Background
1. Bioconductor and DESeq2 setup
2. Import countData and colData
3. Toy differential gene expression
4. Adding annotation data
5. DESeq2 analysis
6. Data Visualization
Session Information

BIMM-143, Lecture 14

Code

Transcriptomics and the analysis of RNA-Seq data

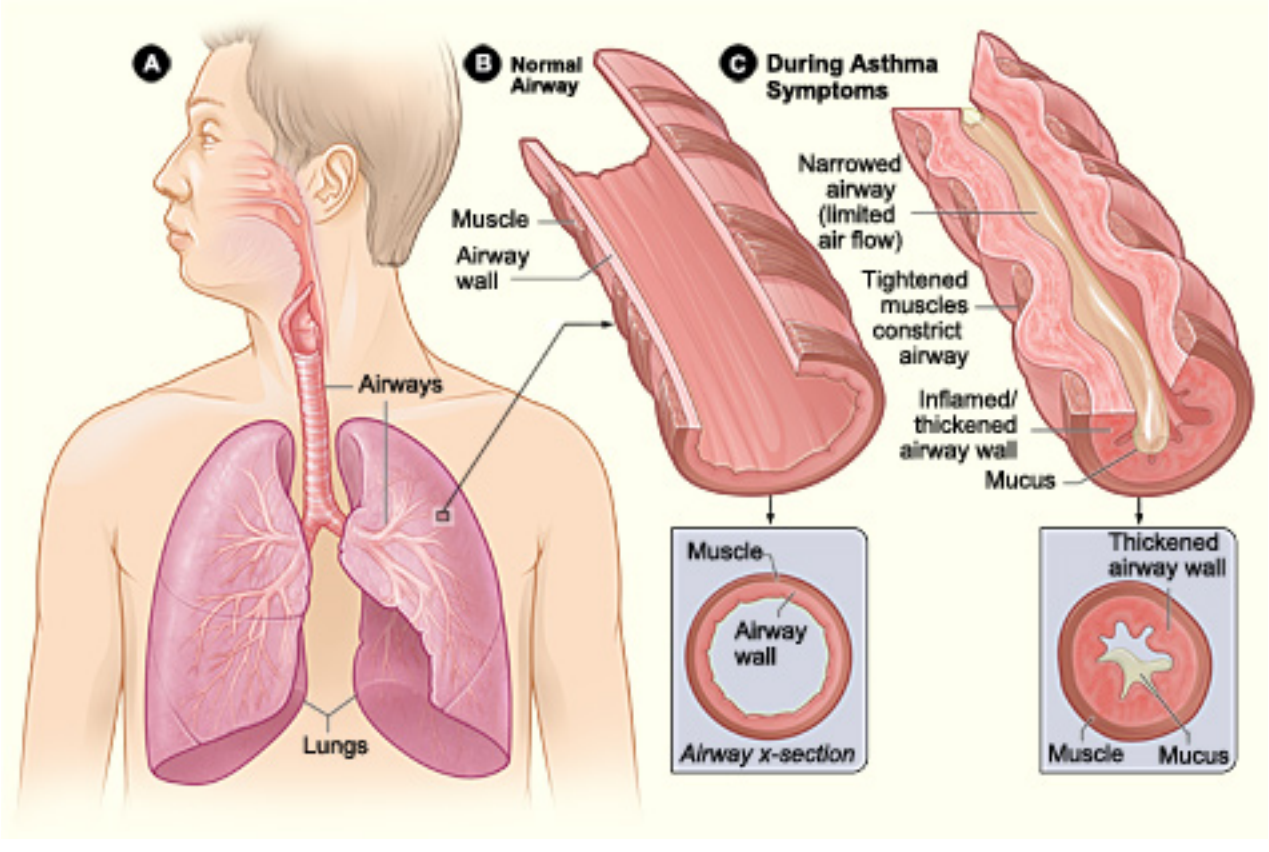
Barry Grant < http://thegrantlab.org/bimm143/ (http://thegrantlab.org/bimm143/) >
2019-05-16 (09:18:47 on Thu, May 16)

Background

The data for this hands-on session comes from a published RNA-seq experiment where airway smooth muscle cells were treated with dexamethasone (https://en.wikipedia.org/wiki/Dexamethasone), a synthetic **glucocorticoid steroid** with

anti-inflammatory effects (Himes et al. 2014 (http://www.ncbi.nlm.nih.gov/pubmed/24926665)).

