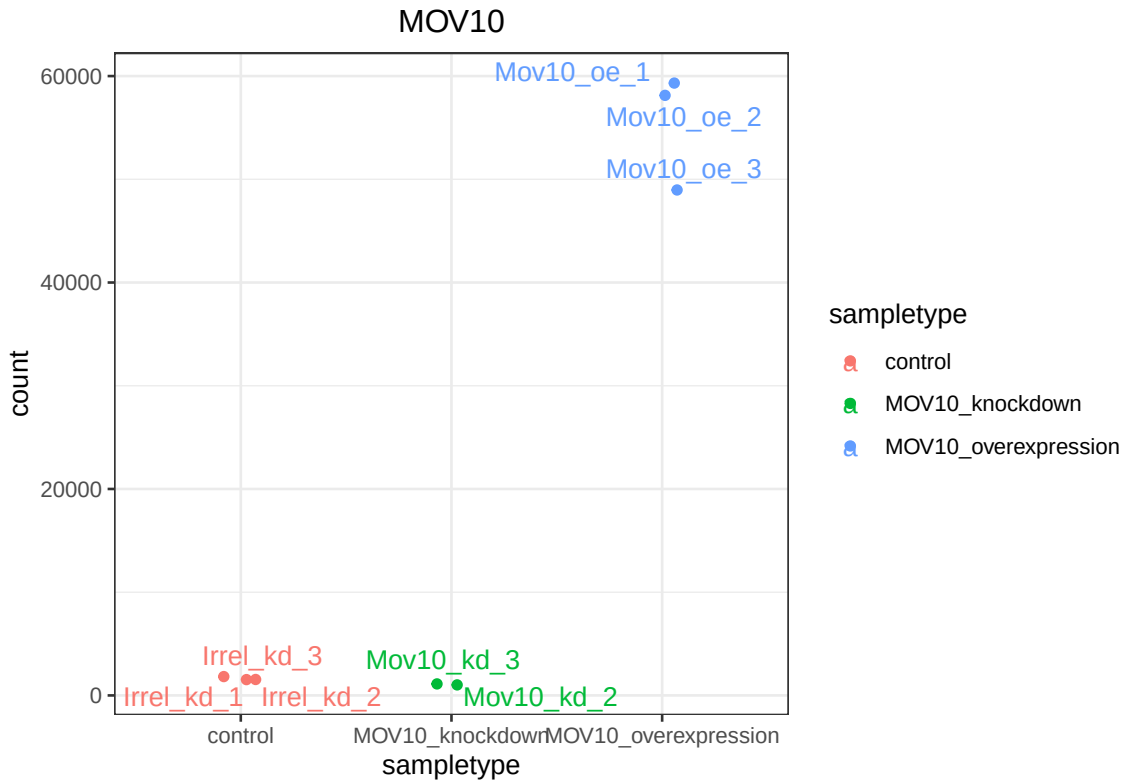



**This function only allows for plotting the counts of a single gene at a time.**

#### 6.0.1.2 Using ggplot2 to plot expression of a single gene

We can also use ggplot2 to plot the `MOV10` counts. We can save the output of `plotCounts()` to a variable specifying the `returnData=TRUE` argument. This will save the normalized counts for the gene `MOV10` to a data frame object. We can then use ggplot2 to plot the normalized counts for `MOV10` across all samples.

```
# Save `plotCounts()` to a data frame object
d <- plotCounts(dds, gene = "MOV10", intgroup = "sampletype",
                returnData=TRUE)

# Plot using ggplot2
ggplot(d, aes(x = sampletype, y = count, color = sampletype)) +
  geom_point(position = position_jitter(w = 0.1,h = 0)) +
  geom_text_repel(aes(label = rownames(d))) +
  theme_bw() +
  ggtitle("MOV10") +
  theme(plot.title = element_text(hjust = 0.5))
```




Note that in the plot below (code above), we are using `geom_text_repel()` from the `ggrepel` package to label our individual points on the plot.

#### 6.0.1.3 Using ggplot2 to plot multiple genes (e.g. top 20)

Often it is helpful to check the expression of multiple genes of interest at the same time. This often first requires some data wrangling.

We are going to plot the normalized count values for the **top 20 differentially expressed genes (by padj values)**.

To do this, we first need to determine the gene names of our top 20 genes by ordering our results and extracting the top 20 genes (by padj values):

```
res_tableOE <- readRDS("data/res_tableOE.RDS")

res_tableOE_tb <- res_tableOE %>%
  data.frame() %>%
  rownames_to_column(var="gene") %>%
  as_tibble()

## Order results by padj values
top20_sigOE_genes <- res_tableOE_tb %>%
        arrange(padj) %>%
#Arrange rows by padj values
        pull(gene) %>%
#Extract character vector of ordered genes
```

```
        head(n=20)
#Extract the first 20 genes
```

Then, we can extract the normalized count values for these top 20 genes:

```
## Normalized counts for top 20 significant genes
top20_sigOE_norm <- normalized_counts %>%
        filter(gene %in% top20_sigOE_genes)
```

Now that we have the normalized counts for each of the top 20 genes for all 8 samples, to plot using `ggplot()`, we need to `pivot_longer` top20_sigOE_norm from a wide format to a long format so the counts for all samples will be in a single column to allow us to give ggplot the one column with the values we want it to plot.

The `pivot_longer()` function in the **tidyr** package will perform this operation and will output the normalized counts for all genes for *Mov10_oe_1* listed in the first 20 rows, followed by the normalized counts for *Mov10_oe_2* in the next 20 rows, so on and so forth.

| | Mov10_oe_1 | Mov10_oe_2 | Mov10_oe_3 | Irrel_kd_1 | Irrel_kd_2 | Irrel_kd_3 |
|---|---|---|---|---|---|---|
| MOV10 | 59319.6663 | 58130.0281 | 48964.1381 | 1544.010015 | 1533.405731 | 1826.761190 |
| H1F0 | 13694.1471 | 13185.0041 | 12283.1157 | 4280.106238 | 4296.860505 | 4787.558609 |
| HSPA6 | 491.8408 | 444.4083 | 384.0865 | 7.127571 | 3.116678 | 6.686534 |
| HIST1H... | 2666.4264 | 2745.8719 | 2610.5638 | 931.929876 | 920.458996 | 928.090971 |
| TXNIP | 7154.5781 | 7528.1688 | 8113.2527 | 2722.732028 | 2846.566195 | 2915.328985 |
| NEAT1 | 29542.0743 | 28240.4493 | 30082.6924 | 14847.620827 | 15355.874063 | 16234.905450 |
| KLF10 | 2183.7400 | 2232.7499 | 2433.0577 | 931.929876 | 1020.192702 | 941.464039 |
| INSIG1 | 16559.4736 | 18116.3295 | 20018.4028 | 7361.889643 | 7576.644986 | 7888.773250 |
| NR1D1 | 1663.6038 | 1775.8483 | 1502.6809 | 563.078090 | 525.679743 | 549.633125 |
| WDFY1 | 1778.4499 | 1756.2158 | 1895.9487 | 834.816725 | 857.086537 | 857.213706 |
| HSPA1A | 40305.9818 | 43833.1110 | 43926.6375 | 24397.674692 | 22399.567057 | 22086.960331 |
| HMGCS1 | 15683.1481 | 15828.2516 | 17752.1397 | 8750.874994 | 8411.914775 | 8715.228898 |
| HSPA1B | 39351.4278 | 38343.1519 | 41718.5229 | 24109.899023 | 21831.292711 | 21228.409318 |
| LAMC1 | 6939.8658 | 6017.3592 | 6720.7480 | 3579.822411 | 3269.395553 | 3448.914428 |
| TMCO1 | 2075.5517 | 1978.4199 | 2156.0870 | 1100.318735 | 1030.581630 | 1060.484351 |
| ADAMTS1 | 13252.2393 | 13267.9960 | 14442.2631 | 6717.735436 | 6454.640792 | 6801.542761 |
| ZFP36L1 | 1838.3695 | 1929.3387 | 1773.5307 | 994.296120 | 947.470208 | 949.487881 |
| C1orf95 | 385.3169 | 399.7890 | 385.6167 | 96.222205 | 98.694813 | 106.984550 |
| MRC2 | 752.3251 | 769.2368 | 661.0572 | 239.664567 | 237.906445 | 243.389851 |
| COL2A1 | 4066.2171 | 4134.4246 | 3825.5624 | 2186.382329 | 2194.141534 | 2356.334712 |

| | gene | samplename | normalized_c |
|---|---|---|---|
| 1 | MOV10 | Mov10_oe_1 | 59319.6 |
| 2 | H1F0 | Mov10_oe_1 | 13694.1 |
| 3 | HSPA6 | Mov10_oe_1 | 491.8 |
| 4 | HIST1H1C | Mov10_oe_1 | 2666.4 |
| 5 | TXNIP | Mov10_oe_1 | 7154.5 |
| 6 | NEAT1 | Mov10_oe_1 | 29542.0 |
| 7 | KLF10 | Mov10_oe_1 | 2183.7 |
| 8 | INSIG1 | Mov10_oe_1 | 16559.4 |
| 9 | NR1D1 | Mov10_oe_1 | 1663.6 |
| 10 | WDFY1 | Mov10_oe_1 | 1778.4 |
| 11 | HSPA1A | Mov10_oe_1 | 40305.9 |
| 12 | HMGCS1 | Mov10_oe_1 | 15683.1 |
| 13 | HSPA1B | Mov10_oe_1 | 39351.4 |
| 14 | LAMC1 | Mov10_oe_1 | 6939.8 |
| 15 | TMCO1 | Mov10_oe_1 | 2075.5 |
| 16 | ADAMTS1 | Mov10_oe_1 | 13252.2 |
| 17 | ZFP36L1 | Mov10_oe_1 | 1838.3 |
| 18 | C1orf95 | Mov10_oe_1 | 385.3 |
| 19 | MRC2 | Mov10_oe_1 | 752.3 |
| 20 | COL2A1 | Mov10_oe_1 | 4066.2 |
| 21 | MOV10 | Mov10_oe_2 | 58130.0 |
| 22 | H1F0 | Mov10_oe_2 | 13185.0 |
| 23 | HSPA6 | Mov10_oe_2 | 444.4 |
| 24 | HIST1H1C | Mov10_oe_2 | 2745.8 |
| 25 | TXNIP | Mov10_oe_2 | 7528.1 |
| 26 | NEAT1 | Mov10_oe_2 | 28240.4 |
| 27 | KLF10 | Mov10_oe_2 | 2232.7 |
| 28 | INSIG1 | Mov10_oe_2 | 18116.3 |
| 29 | NR1D1 | Mov10_oe_2 | 1775.8 |
| 30 | WDFY1 | Mov10_oe_2 | 1756.2 |

```
# Pivot the data frame
pivoted_top20_sigOE <- top20_sigOE_norm %>%
  pivot_longer(colnames(top20_sigOE_norm)[2:9],
               names_to = "samplename",
               values_to = "normalized_counts")

## Check the column header in the "pivoted" data frame
head(pivoted_top20_sigOE)
```

```
## # A tibble: 6 x 3
##   gene    samplename normalized_counts
##   <chr>   <chr>                  <dbl>
## 1 ADAMTS1 Irrel_kd_1             6718.
## 2 ADAMTS1 Irrel_kd_2             6455.
## 3 ADAMTS1 Irrel_kd_3             6802.
## 4 ADAMTS1 Mov10_kd_2             6869.
## 5 ADAMTS1 Mov10_kd_3             8731.
## 6 ADAMTS1 Mov10_oe_1            13252.
```

