Class 1*

Recap: Transcriptomics and the analysis of RNA-Seq data

Barry Grant

2023-04-03

1. Background

The data for this hands-on session comes from a published RNA-seq experiment where airway smooth muscle cells were treated with dexamethasone, a synthetic glucocorticoid steroid with anti-inflammatory effects (Himes et al. 2014).

Glucocorticoids are used, for example, by people with asthma to reduce inflammation of the airways (Figure 1). The anti-inflammatory effects on airway smooth muscle (ASM) cells has been known for some time but the underlying molecular mechanisms are unclear.

Himes et al. used RNA-seq to profile gene expression changes in four different ASM cell lines treated with dexamethasone glucocorticoid.

They found a number of differentially expressed genes but focus much of the discussion on a gene called CRISPLD2.

This gene encodes a secreted protein known to be involved in lung development, and SNPs in this gene in previous GWAS studies are associated with inhaled corticosteroid resistance and bronchodilator response in asthma patients.

Outline

In this class session we will:

- Open a new *RStudio Project* and *Quarto document* for today's class;
- Review how to install both Bioconductor and CRAN packages;
- Explore the Himes *et al.* gene expression data using base R, dplyr and ggplot2 package functions;

*http://thegrantlab.org/teaching/

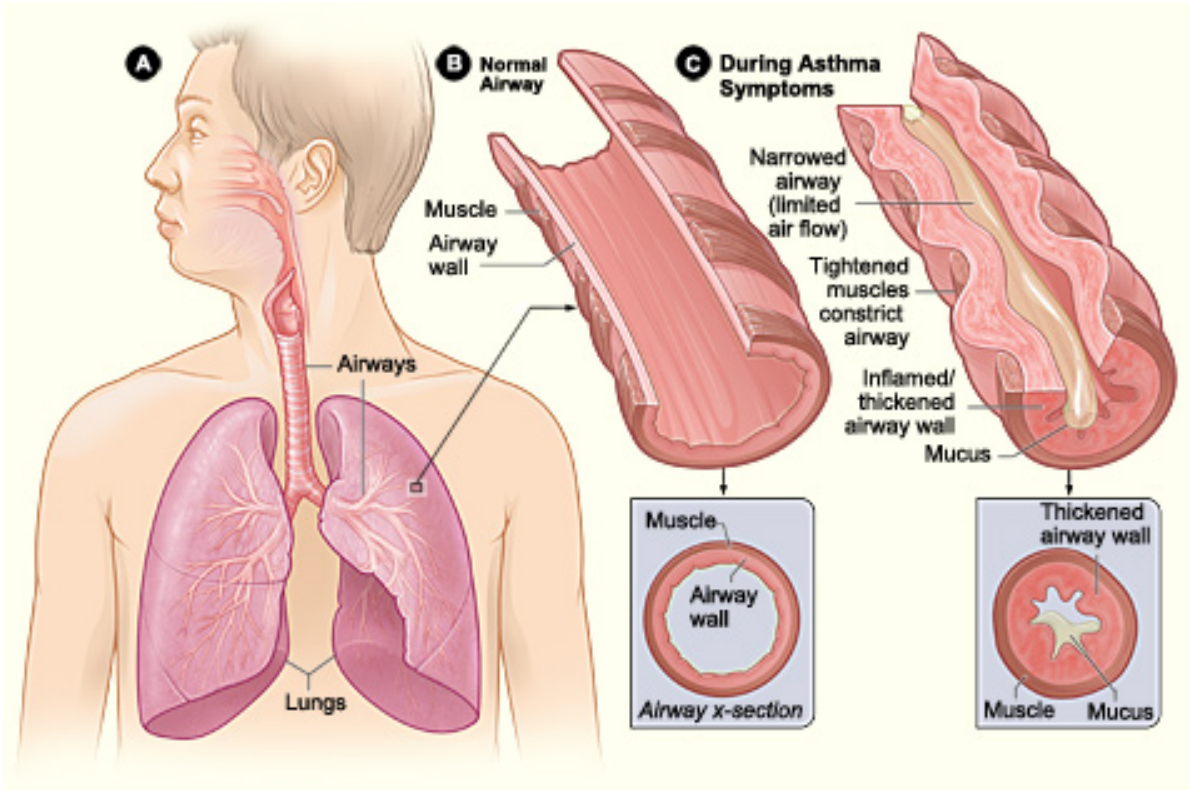


Figure 1: Illustration of normal and asthma airways

- Perform a detailed differential gene expression analysis with the DESeq2 package.
- Render a reproducible *PDF report* of your work with answers to all questions below.

For full details of the original analysis see the PubMed entry 24926665 and for associated data see the GEO entry GSE52778.

2. Bioconductor setup

As we already noted back in BGGN213 that Bioconductor is a large repository and resource for R packages that focus on analysis of high-throughput genomic data.

Recall that Bioconductor packages are installed differently than "regular" R packages from CRAN. To install the core Bioconductor packages, copy and paste the following two lines of code into your **R console** one at a time.

> **N.B.** Remember not to put these install commands in your Quarto report as it will at best re-install the packages every time you render your report, which is not what you want, and at worst cause a confusing rendering error.

```
install.packages("BiocManager")
BiocManager::install()
```

If this finished without yielding obvious error messages we can install the **DESeq2** bioconductor package that we will use in this class:

```
# For this class, you'll also need DESeq2:
BiocManager::install("DESeq2")
```

> More about the install process (click to expand)
>
> The entire install process can take some time as there are many packages with dependencies on other packages.
> For some important notes on the install process please see our Bioconductor setup notes.
> Your install process may produce some notes or other output.
> Generally, as long as you don't get an error message, you're good to move on. If you do see error messages then again please see our Bioconductor setup notes for debugging steps.

Finally, check that you have installed everything correctly by entering the following two commands at the console window:

```
library(BiocManager)
library(DESeq2)
```

If you get a message that says something like: `Error in library(DESeq2) : there is no package called 'DESeq2'`, then the required packages did not install correctly. Please see our Bioconductor setup notes and let us know so we can debug this together.

Side-note: Aligning reads to a reference genome

The computational analysis of an RNA-seq experiment begins from the FASTQ files that contain the nucleotide sequence of each read and a quality score at each position. These reads must first be aligned to a reference genome or transcriptome. The output of this alignment step is commonly stored in a file format called SAM/BAM. This is the workflow we followed last day.
Once the reads have been aligned, there are a number of tools that can be used to count the number of reads/fragments that can be assigned to genomic features for each sample. These often take as input SAM/BAM alignment files and a file specifying the genomic features, e.g. a GFF3 or GTF file specifying the gene models as obtained from ENSEMBLE or UCSC.
In the workflow we'll use here, the abundance of each transcript was quantified using **kallisto** (software, paper) and transcript-level abundance estimates were then summarized to the gene level to produce length-scaled counts using the R package **txImport** (software, paper), suitable for using in count-based analysis tools like DESeq. This is the starting point - a "count matrix", where each cell indicates the number of reads mapping to a particular gene (in rows) for each sample (in columns). This is where we left off last day when analyzing our 1000 genome data.

> **Note**: This is one of several well-established workflows for data pre-processing. The goal here is to provide a reference point to acquire fundamental skills with DESeq2 that will be applicable to other bioinformatics tools and workflows. In this regard, the following resources summarize a number of best practices for RNA-seq data analysis and pre-processing.
>
> 1. Conesa, A. et al. "A survey of best practices for RNA-seq data analysis." *Genome Biology* 17:13 (2016).
> 2. Soneson, C., Love, M. I. & Robinson, M. D. "Differential analyses for RNA-seq: transcript-level estimates improve gene-level inferences." *F1000Res.* 4:1521 (2016).
> 3. Griffith, Malachi, et al. "Informatics for RNA sequencing: a web resource for analysis on the cloud." *PLoS Comput Biol* 11.8: e1004393 (2015).

DESeq2 Required Inputs

As input, the DESeq2 package expects (**1**) a data.frame of **count data** (as obtained from RNA-seq or another high-throughput sequencing experiment) and (**2**) a second data.frame with information about the samples - often called sample metadata (or `colData` in DESeq2-speak because it supplies metadata/information about the columns of the countData matrix) (Figure 2).

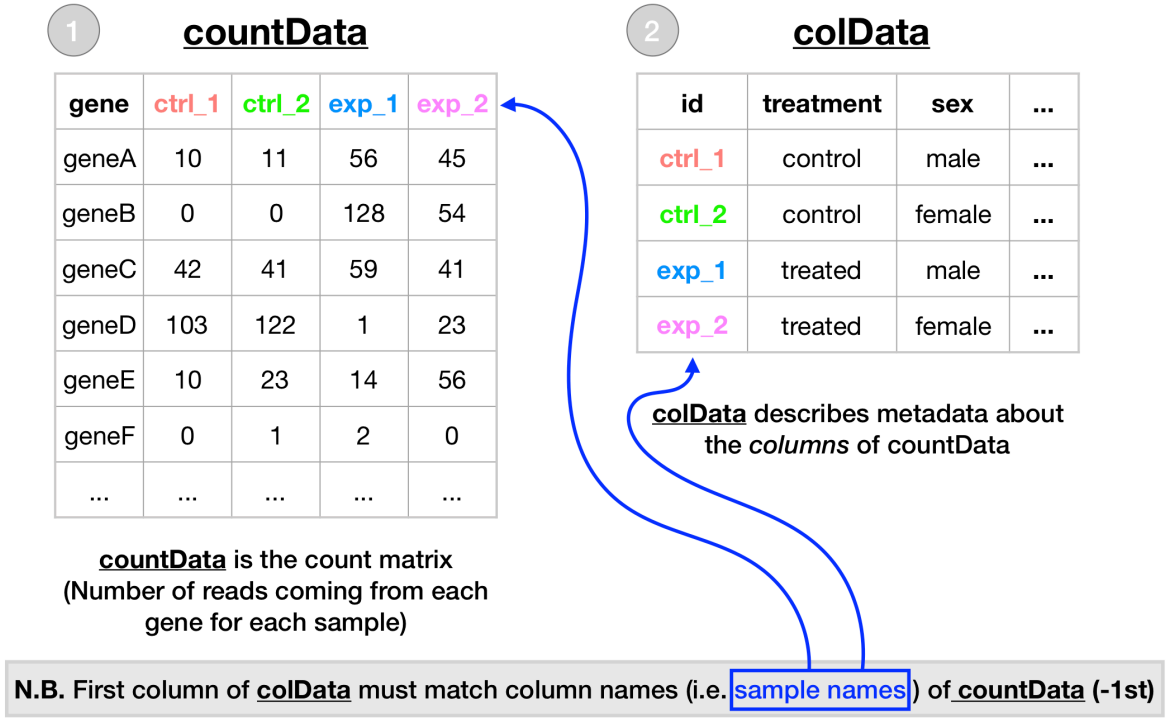


Figure 2: DESeq requires both countData and colData in specific formats.