Glucocorticoids are used, for example, by people with asthma to reduce inflammation of the airways. 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 comparing dexamethasone-treated to control cells, but focus much of the discussion on a single 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. They confirmed the upregulated CRISPLD2 mRNA expression with qPCR and increased protein expression using Western blotting.

In the experiment, four primary human ASM cell lines were treated with 1 micromolar dexamethasone for 18 hours. For each of the four cell lines, we have a treated and an untreated sample. They did their analysis using **Tophat** and **Cufflinks** similar to our work in the last hands-on session. For a more detailed description of their analysis see the PubMed entry 24926665 (http://www.ncbi.nlm.nih.gov/pubmed/24926665) and for raw data see the GEO entry GSE52778 (http://www.ncbi.nlm.nih.gov/geo/query/acc.cgi?acc=GSE52778).

In this session we will read and explore the gene expression data from this experiment using base R functions and then perform a detailed analysis with the **DESeq2** package from Bioconductor (http://www.bioconductor.org).

1. Bioconductor and DESeq2 setup

As we already noted back in Lecture 7 (https://bioboot.github.io/bimm143_S19/lectures/#7) Bioconductor is a large repository and resource for R packages that focus on analysis of high-throughput genomic data.

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

Hide

Hide

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

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

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 (https://bioboot.github.io/bimm143_S19/class-material/bioconductor_setup/). 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 (https://bioboot.github.io/bimm143_S19/class-material/bioconductor_setup/) for debugging steps.

Check that you have installed everything correctly by closing and reopening RStudio and entering the following two commands at the console window:

Hide

Hide

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

If you get a message that says something like:
`Error in library(BiocManager) : there is no package called 'BiocManager'`, then the required packages did not install correctly. Please see our Bioconductor setup notes (https://bioboot.github.io/bimm143_S19/class-material/bioconductor_setup/) 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 (https://en.wikipedia.org/wiki/FASTQ_format) 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 (https://bioboot.github.io/bimm143_W18/class-material/sam_format/). 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 (https://pachterlab.github.io/kallisto/about), paper (http://www.nature.com/nbt/journal/v34/n5/full/nbt.3519.html)) and transcript-level abundance estimates were then summarized to the gene level to produce length-scaled counts using the R package **txImport** (software (https://bioconductor.org/packages/tximport), paper (https://f1000research.com/articles/4-1521/v2)), 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 (http://genomebiology.biomedcentral.com/articles/10.1186/s13059-016-0881-8) (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 (https://f1000research.com/articles/4-1521/v2) (2016).
3. Griffith, Malachi, et al. “Informatics for RNA sequencing: a web resource for analysis on the cloud.” *PLoS Comput Biol* 11.8: e1004393 (http://journals.plos.org/ploscompbiol/article?id=10.1371/journal.pcbi.1004393) (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).

countData

gene	ctrl_1	ctrl_2	exp_1	exp_1
geneA	10	11	56	45
geneB	0	0	128	54
geneC	42	41	59	41
geneD	103	122	1	23
geneE	10	23	14	56
geneF	0	1	2	0
…	…	…	…	…
…	…	…	…	…
…	…	…	…	…

colData

id	treatment	sex
ctrl_1	control	male
ctrl_2	control	female
exp_1	treatment	male
exp_2	treatment	female

Sample names:
ctrl_1, **ctrl_2**, **exp_1**, **exp_2**

countData is the count matrix
(number of reads mapping to each gene for each sample)

colData describes metadata about the *columns* of countData

First column of colData must match column names of countData (-1st)

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).

> **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.

2. Import countData and colData

First, create a new RStudio project in your GitHub tracked directory (File > New Project > New Directory > New Project) and download the input airway_scaledcounts.csv (https://bioboot.github.io/bimm143_W18/class-material/airway_scaledcounts.csv) and airway_metadata.csv (https://bioboot.github.io/bimm143_W18/class-material/airway_metadata.csv) into a new `data` sub-directory of your project directory.

Begin a new Rmarkdown document and use the **read.csv()** function to read these count data and metadata files.