Now, if we want our counts colored by sample group, then we need to combine the metadata information with the melted normalized counts data into the same data frame for input to `ggplot()`:

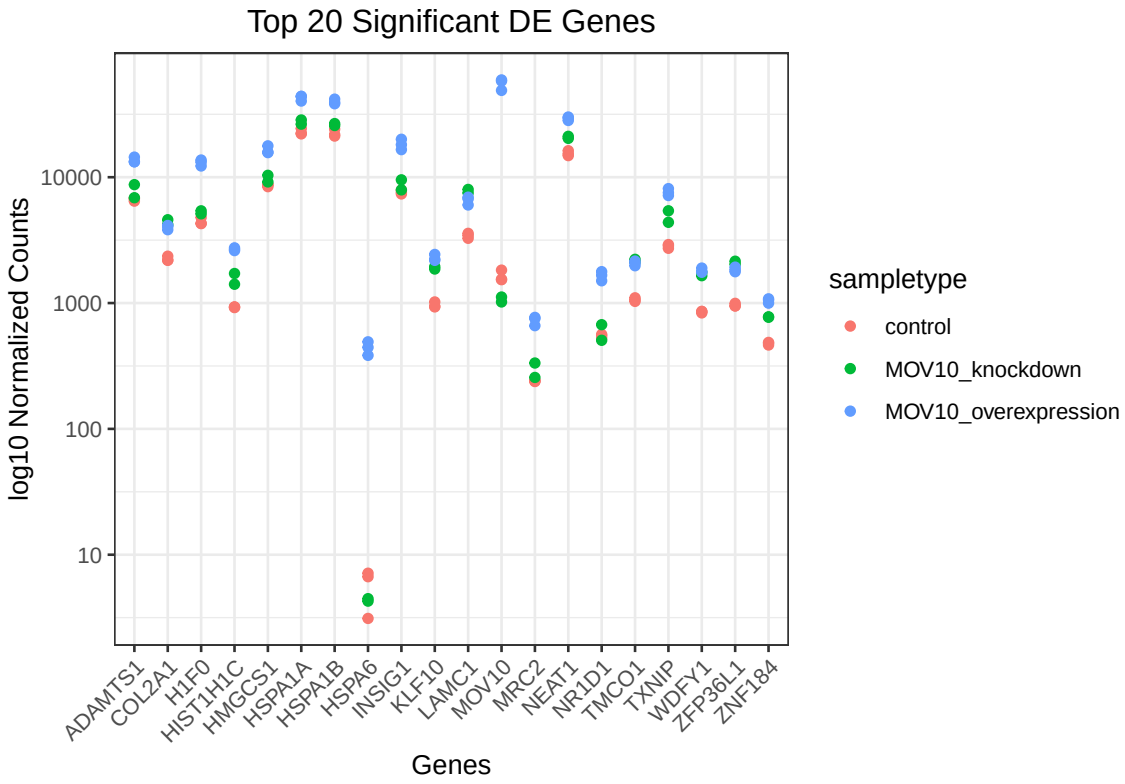
`inner_join(x,y)` will merge 2 data frames by the colname in x that matches a column name in y in this case `samplename` column.

```
pivoted_top20_sigOE <- inner_join(
  mov10_meta,
  pivoted_top20_sigOE)
```

```
## Joining with `by = join_by(samplename)`
```

Now that we have a data frame in a format that can be utilised by ggplot easily, let's plot!

```
## plot using ggplot2
ggplot(pivoted_top20_sigOE) +
        geom_point(aes(x = gene, y = normalized_counts,
          color = sampletype)) +
        scale_y_log10() +
        xlab("Genes") +
        ylab("log10 Normalized Counts") +
        ggtitle("Top 20 Significant DE Genes") +
        theme_bw() +
        theme(axis.text.x = element_text(angle = 45,
                                         hjust = 1)) +
        theme(plot.title = element_text(hjust = 0.5))
```




### 6.0.2 Heatmap

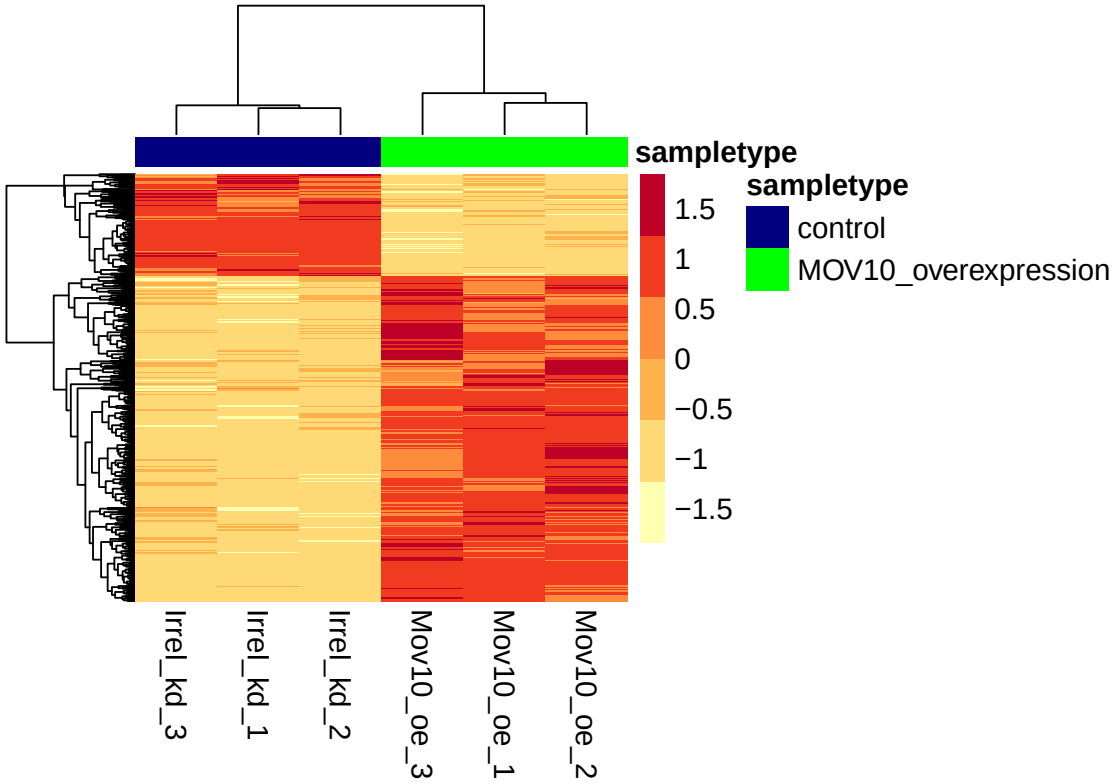
We will plot a heatmap of the `sigOE` genes to visualize their expression across all samples using `pheatmap()`.

```
sigOE = readRDS("data/sigOE.RDS")

norm_OEsig <- normalized_counts[,c(1,2:4,7:9)] %>%
              filter(gene %in% sigOE$gene) %>%
              data.frame() %>%
              column_to_rownames(var = "gene")
```

Now let's draw the heatmap using pheatmap. We can also add annotations to the heatmap to indicate the sample type. The `annotation_col` argument in `pheatmap()`requires a data frame with the same row names as the column names of the matrix used to generate the heatmap.

```
### Annotate our heatmap (optional)
annotation <- mov10_meta %>%
  filter(samplename %in% colnames(norm_OEsig)) %>%
    dplyr::select(samplename, sampletype) %>%
    data.frame(row.names = "samplename")

annotation$sampletype = factor(annotation$sampletype)

### Set a color palette
heat_colors <- brewer.pal(6, "YlOrRd")
```

```
### Set annotation colors
ann_colors = list(
  sampletype = c(control="navy",
  MOV10_overexpression= "green"))

### Run pheatmap
pheatmap(norm_OEsig,
         annotation_colors = ann_colors,
         color = heat_colors,
         cluster_rows = T,
         show_rownames = F,
         annotation_col = annotation,
         border_color = NA,
         fontsize = 12,
         scale = "row",
         fontsize_col = 12)
```




*NOTE:* There are several additional arguments we have included in the function for aesthetics. One important one is `scale="row"`, in which Z-scores are plotted, rather than the actual normalized count value.

### 6.0.3 Volcano plots

A commonly used plot in RNA-seq analysis is the **volcano plot**. in which you have the log transformed adjusted p-values plotted on the y-axis and log2 fold change values on the x-axis.

To generate a volcano plot, we first need to have a column in our results data indicating whether or not the gene is considered differentially expressed based on p-adjusted values.

```
## create a column for thresholding in the results table
res_tableOE_tb <- res_tableOE_tb %>%
                  mutate(threshold_OE = padj < 0.05 &
                  abs(log2FoldChange) >= 0.58)
```

Now we can start plotting. The `geom_point` object is most applicable, as this is essentially a scatter plot:

```
ggplot(res_tableOE_tb) +
        geom_point(aes(x = log2FoldChange,
        y = -log10(padj), colour = threshold_OE)) +
        ggtitle("Mov10 overexpression") +
        xlab("log2 fold change") +
        ylab("-log10 adjusted p-value") +
        theme_bw() +
        theme(legend.position = "none",
        plot.title = element_text(size = rel(1.5),
                                  hjust = 0.5),
        axis.title = element_text(size = rel(1.25)))
```


What if we also wanted to know where the top 10 genes (lowest padj) in our DE list are located on this plot? We could label those dots with the gene name on the volcano plot using `geom_text_repel()`.