The **count matrix** (called the `countData` in DESeq2-speak) the value in the *i-th* row and the *j-th* column of the data.frame tells us how many reads can be assigned to *gene i* in *sample j*. Analogously, for other types of assays, the rows of this matrix might correspond e.g. to binding regions (with ChIP-Seq) or peptide sequences (with quantitative mass spectrometry).

For the sample **metadata** (i.e. `colData` in DESeq2-speak) samples are in rows and metadata about those samples are in columns. Notice that the first column of colData must match the column names of countData (except the first, which is the gene ID column) (Figure 2).

> **Note from the DESeq2 vignette:** The values in the input contData object should be counts of sequencing reads/fragments. This is important for DESeq2's statistical model to hold, as only counts allow assessing the measurement precision

correctly. It is important to never provide counts that were pre-normalized for sequencing depth/library size, as the statistical model is most powerful when applied to un-normalized counts, and is designed to account for library size differences internally.

3. Import countData and colData

If you have not already done so first create a new **RStudio project** (File > New Project > New Directory > New Project) and download the input airway_scaledcounts.csv and airway_metadata.csv into your project directory.

Open a new **Quarto Document** (File > New File > Quarto Document) add a code chunk and use the **read.csv()** function to read these count data and metadata files.

```
# Complete the missing code
counts <- read.csv("___", row.names=1)
metadata <-  ___("airway_metadata.csv")
```

Now, take a look at the head of each.

```
head(counts)
```

```
                SRR1039508 SRR1039509 SRR1039512 SRR1039513 SRR1039516
ENSG00000000003        723        486        904        445       1170
ENSG00000000005          0          0          0          0          0
ENSG00000000419        467        523        616        371        582
ENSG00000000457        347        258        364        237        318
ENSG00000000460         96         81         73         66        118
ENSG00000000938          0          0          1          0          2
                SRR1039517 SRR1039520 SRR1039521
ENSG00000000003       1097        806        604
ENSG00000000005          0          0          0
ENSG00000000419        781        417        509
ENSG00000000457        447        330        324
ENSG00000000460         94        102         74
ENSG00000000938          0          0          0
```

```
head(metadata)
```

```
          id     dex celltype     geo_id
1 SRR1039508 control   N61311 GSM1275862
2 SRR1039509 treated   N61311 GSM1275863
3 SRR1039512 control  N052611 GSM1275866
4 SRR1039513 treated  N052611 GSM1275867
5 SRR1039516 control  N080611 GSM1275870
6 SRR1039517 treated  N080611 GSM1275871
```

You can also use the **View()** function to view the entire object. Notice something here. The sample IDs in the metadata sheet (SRR1039508, SRR1039509, etc.) exactly match the column names of the countdata, except for the rownames, which contains the Ensembl gene ID. This is important, and we'll get more strict about it later on.

- **Q1.** How many genes are in this dataset?
- **Q2.** How many 'control' cell lines do we have?

Hint (click to expand)

The functions `dim()`, `nrow()` and `View()` may be useful for answering these questions.

4. Toy differential gene expression

Lets perform some exploratory differential gene expression analysis. **Note: this analysis is for demonstration only. NEVER do differential expression analysis this way!**

Look at the metadata object again to see which samples are `control` and which are drug `treated`. You can also see this in the metadata printed table below:

Note that the control samples are SRR1039508, SRR1039512, SRR1039516, and SRR1039520. This bit of code will first find the sample `id` for those labeled control. Then calculate the mean counts per gene across these samples:

```
control <- metadata[metadata[,"dex"]=="control",]
control.counts <- counts[ ,control$id]
control.mean <- rowSums( control.counts )/4
head(control.mean)
```

```
ENSG00000000003 ENSG00000000005 ENSG00000000419 ENSG00000000457 ENSG00000000460
         900.75            0.00          520.50          339.75           97.25
```

Metadata for the Himes *et al.* RNASeq experiment
Note the '**dex**' column specifies drug presence/absence

id	dex	celltype	geo_id
SRR1039508	control	N61311	GSM1275862
SRR1039509	treated	N61311	GSM1275863
SRR1039512	control	N052611	GSM1275866
SRR1039513	treated	N052611	GSM1275867
SRR1039516	control	N080611	GSM1275870
SRR1039517	treated	N080611	GSM1275871
SRR1039520	control	N061011	GSM1275874
SRR1039521	treated	N061011	GSM1275875

```
ENSG00000000938
           0.75
```

Side-note: An alternative way to do this same thing using the `dplyr` package from the tidyverse is shown below. Which do you prefer and why?

```
library(dplyr)
control <- metadata %>% filter(dex=="control")
control.counts <- counts %>% select(control$id)
control.mean <- rowSums(control.counts)/4
head(control.mean)
```

```
ENSG00000000003 ENSG00000000005 ENSG00000000419 ENSG00000000457 ENSG00000000460
         900.75            0.00          520.50          339.75           97.25
ENSG00000000938
           0.75
```

- **Q3.** How would you make the above code in either approach more robust? Is there a function that could help here?
- **Q4.** Follow the same procedure for the `treated` samples (i.e. calculate the mean per gene across drug treated samples and assign to a labeled vector called `treated.mean`)

Hint (click to expand)

For Q3 consider what would happen if you were to add more samples. Would the values obtained with the exact code above be correct?
For Q4 you can adapt the above code being sure to substitute `"treated"` for `"control"`. For example:

```
treated <- metadata[metadata[,"dex"]=="___",]
treated.mean <- ___( counts[ ,treated$id] )
```

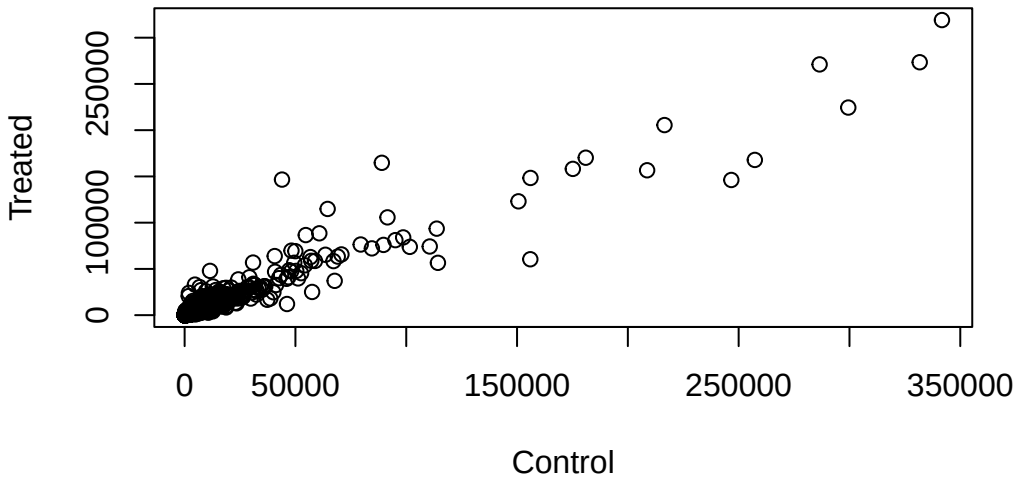
We will combine our meancount data for bookkeeping purposes.

```
meancounts <- data.frame(control.mean, treated.mean)
```

Directly comparing the raw counts is going to be problematic if we just happened to sequence one group at a higher depth than another. Later on we'll do this analysis properly, normalizing by sequencing depth per sample using a better approach. But for now, **colSums()** the data to show the sum of the mean counts across all genes for each group. Your answer should look like this:

```
control.mean treated.mean
    23005324     22196524
```

- **Q5 (a).** Create a scatter plot showing the mean of the treated samples against the mean of the control samples. Your plot should look something like the following.

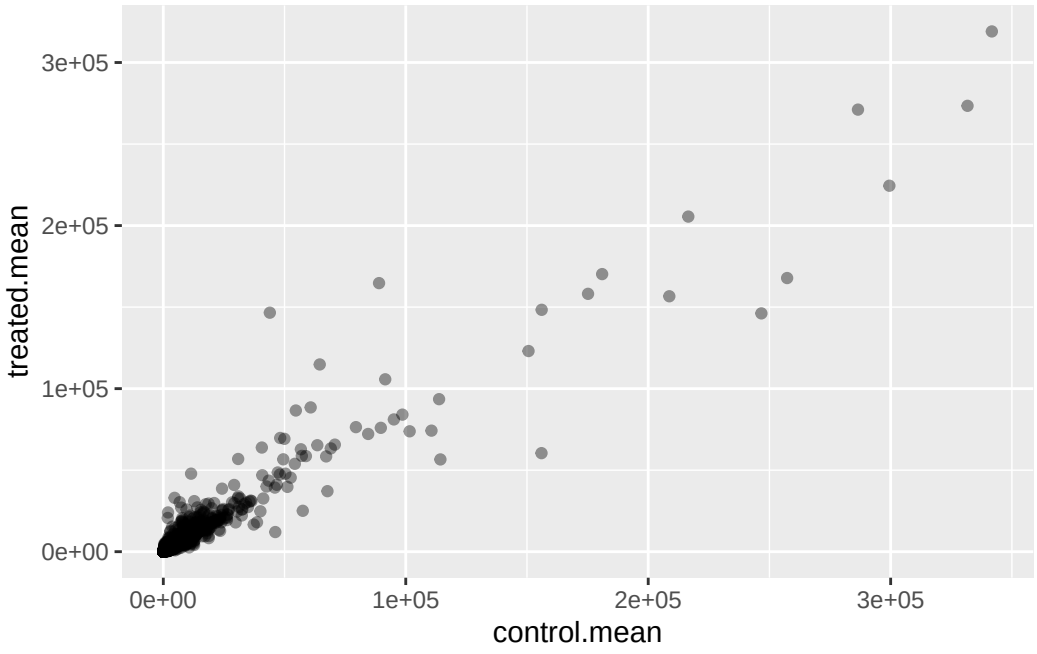


Hint (click to expand)

Here you can call **plot()** with `x=meancounts[,1]` and `y=meancounts[,2]`. Or just call `plot(meancounts)`

```
plot(meancounts[,1],meancounts[,2], xlab="Control", ylab="Treated")
```

- **Q5 (b).**You could also use the **ggplot2** package to make this figure producing the plot below. What **geom_?()** function would you use for this plot?