Hide

Hide

```
counts <- read.csv("data/airway_scaledcounts.csv", stringsAsFactors = F
ALSE)
metadata <-  read.csv("data/airway_metadata.csv", stringsAsFactors = FA
LSE)
```

Now, take a look at each.

Hide

Hide

```
head(counts)
```

ensgene <chr>	SRR1039... <dbl>	SRR1039... <dbl>	SRR1039... <dbl>	SRR1039... <dbl>	SRR1039... <dbl>	S
1 ENSG00000000003	723	486	904	445	1170	
2 ENSG00000000005	0	0	0	0	0	
3 ENSG00000000419	467	523	616	371	582	
4 ENSG00000000457	347	258	364	237	318	
5 ENSG00000000460	96	81	73	66	118	
6 ENSG00000000938	0	0	1	0	2	

6 rows | 1-8 of 10 columns

Hide

Hide

```
head(metadata)
```

	id <chr>	dex <chr>	celltype <chr>	geo_id <chr>
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

6 rows

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 first column, which contains the Ensembl gene ID. This is important, and we'll get more strict about it later on.

3. 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`

Hide

Hide

```
View(metadata)
```

If we look at our metadata, we see 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:

Hide

Hide

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

> **Q1**. How would you make the above code more robust? What would happen if you were to add more samples. Would the values obtained with the excat code above be correct?

> **Q2**. Follow the same procedure for the `treated` samples (i.e. calculate the mean per gene accross drug treated samples and assign to a labeled vector called `treated.mean`)

We will combine our meancount data for bookkeeping purposes.