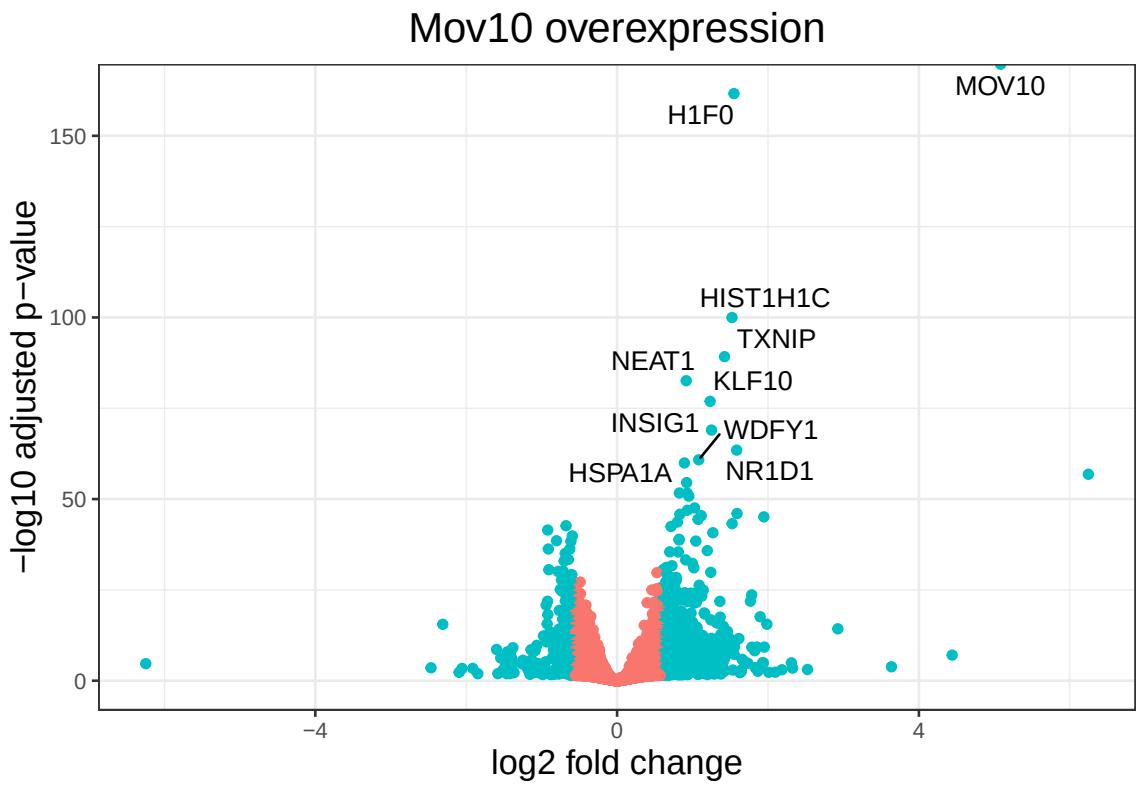
First, we need to order the res_tableOE tibble by `padj`, and add an additional column to it, to include on those gene names we want to use to label the plot.

```
## Create a column to indicate which genes to label
res_tableOE_tb <- res_tableOE_tb %>%
  arrange(padj) %>%
  mutate(genelabels = "")

res_tableOE_tb$genelabels[1:10] <- res_tableOE_tb$gene[1:10]

head(res_tableOE_tb)
```

```
## # A tibble: 6 x 8
##   gene   baseMean log2FoldChange  lfcSE    pvalue      padj threshold_OE genelabels
##   <chr>     <dbl>          <dbl>  <dbl>     <dbl>     <dbl> <lgl>        <chr>
## 1 MOV10    21682.          5.08  0.110  0         0         TRUE         MOV10
## 2 H1F0      7881.          1.55  0.0566 3.00e-166 2.34e-162 TRUE         H1F0
## 3 HIST1~    1741.          1.52  0.0706 2.06e-104 1.07e-100 TRUE         HIST1H1C
## 4 TXNIP     5134.          1.42  0.0697 1.62e- 93 6.33e- 90 TRUE         TXNIP
## 5 NEAT1    21974.          0.916 0.0467 8.28e- 87 2.58e- 83 TRUE         NEAT1
## 6 KLF10     1694.          1.23  0.0652 4.77e- 81 1.24e- 77 TRUE         KLF10
```

Next, we plot it as before with an additiona layer for `geom_text_repel()` wherein we can specify the column of gene labels we just created.

```
ggplot(res_tableOE_tb, aes(x = log2FoldChange,
        y = -log10(padj))) +
        geom_point(aes(colour = threshold_OE)) +
        geom_text_repel(aes(label = genelabels)) +
        ggtitle("Mov10 overexpression") +
        xlab("log2 fold change") +
        ylab("-log10 adjusted p-value") +
        theme_bw() +
        theme(legend.position = "none",
        plot.title = element_text(size = rel(1.5),
                                  hjust = 0.5),
        axis.title = element_text(size = rel(1.25)))
```


Mov10 overexpression
MOV10
H1F0
HIST1H1C
TXNIP
NEAT1
KLF10
INSIG1
WDFY1
HSPA1A
NR1D1
150
100
50
0
−4
0
4
−log10 adjusted p−value
log2 fold change

# Chapter 7

# Summary of DGE workflow

**Homework: modify this file to analyze the MOV dataset, starting with Mov10_full_counts.txt in your data folder. Compare the "MOV10_knockdown" to the "control". Include a heatmap and a volcano plot** points = +10

```
## Setup
library(DESeq2)
```

We have detailed the various steps in a differential expression analysis workflow, providing theory with example code. To provide a more succinct reference for the code needed to run a DGE analysis, we have summarized the steps in an analysis below:

### 7.0.1 1. Import data into dds object:

```
# Check that the row names of the metadata equal the column names
#  of the **raw counts** data
all(colnames(raw_counts) == rownames(metadata))

# Create DESeq2Dataset object
dds <- DESeqDataSetFromMatrix(countData = raw_counts, colData = metadata,
                              design = ~ condition)
```

### 7.0.2 2. Exploratory data analysis (PCA & heirarchical clustering) - identifying outliers and sources of variation in the data:

```
# Transform counts for data visualization
rld <- rlog(dds, blind = TRUE)

# Plot PCA
plotPCA(rld, intgroup = "sampletype")
```

```
# Extract the rlog matrix from the object
rld_mat <- assay(rld)

# Compute pairwise correlation values
rld_cor <- cor(rld_mat)

# Plot heatmap
pheatmap(rld_cor)
```

### 7.0.3 3. Run DESeq2:

```
# **Optional step** - Re-create DESeq2 dataset if the design formula has changed after QC analysis in include

dds <- DESeqDataSetFromMatrix(countData = raw_counts,
                              colData = metadata, design = ~ condition)

# Run DESeq2 differential expression analysis
dds <- DESeq(dds)

#  Output normalized counts to save as a file to access outside RStudio
normalized_counts <- counts(dds, normalized = TRUE)

write.table(normalized_counts, file = "data/normalized_counts.txt",
            sep = "\t", quote = F, col.names = NA)
```

### 7.0.4 4. Check the fit of the dispersion estimates:

```
# Plot dispersion estimates
plotDispEsts(dds)
```

### 7.0.5 5. Create contrasts to perform Wald testing on the shrunken log2 foldchanges between specific conditions:

```
# Output results of Wald test for contrast
contrast <- c("condition", "level_to_compare", "base_level")

res <- results(dds, contrast = contrast)

coef = resultsNames(dds)

res_table <- lfcShrink(dds, coef = coef[2], res = res,
                       type = "apeglm")
```

### 7.0.6 6. Output significant results:

```
### Set thresholds
padj.cutoff <- 0.05
lfc.cutoff <- 0.58 ## change in expression of 1.5

# Turn the results object into a data frame
res_df <- res %>%
  data.frame() %>%
  rownames_to_column(var = "gene")

# Subset the significant results
sig_res <- dplyr::filter(res_df, padj < padj.cutoff &
                  abs(log2FoldChange) > lfc.cutoff)
```

### 7.0.7 7. Visualize results: volcano plots, heatmaps, normalized counts plots of top genes, etc.

### 7.0.8 8. Make sure to output the versions of all tools used in the DE analysis:

```
sessionInfo()
```

```
## R version 4.2.3 (2023-03-15)
## Platform: x86_64-apple-darwin17.0 (64-bit)
## Running under: macOS 14.6.1
##
## Matrix products: default
## LAPACK: /Library/Frameworks/R.framework/Versions/4.2/Resources/lib/libRlapack.dylib
##
## locale:
## [1] en_CA.UTF-8/en_CA.UTF-8/en_CA.UTF-8/C/en_CA.UTF-8/en_CA.UTF-8
##
## attached base packages:
## [1] stats4    stats     graphics  grDevices utils     datasets  methods   base
##
## other attached packages:
##  [1] GOenrichment_0.5.0          GOSemSim_2.24.0
##  [3] GO.db_3.16.0                visNetwork_2.1.2
##  [5] ggraph_2.2.1                igraph_2.1.1
##  [7] fgsea_1.24.0                enrichplot_1.18.4
##  [9] clusterProfiler_4.6.2       org.Hs.eg.db_3.16.0
## [11] AnnotationDbi_1.60.2        ggrepel_0.9.6
## [13] pheatmap_1.0.12             DESeq2_1.38.3
## [15] SummarizedExperiment_1.28.0 Biobase_2.58.0
## [17] MatrixGenerics_1.10.0       matrixStats_1.4.1
## [19] GenomicRanges_1.50.2        GenomeInfoDb_1.34.9
## [21] IRanges_2.32.0              S4Vectors_0.36.2
## [23] BiocGenerics_0.44.0         RColorBrewer_1.1-3
```

```
## [25] lubridate_1.9.3            forcats_1.0.0
## [27] stringr_1.5.1              dplyr_1.1.4
## [29] purrr_1.0.2                readr_2.1.5
## [31] tidyr_1.3.1                tibble_3.2.1
## [33] ggplot2_3.5.1              tidyverse_2.0.0
## [35] bookdown_0.41
##
## loaded via a namespace (and not attached):
##   [1] shadowtext_0.1.4       fastmatch_1.1-4        plyr_1.8.9
##   [4] lazyeval_0.2.2         splines_4.2.3          BiocParallel_1.32.6
##   [7] digest_0.6.37          yulab.utils_0.1.8      htmltools_0.5.8.1
##  [10] viridis_0.6.5          rsconnect_1.3.2        fansi_1.0.6
##  [13] magrittr_2.0.3         memoise_2.0.1          tzdb_0.4.0
##  [16] Biostrings_2.66.0      annotate_1.76.0        graphlayouts_1.0.1
##  [19] bdsmatrix_1.3-7        timechange_0.3.0       colorspace_2.1-1
##  [22] blob_1.2.4             apeglm_1.20.0          xfun_0.49
##  [25] crayon_1.5.3           RCurl_1.98-1.16        jsonlite_1.8.9
##  [28] scatterpie_0.2.4       ape_5.8                glue_1.8.0
##  [31] polyclip_1.10-7        gtable_0.3.6           zlibbioc_1.44.0
##  [34] XVector_0.38.0         DelayedArray_0.24.0    scales_1.3.0
##  [37] DOSE_3.24.2            mvtnorm_1.3-2          DBI_1.2.3
##  [40] Rcpp_1.0.13-1          viridisLite_0.4.2      xtable_1.8-4
##  [43] emdbook_1.3.13         gridGraphics_0.5-1     tidytree_0.4.6
##  [46] bit_4.5.0              htmlwidgets_1.6.4      httr_1.4.7
##  [49] pkgconfig_2.0.3        XML_3.99-0.17          farver_2.1.2
##  [52] sass_0.4.9             locfit_1.5-9.10        utf8_1.2.4
##  [55] ggplotify_0.1.2        tidyselect_1.2.1       labeling_0.4.3
##  [58] rlang_1.1.4            reshape2_1.4.4         munsell_0.5.1
##  [61] tools_4.2.3            cachem_1.1.0           downloader_0.4
##  [64] cli_3.6.3              generics_0.1.3         RSQLite_2.3.4
##  [67] gson_0.1.0             evaluate_1.0.1         fastmap_1.2.0
##  [70] yaml_2.3.10            ggtree_3.6.2           knitr_1.49
##  [73] bit64_4.5.2            fs_1.6.5               tidygraph_1.3.1
##  [76] KEGGREST_1.38.0        nlme_3.1-166           aplot_0.2.3
##  [79] compiler_4.2.3         rstudioapi_0.17.1      png_0.1-8
##  [82] treeio_1.22.0          tweenr_2.0.3           geneplotter_1.76.0
##  [85] bslib_0.8.0            stringi_1.8.4          lattice_0.22-6
##  [88] Matrix_1.5-3           vctrs_0.6.5            pillar_1.9.0
##  [91] lifecycle_1.0.4        jquerylib_0.1.4        data.table_1.16.2
##  [94] cowplot_1.1.3          bitops_1.0-9           patchwork_1.3.0
##  [97] qvalue_2.30.0          R6_2.5.1               gridExtra_2.3
## [100] codetools_0.2-20       MASS_7.3-58.2          withr_3.0.2
## [103] GenomeInfoDbData_1.2.9 parallel_4.2.3         hms_1.1.3
## [106] grid_4.2.3             ggfun_0.1.7            HDO.db_0.99.1
## [109] coda_0.19-4.1          rmarkdown_2.29         ggnewscale_0.5.0
## [112] bbmle_1.0.25.1         ggforce_0.4.2          numDeriv_2016.8-1.1
```