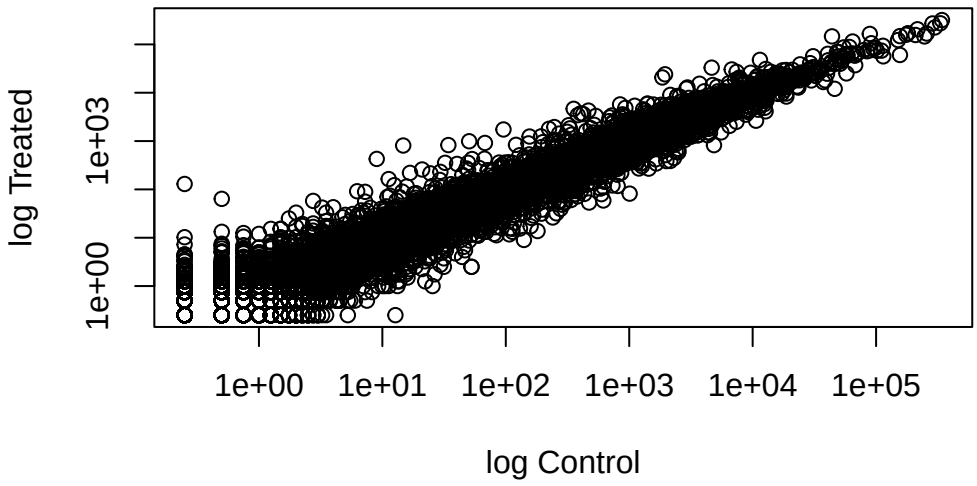


Wait a sec. There are 60,000-some rows in this data, but I'm only seeing a few dozen dots at most outside of the big clump around the origin.

- **Q6.** Try plotting both axes on a log scale. What is the argument to **plot()** that allows you to do this?

Hint (click to expand)

See the help for **?plot.default** to see how to set log axis.



If you are using **ggplot** have a look at the function `scale_x_continuous(trans="log2")` and of course do the same for the y axis.

We can find candidate differentially expressed genes by looking for genes with a large change between control and dex-treated samples. We usually look at the log2 of the fold change, because this has better mathematical properties.

Here we calculate log2foldchange, add it to our `meancounts` data.frame and inspect the results either with the **head()** or the **View()** function for example.

```
meancounts$log2fc <- log2(meancounts[,"treated.mean"]/meancounts[,"control.mean"])
head(meancounts)
```

```
                control.mean treated.mean      log2fc
ENSG00000000003       900.75       658.00 -0.45303916
ENSG00000000005         0.00         0.00         NaN
ENSG00000000419       520.50       546.00  0.06900279
ENSG00000000457       339.75       316.50 -0.10226805
ENSG00000000460        97.25        78.75 -0.30441833
ENSG00000000938         0.75         0.00        -Inf
```

There are a couple of "weird" results. Namely, the NaN ("not a number") and -Inf (negative infinity) results.

The NaN is returned when you divide by zero and try to take the log. The -Inf is returned when you try to take the log of zero. It turns out that there are a lot of genes with zero expression. Let's filter our data to remove these genes. Again inspect your result (and the intermediate steps) to see if things make sense to you

```
zero.vals <- which(meancounts[,1:2]==0, arr.ind=TRUE)

to.rm <- unique(zero.vals[,1])
mycounts <- meancounts[-to.rm,]
head(mycounts)
```

```
                control.mean treated.mean      log2fc
ENSG00000000003       900.75       658.00 -0.45303916
ENSG00000000419       520.50       546.00  0.06900279
ENSG00000000457       339.75       316.50 -0.10226805
ENSG00000000460        97.25        78.75 -0.30441833
ENSG00000000971      5219.00      6687.50  0.35769358
ENSG00000001036      2327.00      1785.75 -0.38194109
```

- **Q7.** What is the purpose of the `arr.ind` argument in the **which()** function call above? Why would we then take the first column of the output and need to call the **unique()** function?

Hint (click to expand)

The `arr.ind=TRUE` argument will clause **which()** to return both the row and column indices (i.e. positions) where there are TRUE values. In this case this will tell us which genes (rows) and samples (columns) have zero counts. We are going to ignore any genes that have zero counts in any sample so we just focus on the `row` answer. Calling **unique()** will ensure we don't count any row twice if it has zero entries in both samples. Ask Barry to discuss and demo this further;-)

A common threshold used for calling something differentially expressed is a log2(FoldChange) of greater than 2 or less than -2. Let's filter the dataset both ways to see how many genes are up or down-regulated.

```
up.ind <- mycounts$log2fc > 2
down.ind <- mycounts$log2fc < (-2)
```

- **Q8.** Using the `up.ind` vector above can you determine how many up regulated genes we have at the greater than 2 fc level?
- **Q9.** Using the `down.ind` vector above can you determine how many down regulated genes we have at the greater than 2 fc level?
- **Q10.** Do you trust these results? Why or why not?

Hint (click to expand)

What type of vectors are `up.ind` and `down.ind`? How could you count the number of TRUE elements? Your answers should look like:

```
[1] "Up: 250"
```

```
[1] "Down: 367"
```

For question 10, all our analysis has been done based on fold change. However, fold change can be large (e.g. »two-fold up- or down-regulation) without being statistically significant (e.g. based on p-values). We have not done anything yet to determine whether the differences we are seeing are significant. These results in their current form are likely to be very misleading. In the next section we will begin to do this properly with the help of the **DESeq2** package.

In total, you should have over 600 differentially expressed genes, in either direction.

5. Setting up for DESeq

Let's do this the right way. DESeq2 is an R package specifically for analyzing count-based NGS data like RNA-seq. It is available from Bioconductor. Bioconductor is a project to provide tools for analyzing high-throughput genomic data including RNA-seq, ChIP-seq and arrays. You can explore Bioconductor packages here.

Bioconductor packages usually have great documentation in the form of *vignettes*. For a great example, take a look at the DESeq2 vignette for analyzing count data. This 40+ page manual is packed full of examples on using DESeq2, importing data, fitting models, creating visualizations, references, etc.

Just like R packages from CRAN, you only need to install Bioconductor packages once (instructions here), then load them every time you start a new R session.

```
library(DESeq2)
citation("DESeq2")
```

Take a second and read through all the stuff that flies by the screen when you load the DESeq2 package. When you first installed DESeq2 it may have taken a while, because DESeq2 *depends* on a number of other R packages (S4Vectors, BiocGenerics, parallel, IRanges, etc.) Each of these, in turn, may depend on other packages. These are all loaded into your working environment when you load DESeq2. Also notice the lines that start with `The following objects are masked from 'package:....`

Importing data

Bioconductor software packages often define and use custom class objects for storing data. This helps to ensure that all the needed data for analysis (and the results) are available. DESeq works on a particular type of object called a *DESeqDataSet*. The DESeqDataSet is a single object that contains input values, intermediate calculations like how things are normalized, and all results of a differential expression analysis.

You can construct a DESeqDataSet from (1) a count matrix, (2) a metadata file, and (3) a formula indicating the design of the experiment.

We have talked about (1) and (2) previously. The third needed item that has to be specified at the beginning of the analysis is a *design formula*. This tells DESeq2 which columns in the sample information table (`colData`) specify the experimental design (i.e. which groups the samples belong to) and how these factors should be used in the analysis. Essentially, this *formula* expresses how the counts for each gene depend on the variables in colData.

Take a look at `metadata` again. The thing we're interested in is the `dex` column, which tells us which samples are treated with dexamethasone versus which samples are untreated controls. We'll specify the design with a tilde, like this: `design=~dex`. (The tilde is the shifted key to the left of the number 1 key on my keyboard. It looks like a little squiggly line).

We will use the **DESeqDataSetFromMatrix()** function to build the required *DESeqDataSet* object and call it `dds`, short for our DESeqDataSet. If you get a warning about "some variables in design formula are characters, converting to factors" don't worry about it. Take a look at the `dds` object once you create it.

```
dds <- DESeqDataSetFromMatrix(countData=counts,
                              colData=metadata,
                              design=~dex)
dds
```

```
class: DESeqDataSet
dim: 38694 8
metadata(1): version
assays(1): counts
rownames(38694): ENSG00000000003 ENSG00000000005 ... ENSG00000283120
  ENSG00000283123
rowData names(0):
colnames(8): SRR1039508 SRR1039509 ... SRR1039520 SRR1039521
colData names(4): id dex celltype geo_id
```

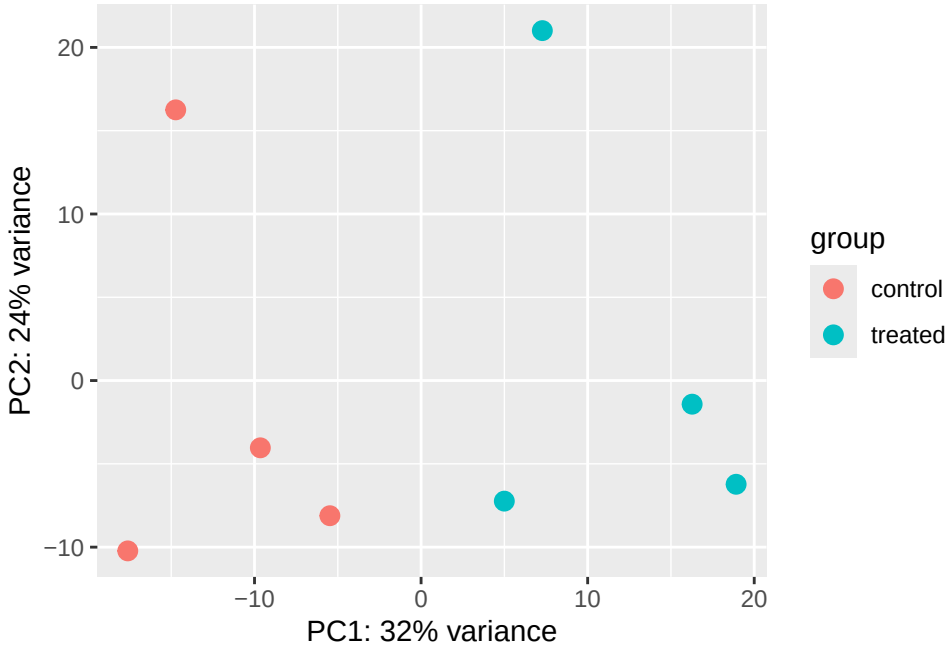
6. Principal Component Analysis (PCA)

Before running DESeq analysis we can look how the count data samples are related to one another via our old friend Principal Component Analysis (PCA). We will follow the DESeq recommended procedure and associated functions for PCA. First calling `vst()` to apply a variance stabilizing transformation (read more about this in the expandable section below) and then `plotPCA()` to calculate our PCs and plot the results.