Hide

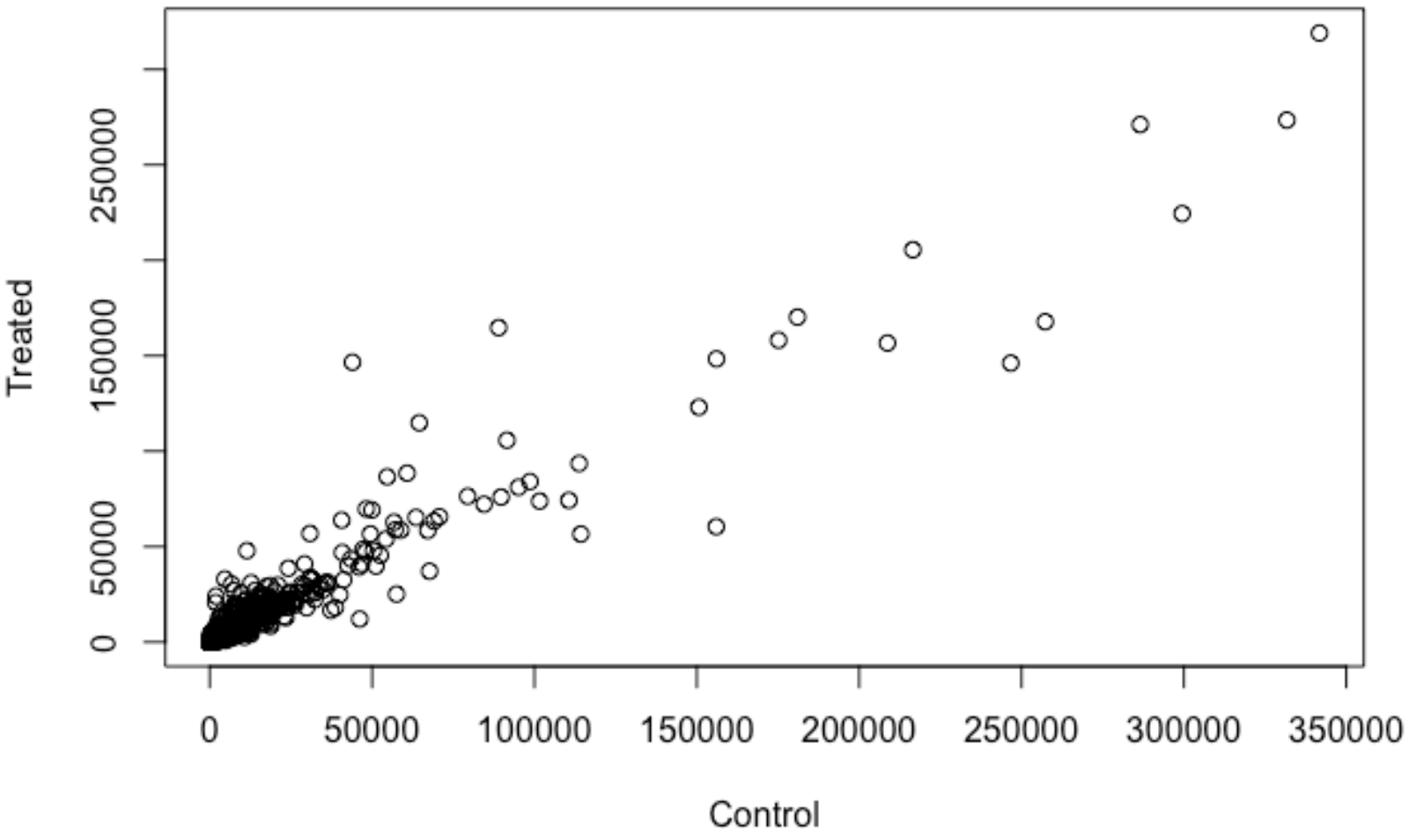
Hide

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

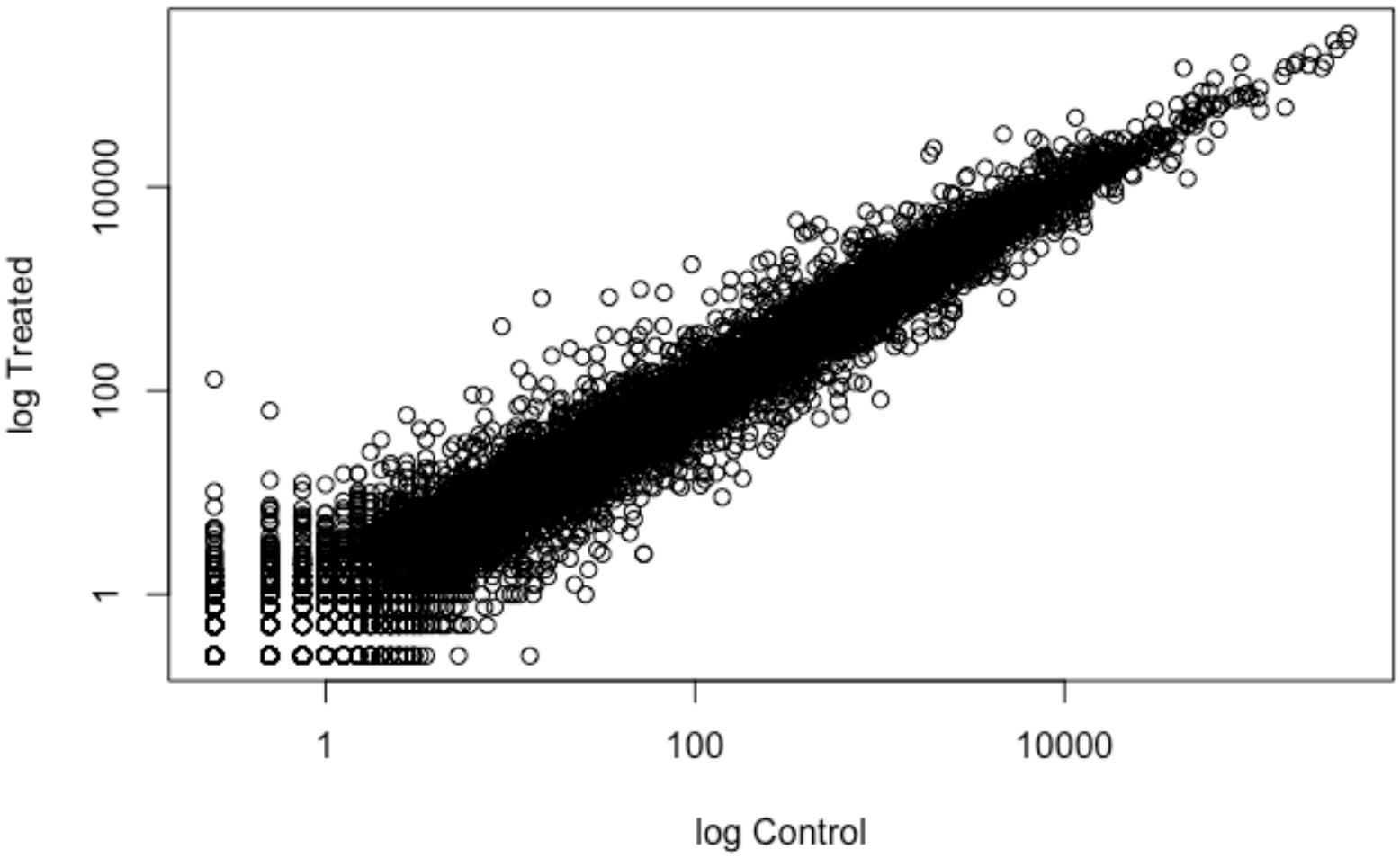
Directly comparing the raw counts is going to be problematic if we just happened to sequence one group at a higher depth than another. Later on we'll do this analysis properly, normalizing by sequencing depth per sample using a better approach. But for now, **colSums()** the data to show the sum of the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