# Chapter 8

# Functional analysis of RNAseq data

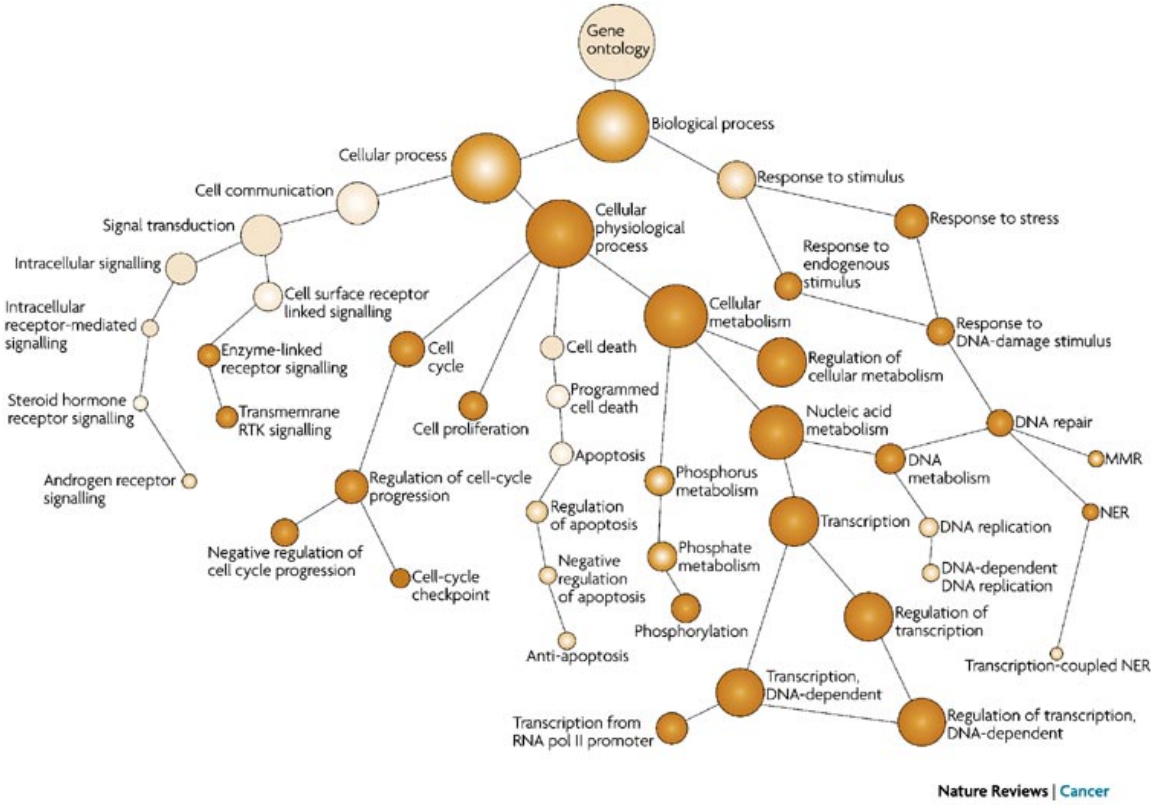
### 8.0.1 Ontology and GO Terms

**Ontology**: An ontology is a representation of something we know about. Ontologies consist of representations of things that are detectable or directly observable and the relationships between those things. The **Gene Ontology (GO)** project provides an ontology of defined terms representing gene product properties. These terms are connected to each other via formally defined relationships, allowing for consistent annotation and analysis of gene functions across different species and research areas.

### 8.0.2 GO Terms (Gene Ontology Terms)

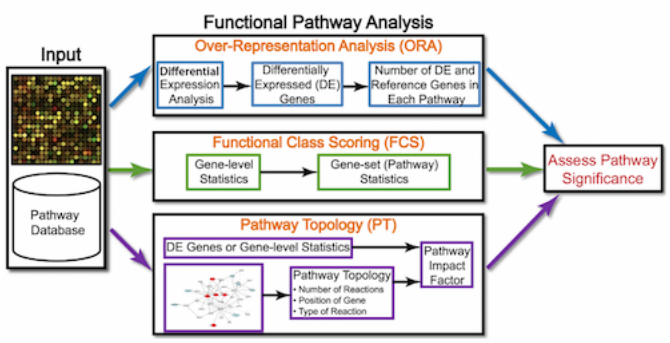
The GO terms describe three aspects of gene products:

- **Biological Process**: Refers to the biological objectives achieved by gene products, such as "cell division" or "response to stress."




- **Molecular Function**: Describes the biochemical activities at the molecular level, such as "enzyme activity" or "DNA binding."
- **Cellular Component**: Indicates where gene products are located within the cell, like "nucleus" or "mitochondrion."

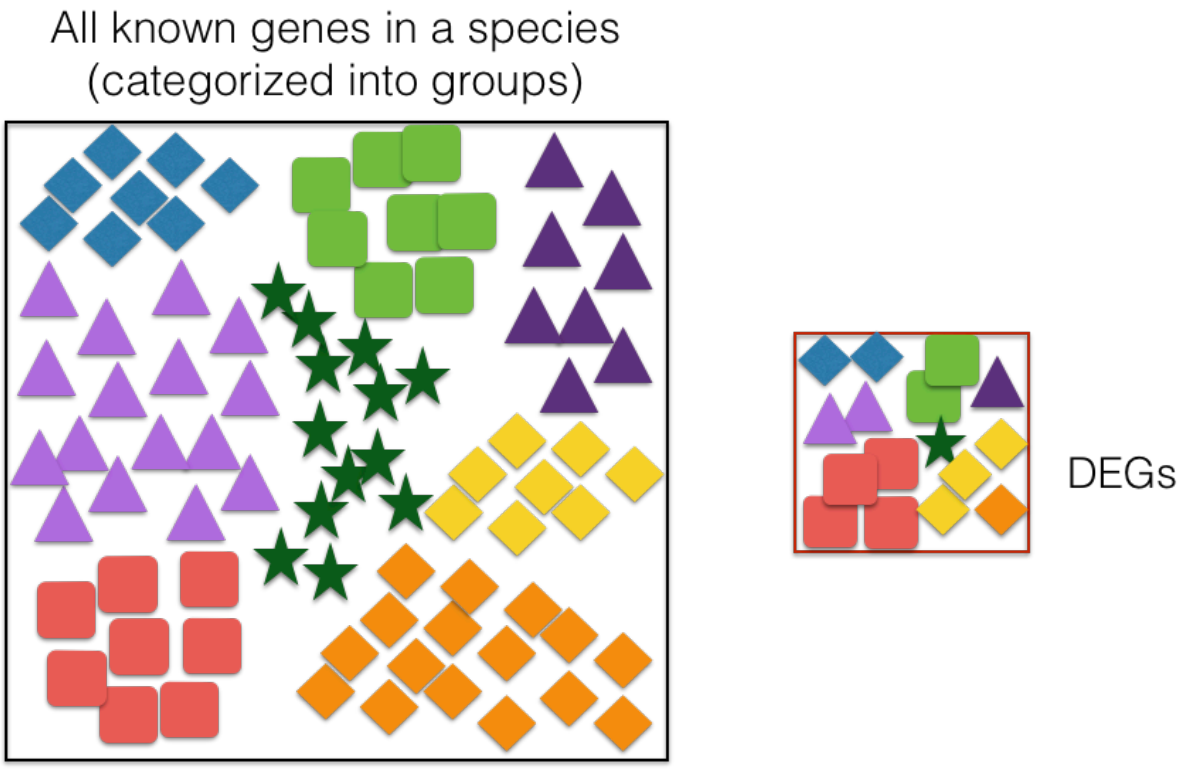
### 8.0.3 GO Enrichment Methods




GO enrichment methods help interpret differentially expressed gene lists from RNA-Seq data and include the following:

1. **Overrepresentation Analysis (ORA)**:

- ORA assesses whether specific GO terms are over-represented in a gene list compared to the background set of genes (e.g., all genes in the genome). Over-represented GO terms indicate biological functions or processes that are significantly associated with the genes of interest.




| Genes categories | Organism-specific Background | DE results | Over-represented? |
|---|---|---|---|
| Functional category 1 | 35/13000 | 25/1000 | Likely |
| Functional category 2 | 56/13000 | 4/1000 | Unlikely |
| Functional category 3 | 90/13000 | 8/1000 | Unlikely |
| Functional category 4 | 15/13000 | 10/1000 | Likely |
| … | | | |
| … | | | |

- **Hypergeometric Test**: The hypergeometric test in GO enrichment analysis is used to determine whether a set of genes is enriched for specific GO terms compared to the background set of genes. The test calculates the probability of observing the overlap between the query gene set and the gene set associated with a particular GO term, given the total number of genes in the dataset and the number of genes in the background set.

The formula for the cumulative probability of observing **at most** $q$ successes (the lower-tail probability) in a sample of size

$k$ from a population with $m$ successes and $N - m$ failures is:

$$P(X \leq q) = \sum_{i=0}^{q} \frac{\binom{m}{i}\binom{N-m}{k-i}}{\binom{N}{k}}$$

Where:

`q`: The **number of genes in your list of interest that are annotated with a specific GO term** (e.g., genes in your differential expression results that are associated with "cell cycle regulation").

`m`: The **total number of genes in the background set (e.g., all genes in the genome) that are annotated with the specific GO term** (e.g., all genes associated with "cell cycle regulation" in the entire dataset or genome).