> Variance stabilizing transformation (click to expand)
>
> The purpose of the variance stabilizing transformation in DESeq2 is to normalize count data and stabilize the variance across the mean expression level. This is important because count data often have different variances at different levels of expression, and this can lead to false positive or false negative results in downstream analysis. The variance stabilizing transformation is a type of power transformation that aims to stabilize the variance across the mean expression level of genes.
> The `vst()` function in DESeq2 transforms raw count data into a log2-counts per million (logCPM) space, where the variance is approximately independent of the mean. This transformation improves the performance of statistical tests and reduces the impact of outliers on the analysis. The variance-stabilized data can then be used for exploratory data analysis, visualization, and downstream statistical analysis, such as differential expression analysis, clustering, and dimension reduction. For motivated readers this vignette has additional details.

```
vsd <- vst(dds, blind = FALSE)
plotPCA(vsd, intgroup = c("dex"))
```

```
using ntop=500 top features by variance
```



The `plotPCA()` function comes with DESeq2 and `intgroup` are our `dex` groups for labeling the samples; they tell the function to use them to choose colors.

We can also build the PCA plot from scratch using the **ggplot2 package**. This is done by asking the plotPCA function to return the data used for plotting rather than building the plot.

```
pcaData <- plotPCA(vsd, intgroup=c("dex"), returnData=TRUE)
```

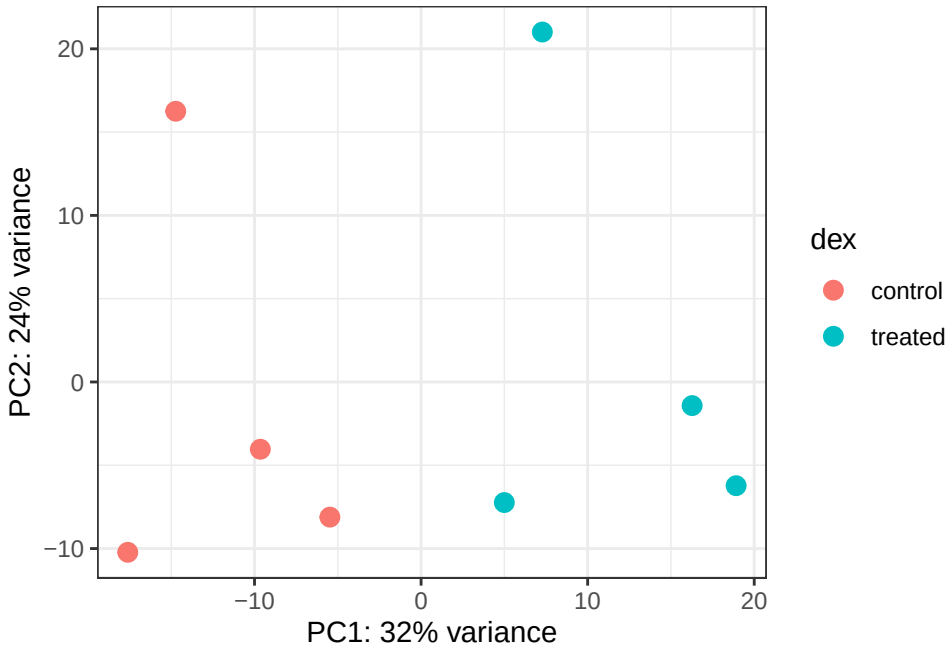
```
using ntop=500 top features by variance
```

```
head(pcaData)
```

```
                 PC1        PC2   group     dex       name
SRR1039508 -17.607922 -10.225252 control control SRR1039508
SRR1039509   4.996738  -7.238117 treated treated SRR1039509
SRR1039512  -5.474456  -8.113993 control control SRR1039512
SRR1039513  18.912974  -6.226041 treated treated SRR1039513
SRR1039516 -14.729173  16.252000 control control SRR1039516
SRR1039517   7.279863  21.008034 treated treated SRR1039517
```

```
# Calculate percent variance per PC for the plot axis labels
percentVar <- round(100 * attr(pcaData, "percentVar"))
```

```
ggplot(pcaData) +
  aes(x = PC1, y = PC2, color = dex) +
  geom_point(size =3) +
  xlab(paste0("PC1: ", percentVar[1], "% variance")) +
  ylab(paste0("PC2: ", percentVar[2], "% variance")) +
  coord_fixed() +
  theme_bw()
```



7. DESeq analysis

Finally, let's run the DESeq analysis pipeline on the dataset, and reassign the resulting object back to the same variable. Note that before we start, `dds` is a bare-bones DESeqDataSet. The `DESeq()` function takes a DESeqDataSet and returns a DESeqDataSet, but with additional information filled in (including the differential expression results we are after). Notice how if we try to access these results before running the analysis, nothing exists.

```
results(dds)
```

```
Error in results(dds): couldn't find results. you should first run DESeq()
```

Here, we're running the DESeq pipeline on the `dds` object, and reassigning the whole thing back to `dds`, which will now be a DESeqDataSet populated with all those values. Get some help on `?DESeq` (notice, no "2" on the end). This function calls a number of other functions within the package to essentially run the entire pipeline (normalizing by library size by estimating the "size factors," estimating dispersion for the negative binomial model, and fitting models and getting statistics for each gene for the design specified when you imported the data).

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

```
estimating size factors
```

```
estimating dispersions
```

```
gene-wise dispersion estimates
```

```
mean-dispersion relationship
```

```
final dispersion estimates
```

```
fitting model and testing
```

Getting results

Since we've got a fairly simple design (single factor, two groups, treated versus control), we can get results out of the object simply by calling the `results()` function on the DESeqDataSet that has been run through the pipeline. The help page for `?results` and the vignette both have extensive documentation about how to pull out the results for more complicated models (multi-factor experiments, specific contrasts, interaction terms, time courses, etc.).

```
res <- results(dds)
res
```

```
log2 fold change (MLE): dex treated vs control
Wald test p-value: dex treated vs control
DataFrame with 38694 rows and 6 columns
                 baseMean log2FoldChange     lfcSE      stat    pvalue
                <numeric>      <numeric> <numeric> <numeric> <numeric>
ENSG00000000003  747.1942     -0.3507030  0.168246 -2.084470 0.0371175
ENSG00000000005    0.0000             NA        NA        NA        NA
ENSG00000000419  520.1342      0.2061078  0.101059  2.039475 0.0414026
ENSG00000000457  322.6648      0.0245269  0.145145  0.168982 0.8658106
ENSG00000000460   87.6826     -0.1471420  0.257007 -0.572521 0.5669691
...                    ...            ...       ...       ...       ...
ENSG00000283115  0.000000             NA        NA        NA        NA
ENSG00000283116  0.000000             NA        NA        NA        NA
ENSG00000283119  0.000000             NA        NA        NA        NA
ENSG00000283120  0.974916      -0.668258   1.69456 -0.394354  0.693319
ENSG00000283123  0.000000             NA        NA        NA        NA
                     padj
                <numeric>
ENSG00000000003  0.163035
ENSG00000000005        NA
ENSG00000000419  0.176032
ENSG00000000457  0.961694
ENSG00000000460  0.815849
...                   ...
ENSG00000283115        NA
ENSG00000283116        NA
ENSG00000283119        NA
ENSG00000283120        NA
ENSG00000283123        NA
```

Convert the `res` object to a *data.frame* with the `as.data.frame()` function and then pass it to `View()` to bring it up in a data viewer.

Why do you think so many of the adjusted p-values are missing (`NA`)? Try looking at the `baseMean` column, which tells you the average overall expression of this gene, and how that relates to whether or not the p-value was missing. Go to the DESeq2 vignette and read the section about "Independent filtering and multiple testing."

> **Note**. The goal of independent filtering is to filter out those tests from the procedure that have no, or little chance of showing significant evidence, without even looking at the statistical result. Genes with very low counts are not likely to see significant differences typically due to high dispersion. This results in increased detection power at the same experiment-wide type I error [*i.e., better FDRs*].

We can summarize some basic tallies using the summary function.

```
summary(res)
```

```
out of 25258 with nonzero total read count
adjusted p-value < 0.1
LFC > 0 (up)       : 1563, 6.2%
LFC < 0 (down)     : 1188, 4.7%
outliers [1]       : 142, 0.56%
low counts [2]     : 9971, 39%
(mean count < 10)
[1] see 'cooksCutoff' argument of ?results
[2] see 'independentFiltering' argument of ?results
```

The results function contains a number of arguments to customize the results table. By default the argument `alpha` is set to 0.1. If the adjusted p value cutoff will be a value other than 0.1, alpha should be set to that value:

```
res05 <- results(dds, alpha=0.05)
summary(res05)
```

```
out of 25258 with nonzero total read count
adjusted p-value < 0.05
LFC > 0 (up)       : 1236, 4.9%
LFC < 0 (down)     : 933, 3.7%
outliers [1]       : 142, 0.56%
low counts [2]     : 9033, 36%
(mean count < 6)
[1] see 'cooksCutoff' argument of ?results
[2] see 'independentFiltering' argument of ?results
```

8. Adding annotation data

Our result table so far only contains the Ensembl gene IDs. However, alternative gene names and extra annotation are usually required for informative interpretation of our results. In this section we will add this necessary annotation data to our results.