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

> **Q3**. 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.



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. Try plotting both axes on a log scale (hint: see the help for **?plot.default** to see how to set log 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.

Hide

Hide

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

	control.mean <dbl>	treated.mean <dbl>	log2fc <dbl>
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

6 rows

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

Hide

Hide

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

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

	control.mean <dbl>	treated.mean <dbl>	log2fc <dbl>
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

6 rows

Q4. 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?

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.

Hide

Hide

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

> **Q5**. Using the `up.ind` and `down.ind` vectors above can you determine how many up and down regulated genes we have at the greater than 2 fc level?

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

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

In total, you should of reported 617 differentially expressed genes, in either direction.

4. Adding annotation data

Our `mycounts` result table so far only contains the Ensembl gene IDs. However, alternative gene names and extra annotation are usually required for informative for interpretation.

We can add annotation from a supplied CSV file, such as those available from ENSEMBLE or UCSC. The annotables_grch38.csv (https://bioboot.github.io/bimm143_W18/class-material/annotables_grch38.csv) annotation table links the unambiguous Ensembl gene ID to other useful annotation like the gene symbol, full gene name, location, Entrez gene ID, etc.

Hide

Hide

```
anno <- read.csv("data/annotables_grch38.csv")
head(anno)
```

ensgene	ent...	symbol	chr	start	end	stra...	biotype

	<fctr>	<int>	<fctr>	<fctr>	<int>	<int>	<int>	<fctr>
1	ENSG00000000003	7105	TSPAN6	X	100627109	100639991	-1	protein_cod
2	ENSG00000000005	64102	TNMD	X	100584802	100599885	1	protein_cod
3	ENSG00000000419	8813	DPM1	20	50934867	50958555	-1	protein_cod
4	ENSG00000000457	57147	SCYL3	1	169849631	169894267	-1	protein_cod
5	ENSG00000000460	55732	C1orf112	1	169662007	169854080	1	protein_cod
6	ENSG00000000938	2268	FGR	1	27612064	27635277	-1	protein_cod

6 rows | 1-9 of 10 columns

Ideally we want this annotation data mapped (or merged) with our `mycounts` data. In a previous class on writing R functions we introduced the **merge()** function, which is one common way to do this.

> **Q6**. From consulting the help page for the **merge()** function can you set the `by.x` and `by.y` arguments appropriately to annotate our `mycounts` data.frame with all the available annotation data in your `anno` data.frame?

Examine your results with the **View()** function. It should look something like the following:

	Row.names <S3: AsIs>	**control.mean** <dbl>	**treated.mean** <dbl>	**log2fc** <dbl>	**ent…** <int>	**symbol** <fctr>	**chr** <fc
1	ENSG00000000003	900.75	658.00	-0.45303916	7105	TSPAN6	X
2	ENSG00000000419	520.50	546.00	0.06900279	8813	DPM1	20
3	ENSG00000000457	339.75	316.50	-0.10226805	57147	SCYL3	1
4	ENSG00000000460	97.25	78.75	-0.30441833	55732	C1orf112	1
5	ENSG00000000971	5219.00	6687.50	0.35769358	3075	CFH	1
6	ENSG00000001036	2327.00	1785.75	-0.38194109	2519	FUCA2	6

6 rows | 1-9 of 13 columns

In cases where you don't have a preferred annotation file at hand you can use other Bioconductor packages for annotation.