`N`: The **total number of genes in the background set.**

`k`: The **total number of genes in your gene list of interest**

### 8.0.4 Example

Suppose:your background population has $N = 20,000$ genes (e.g., the entire genome). Out of these, $m = 500$ genes are associated with a specific GO term (e.g., "cell cycle regulation"). You have a list of $k = 1000$ genes of interest. In this list, $q = 30$ genes are associated with the "cell cycle regulation" GO term.

```
# Define parameters
N <- 20000          # Total genes in the background (population)
m <- 500            # Total genes associated with the GO term in the background
k <- 1000           # Sample size (genes in your list)
q <- 30             # Observed number of successes (overlap with GO term)

# Perform the hypergeometric test (upper-tail probability)
p_value <- phyper(q - 1, m, N - m, k, lower.tail = FALSE)
p_value
```

```
## [1] 0.1736549
```

2. **Gene Set Enrichment Analysis (GSEA)**: Instead of only considering genes above a certain threshold (e.g., differentially expressed genes), GSEA looks at the **entire ranked list of genes** (e.g., ranked by their expression levels) and checks whether genes associated with a certain GO term are *statistically over-represented at the top or bottom of a ranked gene list, indicating coordinated changes in gene expression.*

Here's how GSEA works:

- **Gene Ranking**: Genes are ranked based on a metric that reflects their differential expression, such as log2FoldChange.
- **Enrichment Score (ES)**: GSEA calculates an **Enrichment Score (ES)** for each gene set by walking down the ranked list, increasing the score when a gene belongs to the set and decreasing it otherwise. A positive ES suggests upregulation, while a negative ES suggests downregulation of the gene set.
- **Example:** The GSEA enrichment plot provides a graphical view of the enrichment score for a gene set:

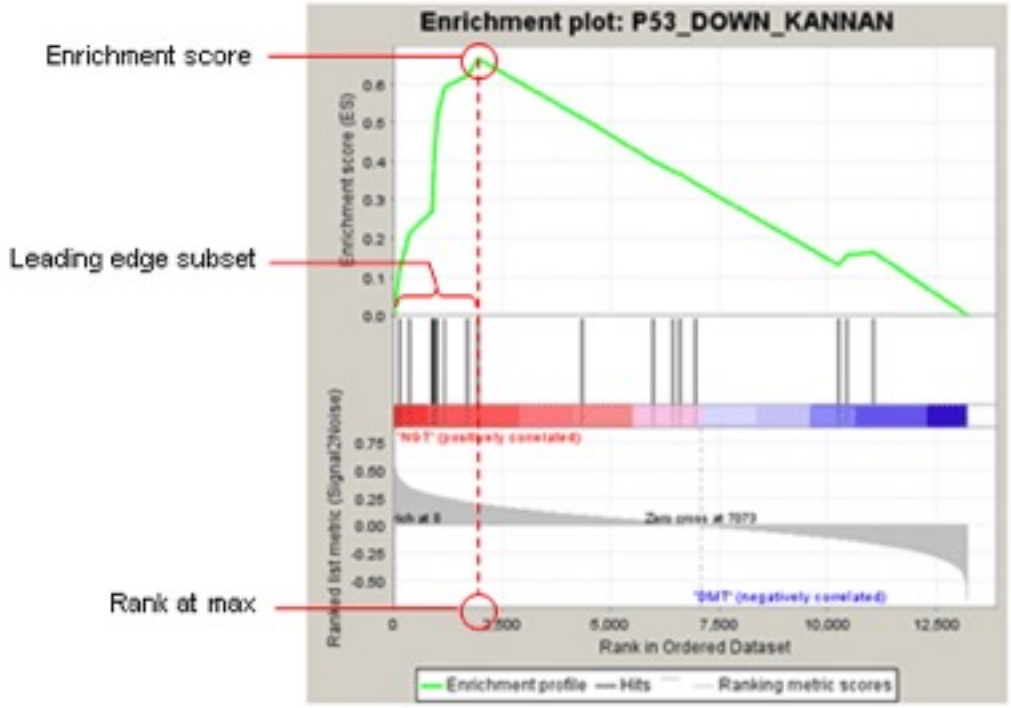



*Fig 1: Enrichment plot: P53_DOWN_KANNAN*
*Profile of the Running ES Score & Positions of GeneSet Members on the Rank Ordered List* - The top portion of the plot shows the running ES for the gene set as the analysis walks down the ranked list. The score at the peak of the plot (the score furthest from 0.0) is the ES for the gene set.

- The middle portion of the plot shows where the members of the gene set appear in the ranked list of genes.
- The leading edge subset of a gene set is the subset of members that contribute most to the ES.
- The bottom portion of the plot shows the value of the ranking metric (such as log2FoldChange) as you move down the list of ranked genes.

### 8.0.5 Over-representation Analysis (ORA) using `clusterProfiler`

```
# Uncomment the following if you haven't yet installed the required packages
# BiocManager::install(c("ggraph", "igraph", "visNetwork",
# "GO.db", "GOSemSim"), force = TRUE)

# IMPORTANT!!! Uncomment the following to update the GOenrichment package
# remove.package("GOenrichment")
# devtools::install_github("gurinina/GOenrichment", force = TRUE)

## Load required libraries
library(org.Hs.eg.db)#
library(clusterProfiler)#
library(tidyverse)
library(enrichplot)#
library(fgsea) #
```

```
library(igraph)
library(ggraph)
library(visNetwork)
library(GO.db)
library(GOSemSim)
library(GOenrichment)
library(tidyr)
```

#### 8.0.5.1 Running clusterProfiler: we first need to load the results of the differential expression analysis.

```
res_tableOE = readRDS("data/res_tableOE.RDS")

res_tableOE_tb <- res_tableOE %>%
  data.frame() %>% rownames_to_column(var = "gene") %>%
  dplyr::filter(!is.na(log2FoldChange))  %>% as_tibble()
```

To perform the over-representation analysis, we need a list of background genes and a list of significant genes:

```
## background set of genes
allOE_genes <- res_tableOE_tb$gene


sigOE = dplyr::filter(res_tableOE_tb, padj < 0.05)

## significant genes
sigOE_genes = sigOE$gene
```

The`enrichGO()` function performs the ORA for the significant genes of interest (`sigOE_genes`) compared to the background gene list (`allOE_genes`) and returns the enriched GO terms and their associated p-values.

```
## Run GO enrichment analysis
ego <- enrichGO(gene = sigOE_genes,
 universe = allOE_genes,
 keyType = "SYMBOL",
 OrgDb = org.Hs.eg.db,
 minGSSize = 20,
 maxGSSize = 300,
 ont = "BP",
 pAdjustMethod = "BH",
 qvalueCutoff = 0.05,
 readable = TRUE)
```

We can extract the gene lists associated with each enriched GO term from the `ego` object to create edges between terms based on shared genes.

```
# Extract the result as a data frame
enrich_df <- as.data.frame(ego)

names(enrich_df)
```

```
## [1] "ID"          "Description" "GeneRatio"   "BgRatio"     "pvalue"
## [6] "p.adjust"    "qvalue"      "geneID"      "Count"
```

```
head(enrich_df)
```

```
##                    ID                          Description GeneRatio   BgRatio
## GO:0006403 GO:0006403                    RNA localization  110/5343 188/14702
## GO:0042254 GO:0042254                  ribosome biogenesis  153/5343 282/14702
## GO:1903311 GO:1903311 regulation of mRNA metabolic process  149/5343 274/14702
## GO:0043484 GO:0043484           regulation of RNA splicing  102/5343 172/14702
## GO:0016571 GO:0016571                 histone methylation   86/5343 140/14702
## GO:0051168 GO:0051168                      nuclear export   90/5343 150/14702
##                    pvalue     p.adjust       qvalue
## GO:0006403 4.241779e-10 6.338239e-07 5.335773e-07
## GO:0042254 4.727052e-10 6.338239e-07 5.335773e-07
## GO:1903311 6.423898e-10 6.338239e-07 5.335773e-07
## GO:0043484 6.426604e-10 6.338239e-07 5.335773e-07
## GO:0016571 1.235279e-09 9.746355e-07 8.204856e-07
## GO:0051168 2.835770e-09 1.864519e-06 1.569623e-06
##
## GO:0006403
## GO:0042254              AATF/ABCE1/ABT1/BMS1/BRIX1/BYSL/C1QBP/CHD7/CUL4A/CUL4B/DDX18/DDX21/DDX27/DDX28/DDX4
## GO:1903311 AHCYL1/AKT1/ALKBH5/ANGEL2/APEX1/BTG2/C1QBP/CARHSP1/CASC3/CDK9/CELF1/CELF2/CELF3/CELF4/CIRBP/CLNS
## GO:0043484
## GO:0016571
## GO:0051168
##            Count
## GO:0006403   110
## GO:0042254   153
## GO:1903311   149
## GO:0043484   102
## GO:0016571    86
## GO:0051168    90
```

```
# GO term IDs
term_ids <- enrich_df$ID

# Split gene lists by "/" and assign term IDs as names
gene_lists <- strsplit(enrich_df$geneID, "/")

# Assign term IDs as names
names(gene_lists) <- term_ids
```

Using the gene lists associated with each GO term, we can create edges between terms based on the shared genes.

```
# Load GO data for semantic similarity calculation
go_data <- GOSemSim::godata(OrgDb = "org.Hs.eg.db",
                            ont = "BP",
                            keytype = "SYMBOL",
                            computeIC = TRUE)
```

```
## preparing gene to GO mapping data...
```

```
## preparing IC data...
```

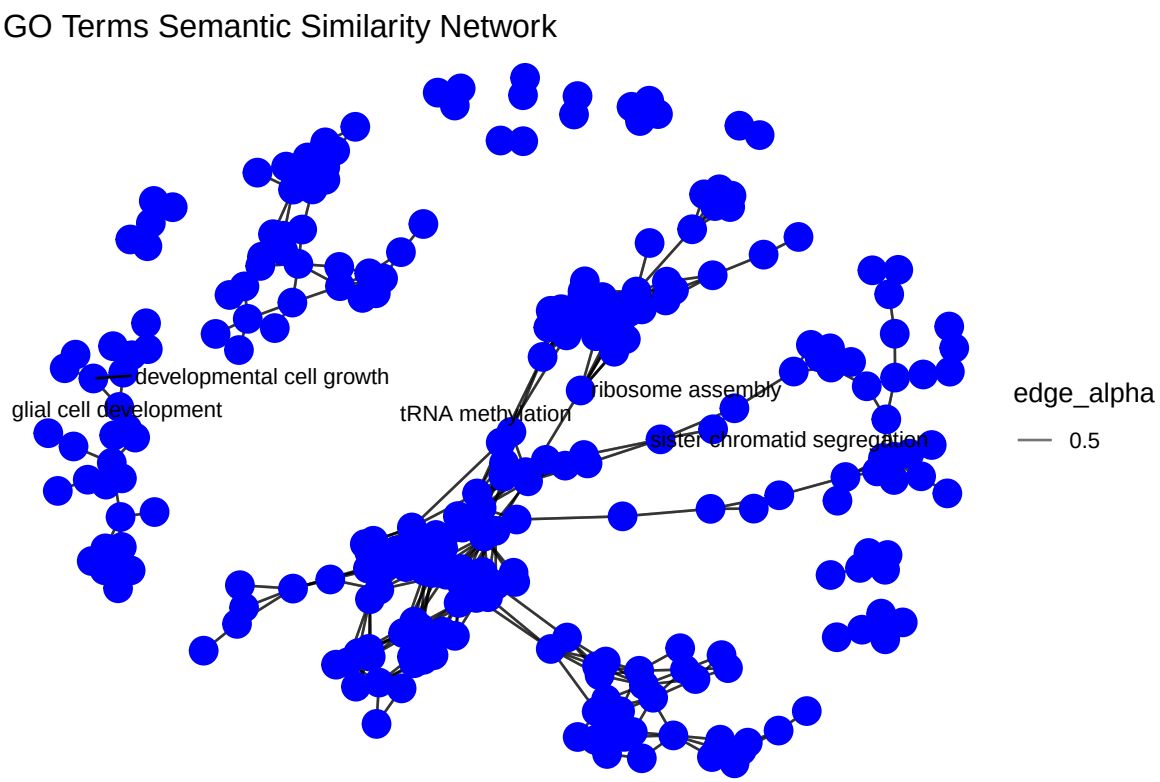
```
# Compute semantic similarity
similarity_matrix <- GOSemSim::mgoSim(term_ids, term_ids,
                      semData = go_data,
                      measure = "Wang", combine = NULL)

# Convert similarity matrix to edges
edges <- data.frame(from = rep(term_ids,
                    each = length(term_ids)),
                    to = rep(term_ids,
                    times = length(term_ids)),
                    similarity = as.vector(similarity_matrix))

# Filter out edges with zero similarity or self-loops
edges <- subset(edges, similarity > 0 & from != to)

edges <- edges %>%
  dplyr::filter(similarity >= 0.5)


# Convert term IDs to names
edges$to <- Term(edges$to)
edges$from <- Term(edges$from)

head(edges)
```

```
##                                  from
## 1                     RNA localization
## 2                  ribosome biogenesis
## 3                  ribosome biogenesis
## 4 regulation of mRNA metabolic process
## 5 regulation of mRNA metabolic process
## 6 regulation of mRNA metabolic process
##                                           to similarity
## 1            establishment of RNA localization      0.728
## 2           ribosomal small subunit biogenesis      0.848
## 3                           ribosome assembly      0.623
## 4                  regulation of RNA splicing      0.752
## 5                regulation of mRNA processing      0.863
## 6 regulation of mRNA splicing, via spliceosome      0.720
```