We will use one of Bioconductor's main annotation packages to help with *mapping* between various ID schemes. Here we load the **AnnotationDbi** package and the annotation data package for humans **org.Hs.eg.db**.

```
library("AnnotationDbi")
library("org.Hs.eg.db")
```

> **Note**: You may have to install these with the `BiocManager::install("AnnotationDbi")` and `BiocManager::install("org.Hs.eg.db")` function calls.

The later of these is is the organism annotation package ("org") for Homo sapiens ("Hs"), organized as an AnnotationDbi database package ("db"), using Entrez Gene IDs ("eg") as primary key. To get a list of all available key types that we can use to map between, use the `columns()` function:

```
columns(org.Hs.eg.db)
```

```
 [1] "ACCNUM"       "ALIAS"        "ENSEMBL"      "ENSEMBLPROT"  "ENSEMBLTRANS"
 [6] "ENTREZID"     "ENZYME"       "EVIDENCE"     "EVIDENCEALL"  "GENENAME"
[11] "GENETYPE"     "GO"           "GOALL"        "IPI"          "MAP"
[16] "OMIM"         "ONTOLOGY"     "ONTOLOGYALL"  "PATH"         "PFAM"
[21] "PMID"         "PROSITE"      "REFSEQ"       "SYMBOL"       "UCSCKG"
[26] "UNIPROT"
```

Side-note: You can also pull up documentation (i.e. help) with a description of these different values with the regular `help()` command, e.g. `help("REFSEQ")`

The main function we will use from the **AnnotationDbi** package is called **mapIds()**.

We can use the **mapIds()** function to add individual columns to our results table. We provide the row names of our results table as a key, and specify that `keytype=ENSEMBL`. The `column` argument tells the mapIds() function which information we want, and the `multiVals` argument tells the function what to do if there are multiple possible values for a single input value. Here we ask to just give us back the first one that occurs in the database.

```
res$symbol <- mapIds(org.Hs.eg.db,
                     keys=row.names(res), # Our genenames
                     keytype="ENSEMBL",        # The format of our genenames
                     column="SYMBOL",          # The new format we want to add
                     multiVals="first")
```

```
head(res)
```

```
log2 fold change (MLE): dex treated vs control
Wald test p-value: dex treated vs control
DataFrame with 6 rows and 7 columns
                  baseMean log2FoldChange     lfcSE      stat    pvalue
                 <numeric>      <numeric> <numeric> <numeric> <numeric>
ENSG00000000003 747.194195     -0.3507030  0.168246 -2.084470 0.0371175
ENSG00000000005   0.000000             NA        NA        NA        NA
ENSG00000000419 520.134160      0.2061078  0.101059  2.039475 0.0414026
ENSG00000000457 322.664844      0.0245269  0.145145  0.168982 0.8658106
ENSG00000000460  87.682625     -0.1471420  0.257007 -0.572521 0.5669691
ENSG00000000938   0.319167     -1.7322890  3.493601 -0.495846 0.6200029
                     padj      symbol
                <numeric> <character>
ENSG00000000003  0.163035      TSPAN6
ENSG00000000005        NA        TNMD
ENSG00000000419  0.176032        DPM1
ENSG00000000457  0.961694       SCYL3
ENSG00000000460  0.815849       FIRRM
ENSG00000000938        NA         FGR
```

- **Q11.** Run the **mapIds()** function two more times to add the Entrez ID and UniProt accession and GENENAME as new columns called `res$entrez`, `res$uniprot` and `res$genename`.

Hint (click to expand)

Your code and results should look like the following:

```
res$entrez <- mapIds(org.Hs.eg.db,
                     keys=row.names(res),
                     column="ENTREZID",
                     keytype="ENSEMBL",
                     multiVals="first")

res$uniprot <- mapIds(org.Hs.eg.db,
                     keys=row.names(res),
                     column="UNIPROT",
                     keytype="ENSEMBL",
                     multiVals="first")

res$genename <- mapIds(org.Hs.eg.db,
                     keys=row.names(res),
                     column="GENENAME",
                     keytype="ENSEMBL",
                     multiVals="first")

head(res)
```

```
log2 fold change (MLE): dex treated vs control
Wald test p-value: dex treated vs control
DataFrame with 6 rows and 10 columns
                  baseMean log2FoldChange     lfcSE      stat    pvalue
                 <numeric>      <numeric> <numeric> <numeric> <numeric>
ENSG00000000003 747.194195     -0.3507030  0.168246 -2.084470 0.0371175
ENSG00000000005   0.000000             NA        NA        NA        NA
ENSG00000000419 520.134160      0.2061078  0.101059  2.039475 0.0414026
ENSG00000000457 322.664844      0.0245269  0.145145  0.168982 0.8658106
ENSG00000000460  87.682625     -0.1471420  0.257007 -0.572521 0.5669691
ENSG00000000938   0.319167     -1.7322890  3.493601 -0.495846 0.6200029
                     padj      symbol      entrez     uniprot
                <numeric> <character> <character> <character>
ENSG00000000003  0.163035      TSPAN6        7105  A0A087WYV6
ENSG00000000005        NA        TNMD       64102      Q9H2S6
ENSG00000000419  0.176032        DPM1        8813      H0Y368
ENSG00000000457  0.961694       SCYL3       57147      X6RHX1
ENSG00000000460  0.815849       FIRRM       55732      A6NFP1
ENSG00000000938        NA         FGR        2268      B7Z6W7
                          genename
                       <character>
ENSG00000000003          tetraspanin 6
ENSG00000000005            tenomodulin
ENSG00000000419 dolichyl-phosphate m..
ENSG00000000457 SCY1 like pseudokina..
ENSG00000000460 FIGNL1 interacting r..
ENSG00000000938 FGR proto-oncogene, ..
```

You can arrange and view the results by the adjusted p-value

```
ord <- order( res$padj )
#View(res[ord,])
head(res[ord,])
```

```
log2 fold change (MLE): dex treated vs control
Wald test p-value: dex treated vs control
DataFrame with 6 rows and 10 columns
                baseMean log2FoldChange     lfcSE      stat      pvalue
               <numeric>      <numeric> <numeric> <numeric>   <numeric>
ENSG00000152583   954.771        4.36836 0.2371268   18.4220 8.74490e-76
ENSG00000179094   743.253        2.86389 0.1755693   16.3120 8.10784e-60
ENSG00000116584  2277.913       -1.03470 0.0650984  -15.8944 6.92855e-57
ENSG00000189221  2383.754        3.34154 0.2124058   15.7319 9.14433e-56
ENSG00000120129  3440.704        2.96521 0.2036951   14.5571 5.26424e-48
ENSG00000148175 13493.920        1.42717 0.1003890   14.2164 7.25128e-46
                      padj      symbol      entrez     uniprot
                 <numeric> <character> <character> <character>
ENSG00000152583 1.32441e-71     SPARCL1        8404      B4E2Z0
ENSG00000179094 6.13966e-56        PER1        5187      A2I2P6
ENSG00000116584 3.49776e-53     ARHGEF2        9181  A0A8Q3SIN5
ENSG00000189221 3.46227e-52        MAOA        4128      B4DF46
ENSG00000120129 1.59454e-44       DUSP1        1843      B4DRR4
ENSG00000148175 1.83034e-42        STOM        2040      F8VSL7
                           genename
                        <character>
ENSG00000152583           SPARC like 1
ENSG00000179094 period circadian reg..
ENSG00000116584 Rho/Rac guanine nucl..
ENSG00000189221    monoamine oxidase A
ENSG00000120129 dual specificity pho..
ENSG00000148175               stomatin
```

Finally, let's write out the ordered significant results with annotations. See the help for ?write.csv if you are unsure here.

```
write.csv(res[ord,], "deseq_results.csv")
```

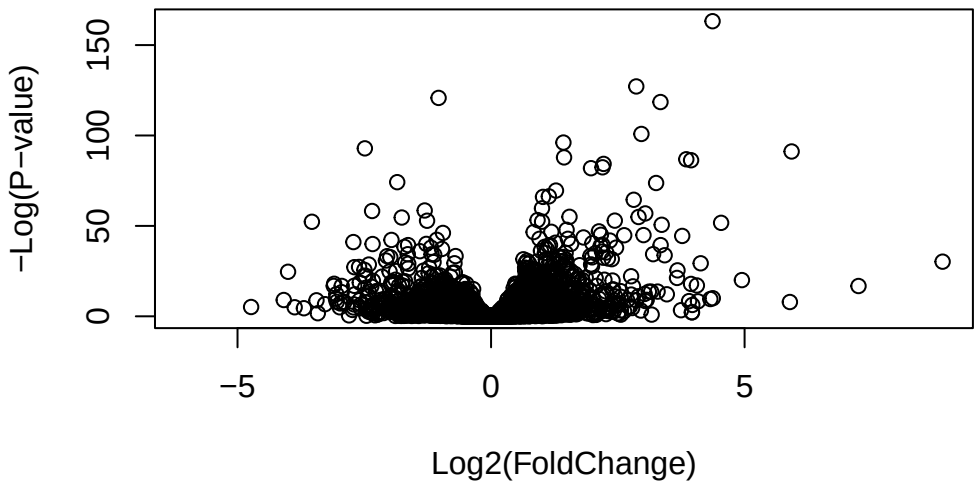
9. Data Visualization

Volcano plots

Let's make a commonly produced visualization from this data, namely a so-called Volcano plot. These summary figures are frequently used to highlight the proportion of genes that are both significantly regulated and display a high fold change.