Bioconductor's annotation packages help with mapping various ID schemes to each other. Here we load the AnnotationDbi package and the annotation package org.Hs.eg.db.

Hide

Hide

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

> **Note**: You may have to install these with the `biocLite("AnnotationDbi")` function etc.

This 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, use:

Hide

Hide

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

```
##  [1] "ACCNUM"       "ALIAS"        "ENSEMBL"      "ENSEMBLPROT"
##  [5] "ENSEMBLTRANS" "ENTREZID"     "ENZYME"       "EVIDENCE"
##  [9] "EVIDENCEALL"  "GENENAME"     "GO"           "GOALL"
## [13] "IPI"          "MAP"          "OMIM"         "ONTOLOGY"
## [17] "ONTOLOGYALL"  "PATH"         "PFAM"         "PMID"
## [21] "PROSITE"      "REFSEQ"       "SYMBOL"       "UCSCKG"
## [25] "UNIGENE"      "UNIPROT"
```

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.

Hide

Hide

```
mycounts$symbol <- mapIds(org.Hs.eg.db,
                          keys=row.names(mycounts),
                          column="SYMBOL",
                          keytype="ENSEMBL",
                          multiVals="first")
```

```
## 'select()' returned 1:many mapping between keys and columns
```

Q7. Run the **mapIds()** function two more times to add the Entrez ID and UniProt accession as new columns called `mycounts$entrez` and `mycounts$uniprot`. The **head()** of your results should look like the following:

```
## 'select()' returned 1:many mapping between keys and columns
## 'select()' returned 1:many mapping between keys and columns
```

	control.mean <dbl>	treated.mean <dbl>	log2fc <dbl>	symbol <chr>	ent… <chr>	unip <chr
ENSG00000000003	900.75	658.00	-0.45303916	TSPAN6	7105	A0A(
ENSG00000000419	520.50	546.00	0.06900279	DPM1	8813	O60
ENSG00000000457	339.75	316.50	-0.10226805	SCYL3	57147	Q8IZ
ENSG00000000460	97.25	78.75	-0.30441833	C1orf112	55732	A0A(
ENSG00000000971	5219.00	6687.50	0.35769358	CFH	3075	A0A(
ENSG00000001036	2327.00	1785.75	-0.38194109	FUCA2	2519	Q9B

6 rows

Q8. Examine your annotated results for those genes with a log2(FoldChange) of greater than 2 (or less than -2 if you prefer) with the **View()** function. What do you notice? Would you trust these results? Why or why not?

Hide

Hide

```
head(mycounts[up.ind,])
```

	control.mean <dbl>	treated.mean <dbl>	log2fc <dbl>	symbol <chr>	ent... <chr>	uniprot <chr>
ENSG00000004799	270.50	1429.25	2.401558	PDK4	5166	A4D1H4
ENSG00000006788	2.75	19.75	2.844349	MYH13	8735	Q9UKX3
ENSG00000008438	0.50	2.75	2.459432	PGLYRP1	8993	O75594
ENSG00000011677	0.50	2.25	2.169925	GABRA3	2556	P34903
ENSG00000015413	0.50	3.00	2.584963	DPEP1	1800	A0A140VJ
ENSG00000015592	0.50	2.25	2.169925	STMN4	81551	Q9H169

6 rows

5. DESeq2 analysis

Let's do this the right way. DESeq2 is an R package for analyzing count-based NGS data like RNA-seq. It is available from Bioconductor (http://www.bioconductor.org/). 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 (http://www.bioconductor.org/packages/release/BiocViews.html#___Software).

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 (http://www.bioconductor.org/packages/release/bioc/vignettes/DESeq2/inst/doc/DESeq2.pdf). 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 (setup.html#bioconductor)), then load them every time you start a new R session.

Hide

Hide

```
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.

Hide

Hide

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

DESeq pipeline

Next, let's run the DESeq pipeline on the dataset, and reassign the resulting object back to the same variable. Before we start, `dds` is a bare-bones DESeqDataSet. The `DESeq()` function takes a DESeqDataSet and returns a DESeqDataSet, but with lots of other information filled in (normalization, dispersion estimates, differential expression results, etc). Notice how if we try to access these objects before running the analysis, nothing exists.

Hide

Hide

```
sizeFactors(dds)
```

```
## NULL
```

Hide

Hide

```
dispersions(dds)
```

```
## NULL
```

Hide

Hide

```
results(dds)
```

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

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).

Hide

Hide

```
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.).