Now we can create a network plot to visualize the relationships between the enriched GO terms based on their semantic similarity.

```
# Create an igraph object from the edges data frame
g <- graph_from_data_frame(edges, directed = FALSE)

# Visualize the graph using ggraph
ggraph(g, layout = 'fr') +  # Fruchterman-Reingold layout
  geom_edge_link(aes(edge_alpha = 0.5)) +  # Edges with some transparency
  geom_node_point(size = 5, color = "blue") +  # Nodes sized uniformly
```

```
  geom_node_text(aes(label = name), repel = TRUE, size = 3) +  # Node labels
  theme_void() +  # Clean theme without axes
  labs(title = "GO Terms Semantic Similarity Network")  # Title for the plot
```



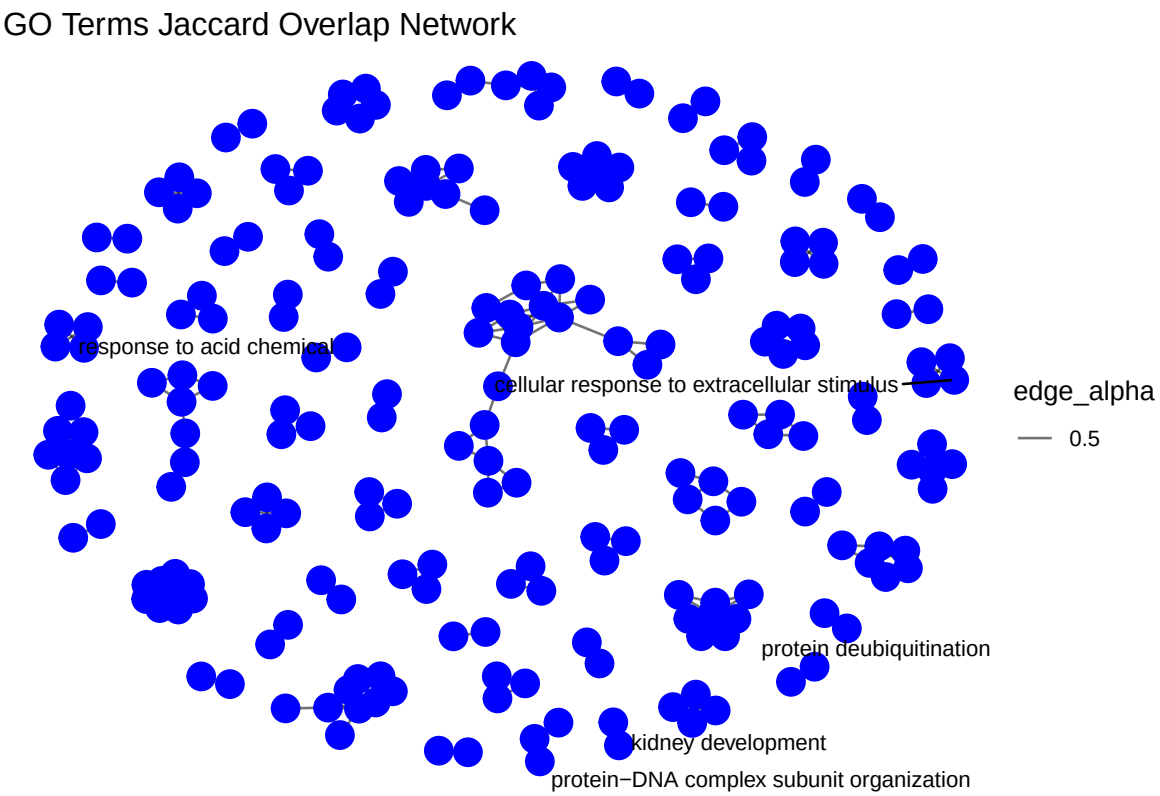

Edges based can also be defined by Jaccard similarity; defined as the size of the intersection each pair of GO terms divided by the size of their union

```
# Define a function to compute Jaccard similarity
jaccard_index <- function(genes1, genes2) {
length(intersect(genes1, genes2)) / length(union(genes1, genes2))
}

# Create an empty data frame to store edges
edges <- data.frame(from = character(), to = character(),
                    similarity = numeric())
  # Compute Jaccard similarity for each pair of terms
  for (i in 1:(length(term_ids) - 1)) {
  for (j in (i + 1):length(term_ids)) {
  similarity <- jaccard_index(gene_lists[[i]], gene_lists[[j]])
  if (similarity > 0) {  # Only keep edges with some overlap
  edges <- rbind(edges, data.frame(from = term_ids[i],
                                   to = term_ids[j],
                                   similarity = similarity))
  }}}

edges <- edges %>%
  dplyr::filter(similarity >= 0.5)
```

```
# Convert term IDs to names
edges$to <- Term(edges$to)
edges$from <- Term(edges$from)
```

Now we can create a network plot to visualize the relationships between the enriched GO terms based on the Jaccard similarity of their gene lists.

```
# Create a graph from the edges
g <- graph_from_data_frame(edges, directed = FALSE)

# Visualize the graph using ggraph
ggraph(g, layout = 'fr') +  # Fruchterman-Reingold layout
  geom_edge_link(aes(edge_alpha = 0.5)) +  # Edges with some transparency
  geom_node_point(size = 5,
                  color = "blue") +
  geom_node_text(aes(label = name),
                 repel = TRUE,
                 size = 3) +  # Node labels
  theme_void() +  # Clean theme without axes
  labs(title = "GO Terms Jaccard Overlap Network")
```

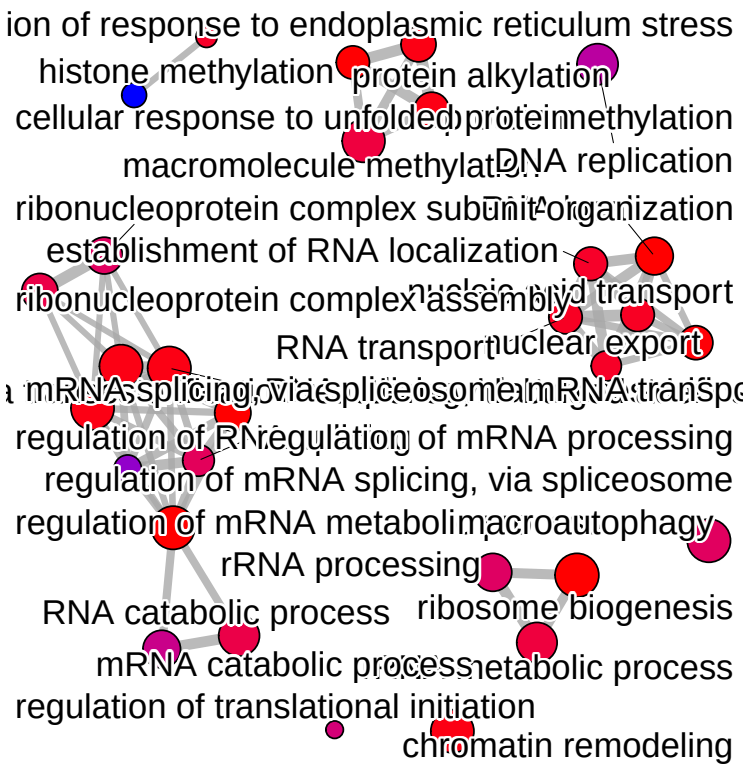



The resulting network plots shows the relationships between the enriched GO terms based on either Semantic or Jaccard similarity of their gene lists.

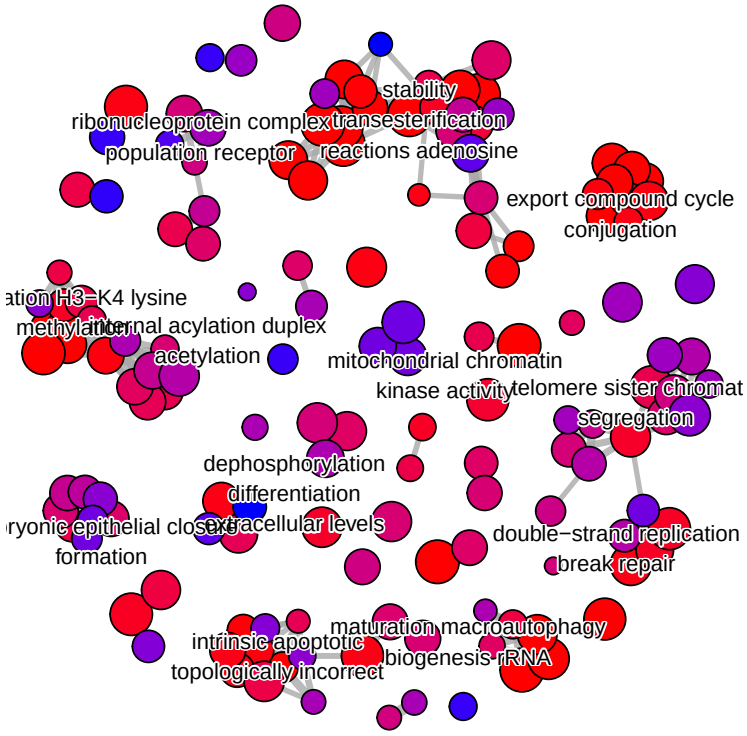
### 8.0.6 Visualizing Enrichment Results

**Network Plot**: The `emapplot()` function generates a network plot where each node represents an enriched gene set, and edges between nodes indicate the similarity or overlap between those gene sets.

```
pwt <- pairwise_termsim(
  ego,
  method = "JC",
  semData = NULL
)

# category labels
emapplot(pwt, showCategory = 30,
         node_label = "category",
         cex_label_group = 3,
         layout.params = list("mds")) +
         theme(legend.position = "none")
```