Typically these plots shows the log fold change on the X-axis, and the $-log_{10}$ of the p-value on the Y-axis (the more significant the p-value, the larger the $-log_{10}$ of that value will be). A very dull (i.e. non colored and labeled) version can be created with a quick call to **plot()** like so:

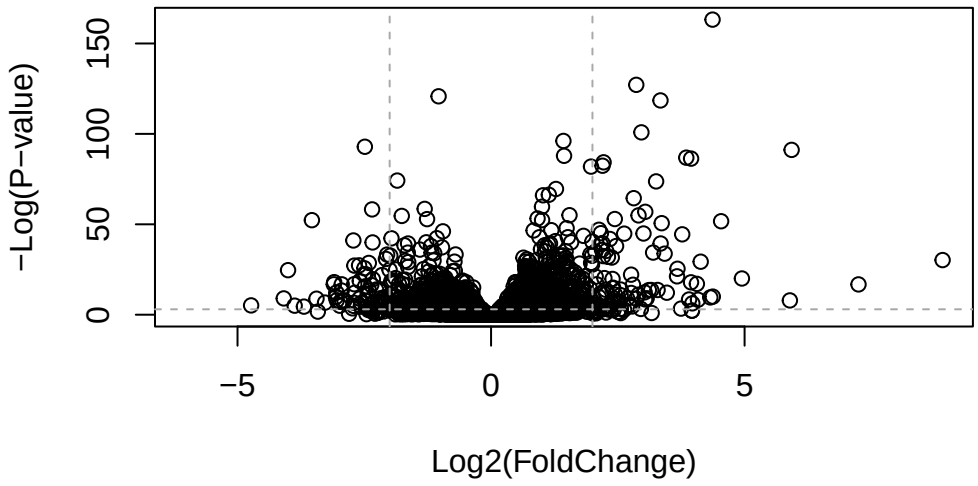
```
plot( res$log2FoldChange,  -log(res$padj),
      xlab="Log2(FoldChange)",
      ylab="-Log(P-value)")
```



To make this more useful we can add some guidelines (with the `abline()` function) and color (with a custom color vector) highlighting genes that have padj<0.05 and the absolute log2FoldChange>2.

```
plot( res$log2FoldChange,  -log(res$padj),
 ylab="-Log(P-value)", xlab="Log2(FoldChange)")

# Add some cut-off lines
abline(v=c(-2,2), col="darkgray", lty=2)
abline(h=-log(0.05), col="darkgray", lty=2)
```



To color the points we will setup a custom color vector indicating transcripts with large fold change and significant differences between conditions:

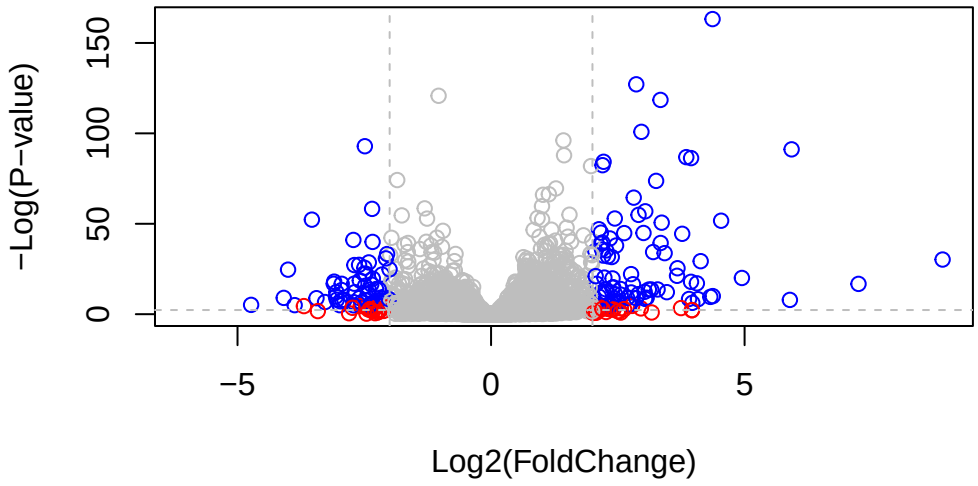
```
# Setup our custom point color vector
mycols <- rep("gray", nrow(res))
mycols[ abs(res$log2FoldChange) > 2 ]  <- "red"

inds <- (res$padj < 0.01) & (abs(res$log2FoldChange) > 2 )
mycols[ inds ] <- "blue"

# Volcano plot with custom colors
plot( res$log2FoldChange,  -log(res$padj),
 col=mycols, ylab="-Log(P-value)", xlab="Log2(FoldChange)" )

# Cut-off lines
```

```
abline(v=c(-2,2), col="gray", lty=2)
abline(h=-log(0.1), col="gray", lty=2)
```



For even more customization you might find the **EnhancedVolcano** bioconductor package useful (Note. It uses **ggplot** under the hood):

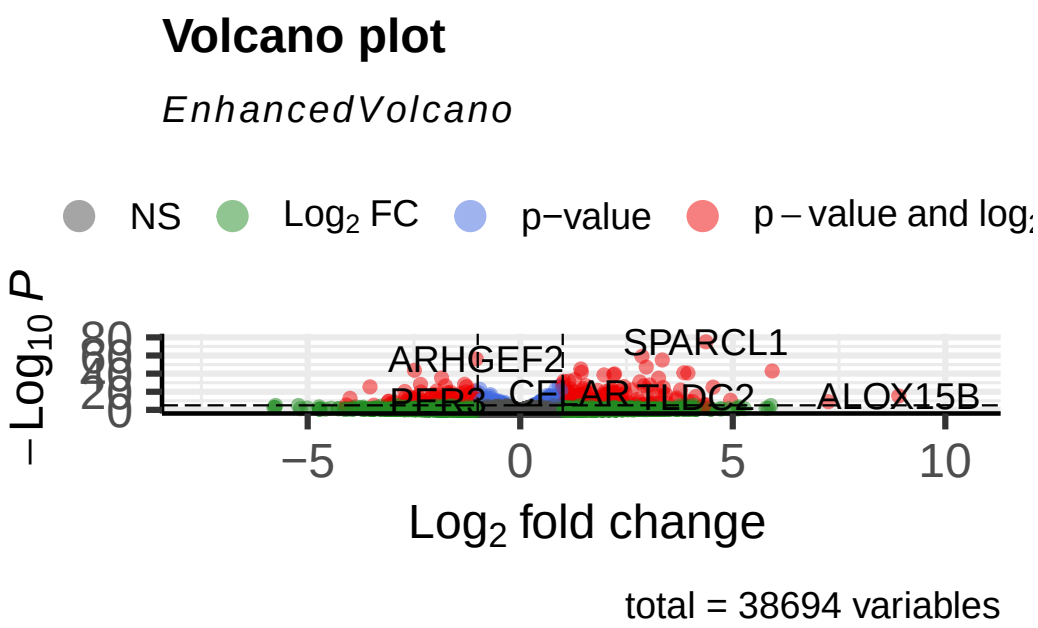
First we will add the more understandable gene *symbol* names to our full results object `res` as we will use this to label the most interesting genes in our final plot.

```
BiocManager::install("EnhancedVolcano")
```

```
library(EnhancedVolcano)
```

```
x <- as.data.frame(res)

EnhancedVolcano(x,
    lab = x$symbol,
    x = 'log2FoldChange',
    y = 'pvalue')
```

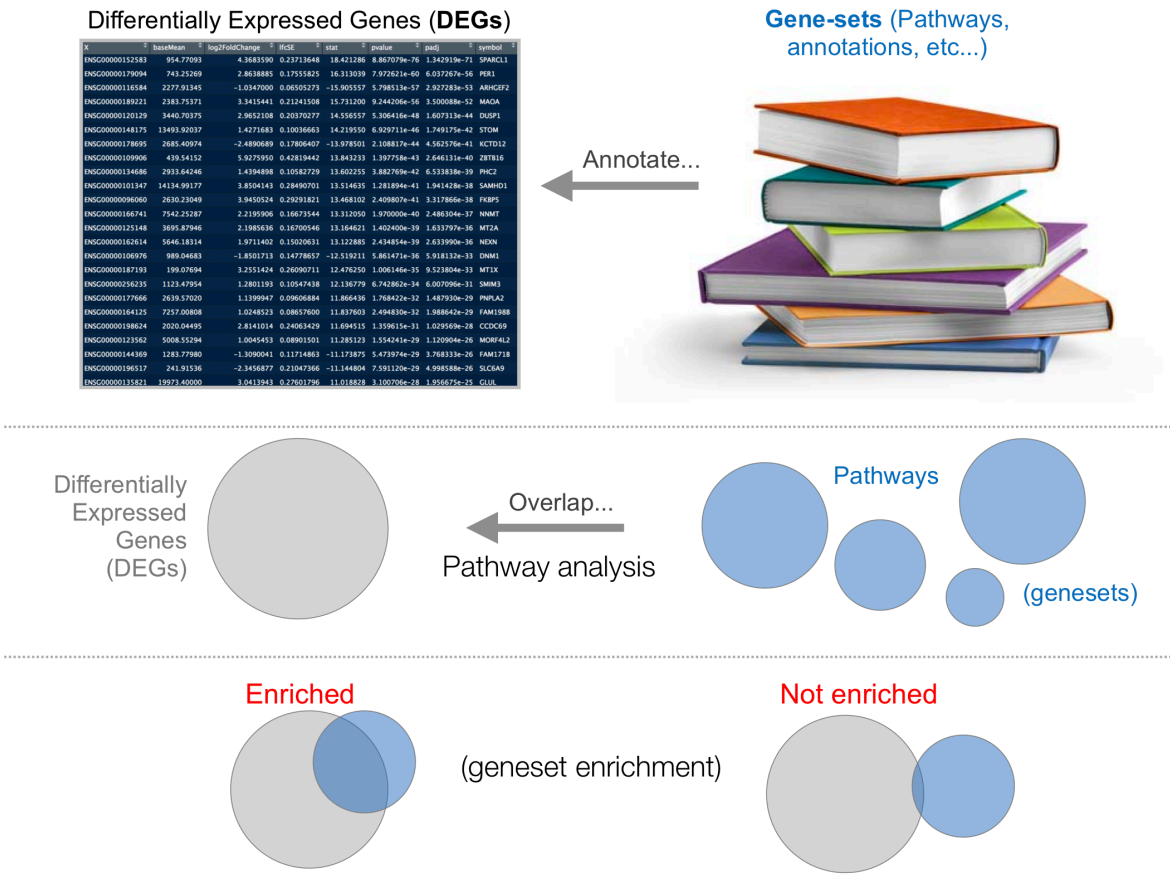


In the final section below we will find out how to derive biological (and hopefully) mechanistic insight from the subset of our most interesting genes highlighted in these types of plots.

10. Pathway analysis

> **OPTIONAL:** This and following sections are optional as we will return to these topics in more detail in class 3 of the course.

Pathway analysis (also known as gene set analysis or over-representation analysis), aims to reduce the complexity of interpreting gene lists via mapping the listed genes to known (i.e. annotated) biological pathways, processes and functions.