Hide

Hide

```
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             lfc
SE
##                         <numeric>          <numeric>         <numeri
c>
## ENSG00000000003  747.194195359907  -0.35070302068658 0.1682456813325
29
## ENSG00000000005                 0                 NA
NA
## ENSG00000000419  520.134160051965  0.206107766417862 0.1010592180080
52
## ENSG00000000457  322.664843927049 0.0245269479387466 0.1451450676492
48
## ENSG00000000460   87.682625164828  -0.14714204922212 0.2570072539946
73
## ...                         ...                ...               .
..
## ENSG00000283115                 0                 NA
NA
## ENSG00000283116                 0                 NA
NA
## ENSG00000283119                 0                 NA
NA
## ENSG00000283120 0.974916032393564  -0.66825846051647  1.694562852418
71
## ENSG00000283123                 0                 NA
NA
##                             stat             pvalue              p
adj
##                        <numeric>          <numeric>         <numer
ic>
## ENSG00000000003  -2.08446967499531 0.0371174658432818 0.163034808641
677
## ENSG00000000005                NA                 NA
NA
## ENSG00000000419   2.03947517584631 0.0414026263001157 0.176031664879
167
```

```
## ENSG00000000457  0.168982303952742  0.865810560623564 0.961694238404
392
## ENSG00000000460  -0.57252099672319  0.566969065257939 0.815848587637
731
## ...                            ...                ...
...
## ENSG00000283115                NA                 NA
NA
## ENSG00000283116                NA                 NA
NA
## ENSG00000283119                NA                 NA
NA
## ENSG00000283120 -0.394354484734893  0.693319342566817
NA
## ENSG00000283123                NA                 NA
NA
```

Either click on the `res` object in the environment pane or 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 (http://www.bioconductor.org/packages/release/bioc/vignettes/DESeq2/inst/doc/DESeq2.pdf) 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.

Hide

Hide

```
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
```

We can order our results table by the smallest p value:

Hide

Hide

```
resOrdered <- res[order(res$pvalue),]
```

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:

Hide

Hide

```
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
```

The more generic way to access the actual subset of the data.frame passing a threshold like this is with the **subset()** function, e.g.:

Hide

Hide

```
resSig05 <- subset(as.data.frame(res), padj < 0.05)
nrow(resSig05)
```

```
## [1] 2181
```

Q9. How many are significant with an adjusted p-value < 0.05? How about 0.01? Save this last set of results as `resSig01`.

Q10. Using either the previously generated `anno` object (annotations from the file `annotables_grch38.csv` file) or the **mapIds()** function (from the AnnotationDbi package) add annotation to your `res01` results data.frame.

```
## [1] 1437
```

```
## 'select()' returned 1:many mapping between keys and columns
```

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

Hide

Hide

```
ord <- order( resSig01$padj )
#View(res01[ord,])
head(resSig01[ord,])
```

	baseMean	log2FoldChange	lfcSE	stat	pvalu
	<dbl>	<dbl>	<dbl>	<dbl>	<db
ENSG00000152583	954.7709	4.368359	0.23712679	18.42204	8.744898e-
ENSG00000179094	743.2527	2.863889	0.17556931	16.31201	8.107836e-
ENSG00000116584	2277.9135	-1.034701	0.06509844	-15.89440	6.928546e-
ENSG00000189221	2383.7537	3.341544	0.21240579	15.73189	9.144326e-

ENSG00000120129	3440.7038	2.965211	0.20369513	14.55710	5.264243e-
ENSG00000148175	13493.9204	1.427168	0.10038904	14.21638	7.251278e-

6 rows | 1-6 of 8 columns

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

Hide

Hide

```
write.csv(resSig01[ord,], "signif01_results.csv")
```

6. Data Visualization

Plotting counts

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:

Hide

Hide

```
i <- grep("CRISPLD2", resSig01$symbol)
resSig01[i,]
```

	baseM...	log2FoldChange	lfcSE	stat	pvalue
	<dbl>	<dbl>	<dbl>	<dbl>	<dbl>
ENSG00000103196	3096.159	2.626034	0.2674445	9.818988	9.327474e-23