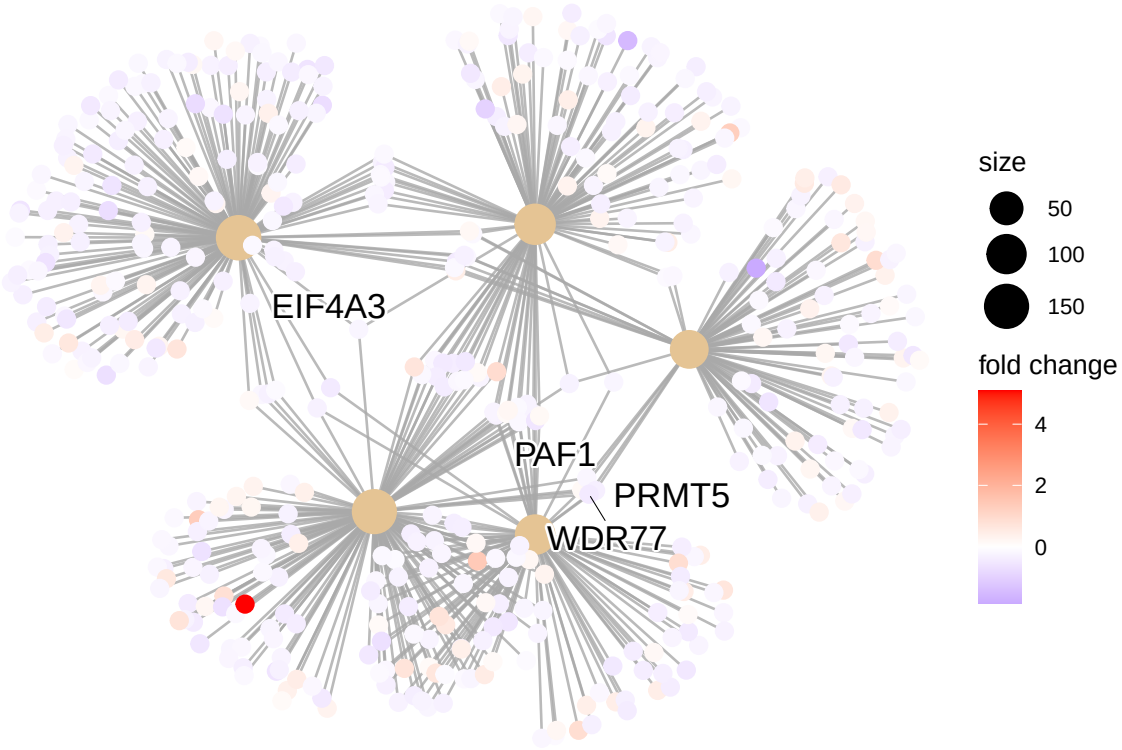

```
# group labels
emapplot(pwt, showCategory = 150,
         node_label = "group") +
         theme(legend.position = "none")
```




To save any of these figures if they become too cluttered, click on the Export button in the RStudio Plots tab and Save as PDF. In the pop-up window, change the PDF size to 24 x 32 to give a figure of appropriate size for the text labels.

**cnetplot**: Visualizes connections between genes and top GO terms, highlighting shared pathways.This plot is particularly useful for hypothesis generation in identifying genes that may be important to several of the most affected processes.

```
## Extract the foldchanges
OE_foldchanges <- sigOE$log2FoldChange

names(OE_foldchanges) <- sigOE$gene

## cnetplot for the top 5 categories
cnetplot(ego,
  categorySize="padj",
  node_label = "all",
  showCategory = 5,
  foldChange = OE_foldchanges,
  vertex.label.font = 6)
```




### 8.0.7 GSEA Using clusterProfiler and fgsea

Prepare fold changes, sort by expression, and run GSEA with `gseGO()` for clusterProfiler or `fgseaSimple()` for `fgsea`.

```
## Extract the foldchanges
foldchanges <- res_tableOE_tb$log2FoldChange

## Name each fold change with the gene name
names(foldchanges) <- res_tableOE_tb$gene

## Sort fold changes in decreasing order
foldchanges <- sort(foldchanges, decreasing = TRUE)
```

We can explore the enrichment of BP Gene Ontology terms using gene set enrichment analysis (GSEA) using gene sets associated with BP Gene Ontology terms

```
gseaGO <- clusterProfiler::gseGO(
  geneList = foldchanges,
  ont = "BP",
  keyType = "SYMBOL",
  eps = 0,
  minGSSize = 20,
  maxGSSize = 300,
  pAdjustMethod = "BH",
  pvalueCutoff = 0.05,
  verbose = TRUE,
```

```
  OrgDb = "org.Hs.eg.db",
  by = "fgsea"
)
```

```
## preparing geneSet collections...
```

```
## GSEA analysis...
```

```
## leading edge analysis...
```

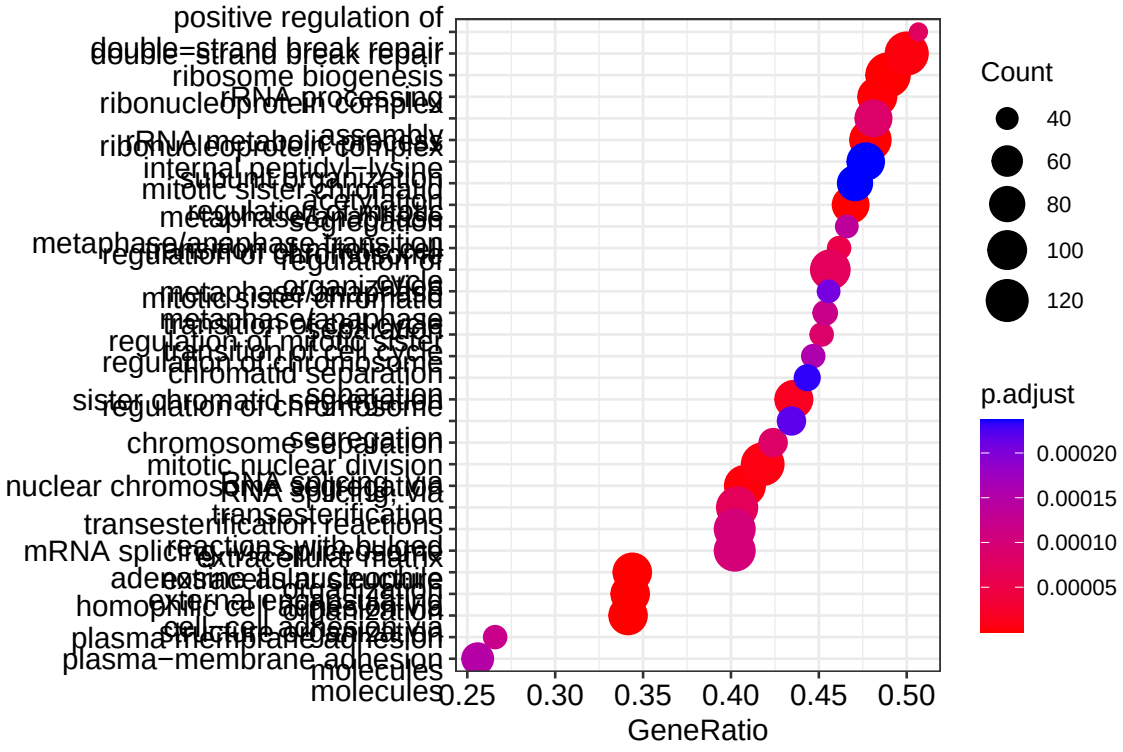
```
## done...
```

```
gseaGO_results <- data.frame(gseaGO)
```

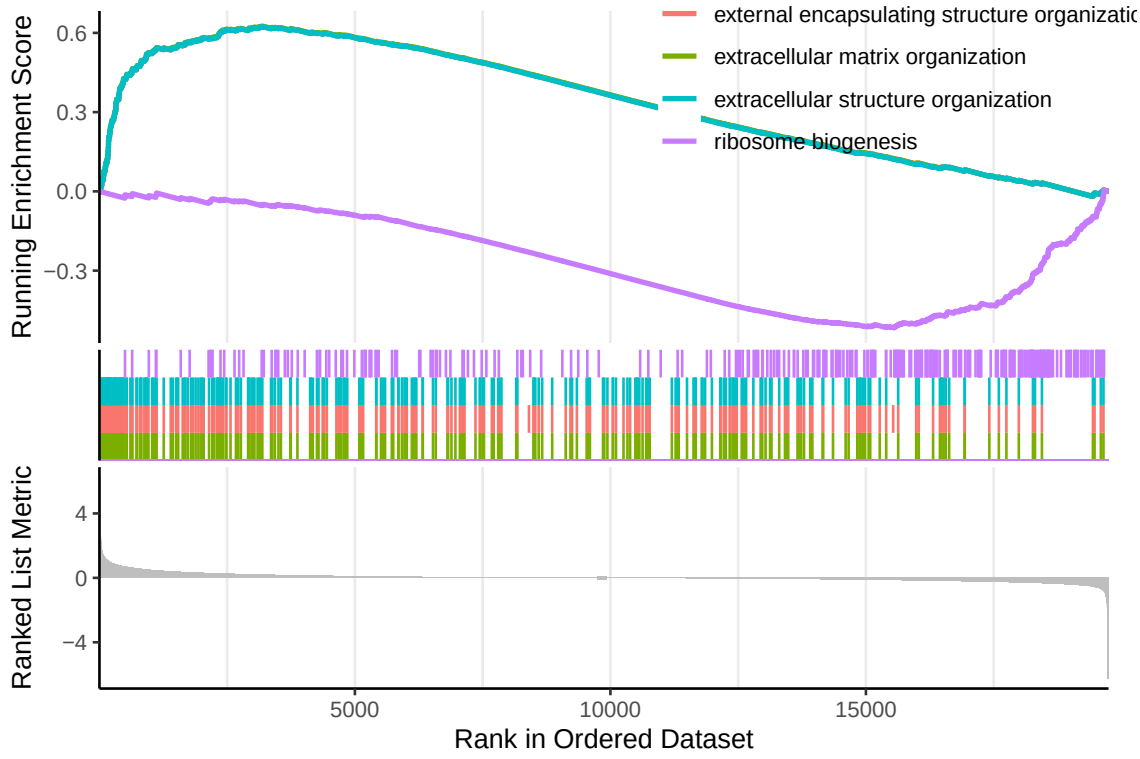
### 8.0.8 Visualizing GSEA Results

Visualize GSEA results with `dotplot()` for significant categories and `gseaplot2()` for gene set ranks.

```
dotplot(gseaGO, showCategory = 30)
```