Side-note: Pathway analysis can actually mean many different things to different people. This includes analysis of Gene Ontology (GO) terms, protein–protein interaction networks, flux-balance analysis from kinetic simulations of pathways, etc. However, pathway analysis most commonly focuses on methods that exploit existing pathway knowledge (e.g. in public repositories such as GO or KEGG), rather than on methods that infer pathways from molecular measurements. These more general approaches are nicely reviewed in this paper:

- Khatri, et al. "*Ten years of pathway analysis: current approaches and outstanding challenges.*" PLoS Comput Biol 8.2 (2012): e1002375.

Patway analysis with R and Bioconductor

There are many freely available tools for pathway or over-representation analysis. At the time of writing Bioconductor alone has over 145 packages categorized under gene set enrichment and over 195 packages categorized under pathways.

Here we play with just one, the rather old (i.e. well established) **GAGE** package (which stands for **G**enerally **A**pplicable **G**ene set **E**nrichment), to do KEGG pathway enrichment analysis on our RNA-seq based differential expression results.

The KEGG pathway database, unlike GO for example, provides functional annotation as well as information about gene products that interact with each other in a given pathway, how they interact (e.g., activation, inhibition, etc.), and where they interact (e.g., cytoplasm, nucleus, etc.). Hence KEGG has the potential to provide extra insight beyond annotation lists of simple molecular function, process etc. from GO terms.

In this analysis, we check for coordinated differential expression over gene sets from KEGG pathways instead of changes of individual genes. The assumption here is that consistent perturbations over a given pathway (gene set) may suggest mechanistic changes.

Once we have a list of enriched pathways from **gage** we will use the **pathview** package to draw pathway diagrams, coloring the molecules in the pathway by their degree of up/down-regulation.

First we need to do our one time install of these required bioconductor packages:

```
# Run in your R console (i.e. not your Rmarkdown doc!)
BiocManager::install( c("pathview", "gage", "gageData") )
```

Now we can load the packages and setup the KEGG data-sets we need. The **gageData** package has pre-compiled databases mapping genes to KEGG pathways and GO terms for common organisms. `kegg.sets.hs` is a named list of 229 elements. Each element is a character vector of member gene Entrez IDs for a single KEGG pathway.

```
library(pathview)
library(gage)
library(gageData)

data(kegg.sets.hs)

# Examine the first 2 pathways in this kegg set for humans
head(kegg.sets.hs, 2)
```

```
$`hsa00232 Caffeine metabolism`
[1] "10"   "1544" "1548" "1549" "1553" "7498" "9"

$`hsa00983 Drug metabolism - other enzymes`
 [1] "10"     "1066"   "10720"  "10941"  "151531" "1548"   "1549"   "1551"
 [9] "1553"   "1576"   "1577"   "1806"   "1807"   "1890"   "221223" "2990"
```

```
[17] "3251"   "3614"   "3615"   "3704"   "51733"  "54490"  "54575"  "54576"
[25] "54577"  "54578"  "54579"  "54600"  "54657"  "54658"  "54659"  "54963"
[33] "574537" "64816"  "7083"   "7084"   "7172"   "7363"   "7364"   "7365"
[41] "7366"   "7367"   "7371"   "7372"   "7378"   "7498"   "79799"  "83549"
[49] "8824"   "8833"   "9"      "978"
```

The main **gage()** function requires a named vector of fold changes, where the names of the values are the Entrez gene IDs.

Note that we used the **mapIDs()** function above to obtain Entrez gene IDs (stored in `res$entrez`) and we have the fold change results from DESeq2 analysis (stored in `res$log2FoldChange`).

```
foldchanges = res$log2FoldChange
names(foldchanges) = res$entrez
head(foldchanges)
```

```
       7105       64102        8813       57147       55732        2268
-0.35070302          NA  0.20610777  0.02452695 -0.14714205 -1.73228897
```

Now, let's run the **gage** pathway analysis.

```
# Get the results
keggres = gage(foldchanges, gsets=kegg.sets.hs)
```

See help on the gage function with `?gage`. Specifically, you might want to try changing the value of `same.dir`. This value determines whether to test for changes in a gene set toward a single direction (all genes up or down regulated) or changes towards both directions simultaneously (i.e. any genes in the pathway dysregulated). Here, we're using the default `same.dir=TRUE`, which will give us separate lists for pathways that are upregulated versus pathways that are down-regulated.

Now lets look at the object returned from **gage()**.

```
attributes(keggres)
```

```
$names
[1] "greater" "less"    "stats"
```

It is a list with three elements, "greater", "less" and "stats".

You can also see this in your *Environmnet* panel/tab window of RStudio or use the R command `str(keggres)`.

Like any list we can use the dollar syntax to access a named element, e.g. `head(keggres$greater)` and `head(keggres$less)`.

Lets look at the first few down (less) pathway results:

```
# Look at the first three down (less) pathways
head(keggres$less, 3)
```

```
                                    p.geomean stat.mean        p.val
hsa05332 Graft-versus-host disease 0.0004250461 -3.473346 0.0004250461
hsa04940 Type I diabetes mellitus  0.0017820293 -3.002352 0.0017820293
hsa05310 Asthma                    0.0020045888 -3.009050 0.0020045888
                                         q.val set.size         exp1
hsa05332 Graft-versus-host disease 0.09053483       40 0.0004250461
hsa04940 Type I diabetes mellitus  0.14232581       42 0.0017820293
hsa05310 Asthma                    0.14232581       29 0.0020045888
```

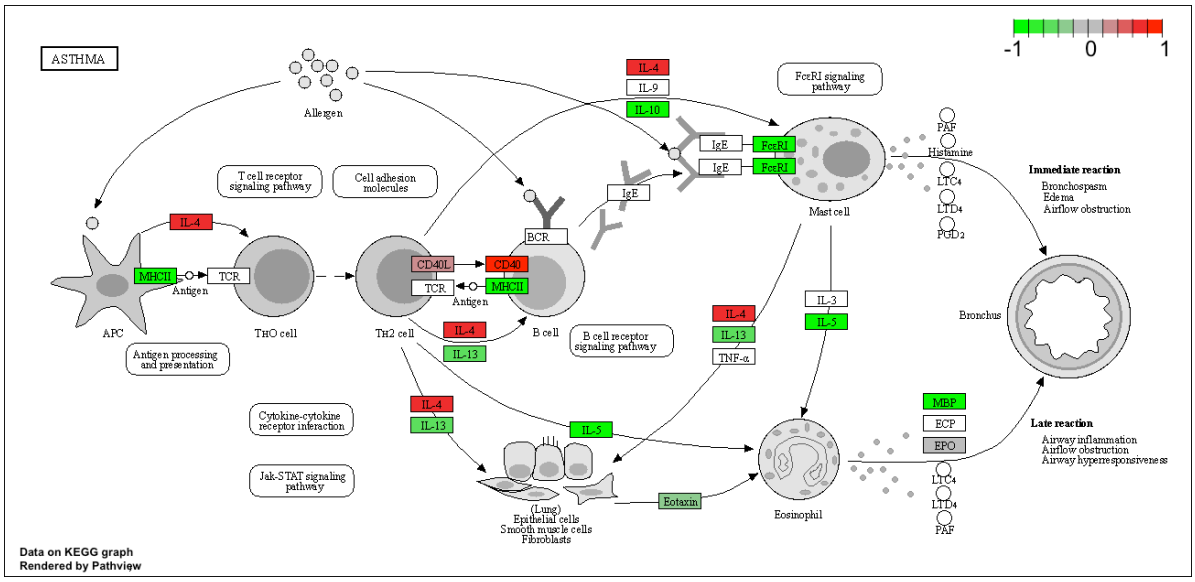
Each `keggres$less` and `keggres$greater` object is data matrix with gene sets as rows sorted by p-value.

The top three Kegg pathways indicated here include Graft-versus-host disease, Type I diabetes and the Asthma pathway (with pathway ID **hsa05310**).

Now, let's try out the **pathview()** function from the pathview package to make a pathway plot with our RNA-Seq expression results shown in color.
To begin with lets manually supply a `pathway.id` (namely the first part of the `"hsa05310 Asthma"`) that we could see from the print out above.

```
pathview(gene.data=foldchanges, pathway.id="hsa05310")
```

This downloads the pathway figure data from KEGG and adds our results to it. Here is the default low resolution raster PNG output from the pathview() call above:



Note how many of the genes in this pathway are perturbed (i.e. colored) in our results.

You can play with the other input arguments to **pathview()** to change the display in various ways including generating a PDF graph. For example:

```
# A different PDF based output of the same data
pathview(gene.data=foldchanges, pathway.id="hsa05310", kegg.native=FALSE)
```

> **Q12**. Can you do the same procedure as above to plot the pathview figures for the top 2 down-reguled pathways?

If you are interested in delving further into pathway analysis you are welcome to check out the optional additional lab session focusing on this topic where we utilize KEGG, GO and Reactome to analyze a different RNA-Seq experiment from start to finish.