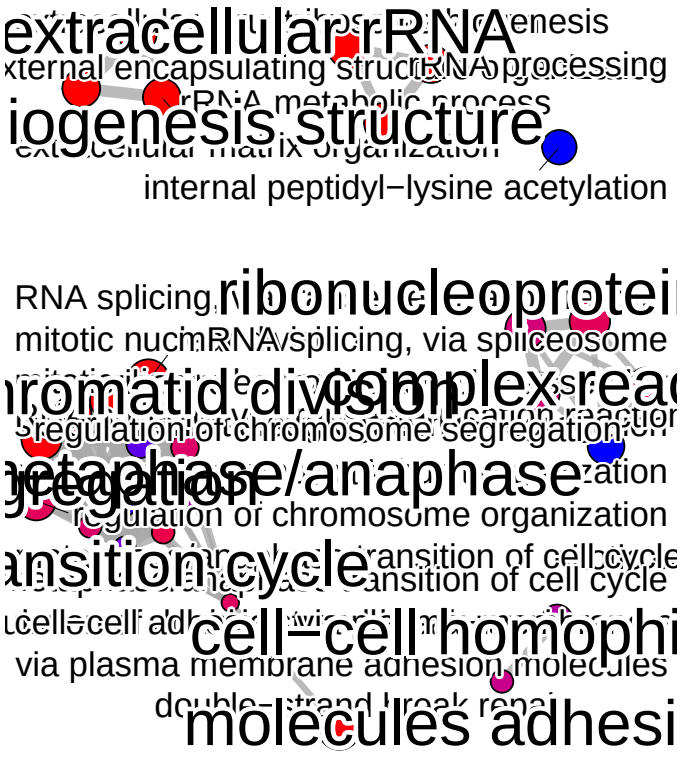


```
gseaplot2(gseaGO, geneSetID = 1:4)
```




The emapplot clusters the 50 most significant (by padj) GO terms to visualize relationships between terms.

```
# find the overlap between the top 50 terms
pwt <- pairwise_termsim(
  gseaGO,
  method = "JC",
  semData = NULL
  )

emapplot(pwt, showCategory = 30, node_label = "all",
         cex_label_group = 3,
         layout.params = list("mds")) +
         theme(legend.position = "none")
```




### 8.0.9 GSEA using fgsea

GO annotations are updated regularly, and the GO terms associated with genes can change over time. This can lead to inconsistencies in the results of GO enrichment analyses when using different versions of the GO annotations. To address this issue, the `GOenrichment` package provides `hGOBP.gmt` a recently downloaded version of `BP` GO annotations. This file is a list of GO terms and their associated genes, which is used as input for the enrichment analysis.

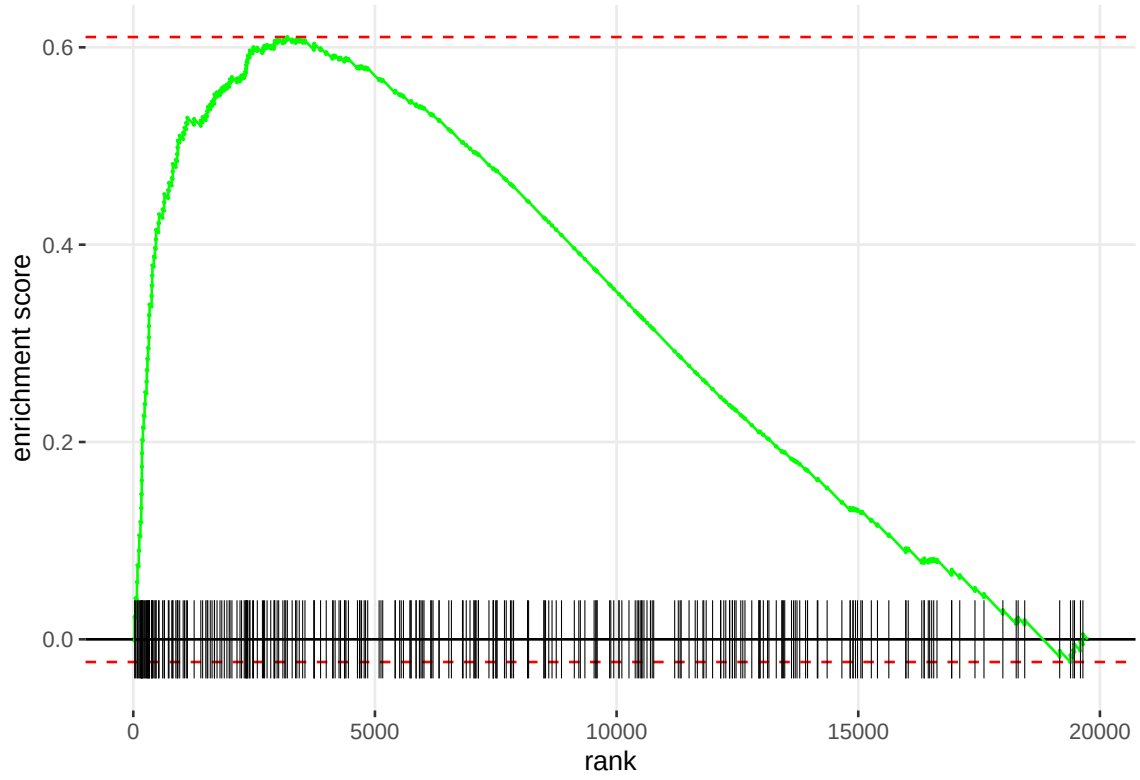
To run GSEA using the `fgsea` package, we need to load the gene sets from the `hGOBP.gmt` file and the gene-level statistics from the `foldchanges` vector. We can then run the GSEA analysis using the `fgseaSimple` function.

```
# Let's look at the hGOBP.gmt file
hGOBP.gmt[1]
```

```
## $`MITOCHONDRIAL GENOME MAINTENANCE`
##  [1] "AKT3"     "CHCHD4"   "DNA2"     "DNAJA3"   "ENDOG"    "FLCN"     "LIG3"
##  [8] "LONP1"    "MEF2A"    "METTL4"   "MGME1"    "MPV17"    "OPA1"     "PARP1"
## [15] "PIF1"     "POLG"     "POLG2"    "PPARGC1A" "PRIMPOL"  "RRM1"     "RRM2B"
## [22] "SESN2"    "SLC25A33" "SLC25A36" "SLC25A4"  "SSBP1"    "STOX1"    "TOP3A"
## [29] "TP53"     "TWNK"     "TYMP"
```

```
fgseaRes <-  fgsea::fgseaSimple(pathways = hGOBP.gmt,
                                stats = foldchanges,
                                nperm = 1000, maxSize = 300,
                                minSize = 20)
```

```
fgsea <- data.frame(fgseaRes,stringsAsFactors = F)

w = which(fgsea$ES > 0)

fposgsea <- fgsea[w,]

fposgsea <- fposgsea %>% arrange(padj)

plotEnrichment(hGOBP.gmt[["EXTRACELLULAR MATRIX ORGANIZATION"]], foldchanges)
```




The `fgseaSimple` function returns a list of enriched gene sets and the enrichment score (ES).

We are going to compare these results to running the GO enrichment function `runGORESP`.

Let's first load the `GOenrichment` package and check the available functions:

```
# Uncomment the following if you haven't yet installed GOenrichment.

# devtools::install_github("gurinina/GOenrichment")

library(GOenrichment)

ls("package:GOenrichment")
```

```
## [1] "compSCORE"  "hGOBP.gmt"  "hyperG"     "runGORESP"  "runNetwork" "visSetup"
```

### 8.0.10 `GOenrichment` Package Analysis

`runGORESP` uses over-representation analysis to identify enriched GO terms and returns two `data.frames`; of enriched GO terms (nodes) and GO term relationships (edges).

`compSCORE`evaluates a matrix of fitness scores to identify the `querySet` with scores above a specified significance threshold.

We'll use a significance cutoff of 0.58, corresponding to a 1.5x change in expression.

```
args(runGORESP)
```

```
## function (scoreMat, curr_exp = "test", fdrThresh = 0.2, bp_path = NULL,
##     bp_input = NULL, go_path = NULL, go_input = NULL, minSetSize = 5,
##     maxSetSize = 300)
## NULL
```

```
# ?runGORESP

# Define the query set
matx <- cbind(foldchanges,foldchanges)

scoreMat = compSCORE(matx, coln = 1, sig =  0.58)

head(scoreMat)
```

```
##          index    score     gene
## HSPA6        1 6.246527    HSPA6
## MOV10        1 5.083301    MOV10
## ASCL1        1 4.441095    ASCL1
## HSPA7        1 3.637043    HSPA7
## SCRT1        1 2.925522    SCRT1
## SIGLEC14     1 2.614563 SIGLEC14
```

```
hresp = runGORESP(fdrThresh = 0.2, scoreMat = scoreMat,
  bp_input = hGOBP.gmt, go_input = NULL, minSetSize = 20,
  maxSetSize = 300)

names(hresp$edgeMat)
```

```
## [1] "source"       "target"       "overlapCoeff" "width"        "label"
```

```
names(hresp$enrichInfo)
```

```
##  [1] "filename"           "term"               "nGenes"
##  [4] "nQuery"             "nOverlap"           "querySetFraction"
##  [7] "geneSetFraction"    "foldEnrichment"     "P"
## [10] "FDR"                "overlapGenes"       "maxOverlapGeneScore"
## [13] "cluster"            "id"                 "size"
## [16] "formattedLabel"
```

```
head(hresp$enrichInfo[,c(2,3,4,5,10)])
```

```
##                                          term nGenes nQuery nOverlap      FDR
## 1             EXTRACELLULAR MATRIX ORGANIZATION    293    722       39 5.13e-09
## 2          EXTRACELLULAR STRUCTURE ORGANIZATION    294    724       39 5.13e-09
## 3 EXTERNAL ENCAPSULATING STRUCTURE ORGANIZATION    294    724       39 5.13e-09
## 4                              PROTEIN REFOLDING     27    727        8 3.71e-03
## 5         REGULATION OF VASCULATURE DEVELOPMENT    273    739       25 2.19e-02
## 6            CELL ADHESION MEDIATED BY INTEGRIN     82    704       12 2.42e-02
```

Let's check the overlap between the enriched terms found using `runGORESP` and those found using `fgseaSimple` as they used the same GO term libraries:

```
w = which(fposgsea$padj <= 0.2)

lens <- length(intersect(fposgsea$pathway[w],hresp$enrichInfo$term))

length(w)
```

```
## [1] 620
```

```
dim(hresp$enrichInfo)
```

```
## [1] 57 16
```

```
percent_overlap <- lens/nrow(hresp$enrichInfo)*100

percent_overlap
```

```
## [1] 82.45614
```

80%, that's very good because we are using two different GO enrichment methods, over-representation analysis and GSEA.

We can visualize the results of the GO enrichment analysis using the `visNetwork` package. This package allows us to create interactive network visualizations.

The equivalent function in the `GOenrichment` package is `runNetwork`:

We use the`visSetup` function to prepare the data for network visualization. We then run the `runNetwork` function to generate the interactive network plot.

Interactive visualization available in the HTML version.

This network analysis is based on Cytoscape, an open source bioinformatics software platform for visualizing molecular interaction networks. out of all the GO packages.

### 8.0.11 Other tools and resources

- GeneMANIA. GeneMANIA finds other genes that are related to a set of input genes, using a very large set of functional association data curated from the literature.

- ReviGO. Revigo is an online GO enrichment tool that allows you to copy-paste your significant gene list and your background gene list. The output is a visualization of enriched GO terms in a hierarchical tree.
- AmiGO. AmiGO is the current official web-based set of tools for searching and browsing the Gene Ontology database.
- etc.