OPTIONAL: Plotting counts for genes of interest

DESeq2 offers a function called `plotCounts()` that takes a DESeqDataSet that has been run through the pipeline, the name of a gene, and the name of the variable in the `colData` that you're interested in, and plots those values. See the help for `?plotCounts`. Let's first see what the gene ID is for the CRISPLD2 gene using:

```
i <- grep("CRISPLD2", res$symbol)
res[i,]
```

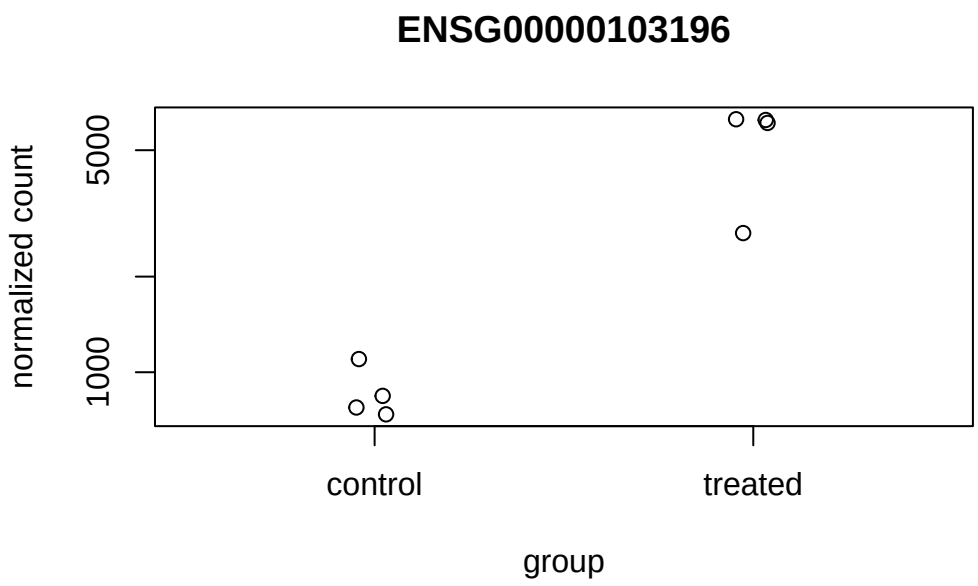
```
log2 fold change (MLE): dex treated vs control
Wald test p-value: dex treated vs control
DataFrame with 1 row and 10 columns
                 baseMean log2FoldChange     lfcSE      stat      pvalue
                <numeric>      <numeric> <numeric> <numeric>   <numeric>
ENSG00000103196   3096.16        2.62603  0.267444   9.81899 9.32747e-23
                      padj      symbol      entrez     uniprot
                  <numeric> <character> <character> <character>
ENSG00000103196 3.36344e-20    CRISPLD2       83716  A0A140VK80
                            genename
                         <character>
ENSG00000103196 cysteine rich secret..
```

```
rownames(res[i,])
```

```
[1] "ENSG00000103196"
```

Now, with that gene ID in hand let's plot the counts, where our `intgroup`, or "interesting group" variable is the "dex" column.

```
plotCounts(dds, gene="ENSG00000103196", intgroup="dex")
```



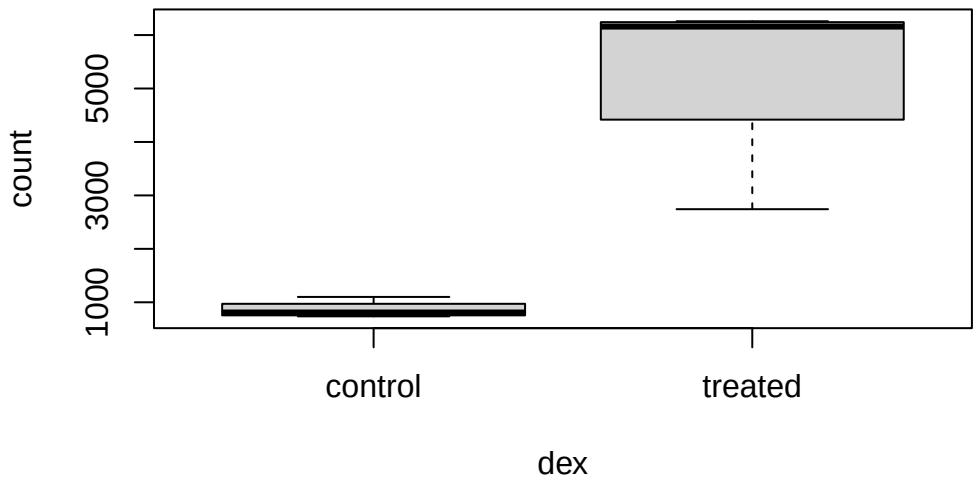
That's just okay. Keep looking at the help for `?plotCounts`. Notice that we could have actually returned the data instead of plotting. We could then pipe this to ggplot and make our own figure. Let's make a boxplot.

```
# Return the data
d <- plotCounts(dds, gene="ENSG00000103196", intgroup="dex", returnData=TRUE)
head(d)
```

```
               count     dex
SRR1039508  774.5002 control
SRR1039509 6258.7915 treated
SRR1039512 1100.2741 control
SRR1039513 6093.0324 treated
SRR1039516  736.9483 control
SRR1039517 2742.1908 treated
```

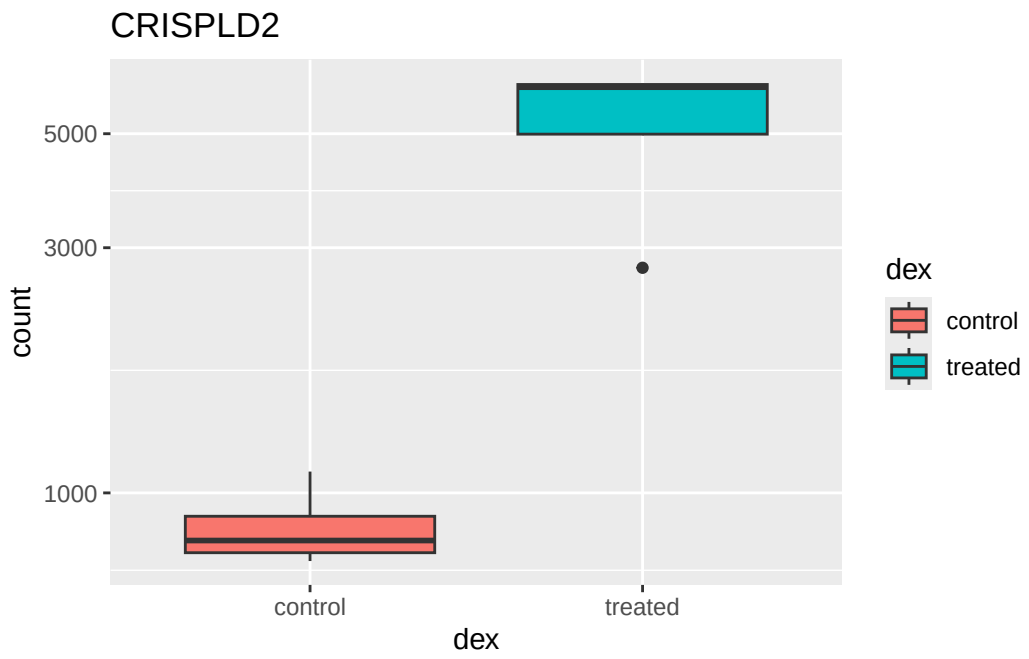
We can mow use this returned object to plot a boxplot with the base graphics function **boxplot()**

```
boxplot(count ~ dex , data=d)
```



As the returned object is a data.frame it is also all setup for ggplot2 based plotting. For example:

```
library(ggplot2)
ggplot(d, aes(dex, count, fill=dex)) +
  geom_boxplot() +
  scale_y_log10() +
  ggtitle("CRISPLD2")
```



Which plot do you prefer? Maybe time to learn more about ggplot via the available DataCamp course ;-) Of course there are many other interesting genes that our results have highlighted and you could now explore these further using the same approach outlined above.

Session Information

The `sessionInfo()` prints version information about R and any attached packages. It's a good practice to always run this command at the end of your R session and record it for the sake of reproducibility in the future.

```
sessionInfo()
```

```
R version 4.4.2 (2024-10-31)
Platform: aarch64-apple-darwin20
Running under: macOS Sequoia 15.3.1

Matrix products: default
BLAS:   /Library/Frameworks/R.framework/Versions/4.4-arm64/Resources/lib/libRblas.0.dylib
LAPACK: /Library/Frameworks/R.framework/Versions/4.4-arm64/Resources/lib/libRlapack.dylib;  L

locale:
```

```
[1] en_US.UTF-8/en_US.UTF-8/en_US.UTF-8/C/en_US.UTF-8/en_US.UTF-8

time zone: America/Los_Angeles
tzcode source: internal

attached base packages:
[1] stats4    stats     graphics  grDevices utils     datasets  methods
[8] base

other attached packages:
 [1] gageData_2.44.0             gage_2.56.0
 [3] pathview_1.46.0             EnhancedVolcano_1.24.0
 [5] ggrepel_0.9.6               org.Hs.eg.db_3.20.0
 [7] AnnotationDbi_1.68.0        ggplot2_3.5.1
 [9] dplyr_1.1.4                 gt_0.11.1
[11] DESeq2_1.46.0               SummarizedExperiment_1.36.0
[13] Biobase_2.66.0              MatrixGenerics_1.18.1
[15] matrixStats_1.5.0           GenomicRanges_1.58.0
[17] GenomeInfoDb_1.42.3         IRanges_2.40.1
[19] S4Vectors_0.44.0            BiocGenerics_0.52.0
[21] BiocManager_1.30.25         labsheet_0.1.2

loaded via a namespace (and not attached):
 [1] tidyselect_1.2.1        farver_2.1.2            blob_1.2.4
 [4] bitops_1.0-9            Biostrings_2.74.1       RCurl_1.98-1.17
 [7] fastmap_1.2.0           XML_3.99-0.18           digest_0.6.37
[10] lifecycle_1.0.4         KEGGgraph_1.66.0        KEGGREST_1.46.0
[13] RSQLite_2.3.9           magrittr_2.0.3          compiler_4.4.2
[16] rlang_1.1.5             tools_4.4.2             yaml_2.3.10
[19] knitr_1.50              S4Arrays_1.6.0          labeling_0.4.3
[22] bit_4.6.0               DelayedArray_0.32.0     xml2_1.3.8
[25] abind_1.4-8             BiocParallel_1.40.0     withr_3.0.2
[28] grid_4.4.2              colorspace_2.1-1        GO.db_3.20.0
[31] scales_1.3.0            cli_3.6.4               rmarkdown_2.29
[34] crayon_1.5.3            generics_0.1.3          rstudioapi_0.17.1
[37] httr_1.4.7              commonmark_1.9.5        DBI_1.2.3
[40] cachem_1.1.0            zlibbioc_1.52.0         parallel_4.4.2
[43] XVector_0.46.0          vctrs_0.6.5             Matrix_1.7-3
[46] jsonlite_1.9.1          litedown_0.6            bit64_4.6.0-1
[49] Rgraphviz_2.50.0        locfit_1.5-9.12         glue_1.8.0
[52] codetools_0.2-20        gtable_0.3.6            UCSC.utils_1.2.0
[55] munsell_0.5.1           tibble_3.2.1            pillar_1.10.1
[58] htmltools_0.5.8.1       graph_1.84.1            GenomeInfoDbData_1.2.13
```

```
[61] R6_2.6.1                evaluate_1.0.3          lattice_0.22-6
[64] markdown_2.0            png_0.1-8               memoise_2.0.1
[67] Rcpp_1.0.14             SparseArray_1.6.2       xfun_0.51
[70] pkgconfig_2.0.3
```