1 row | 1-7 of 8 columns

Hide

Hide

```
rownames(resSig01[i,])
```

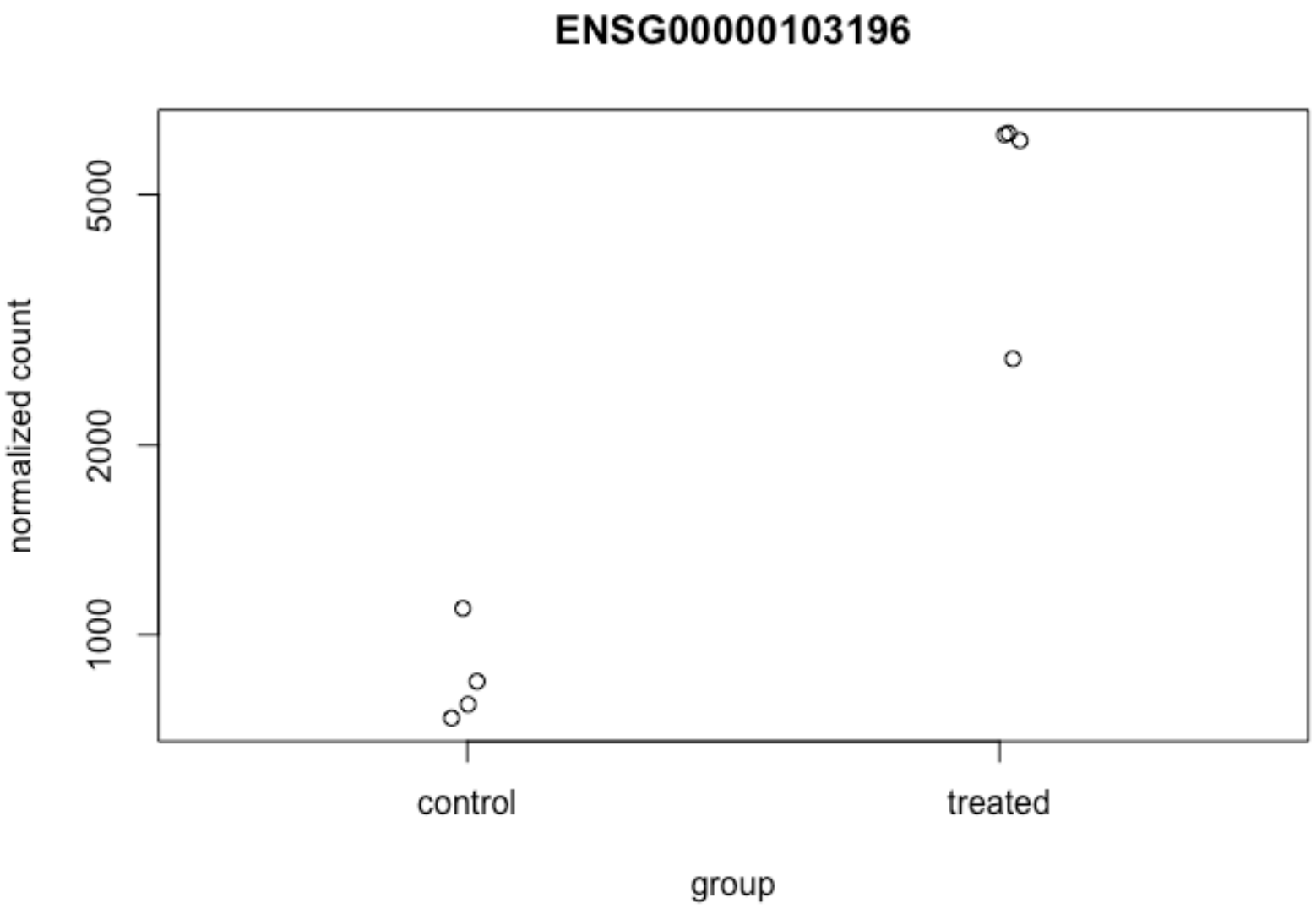
```
## [1] "ENSG00000103196"
```

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

Hide

Hide

```
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.

Hide

Hide

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

	count	dex
	<dbl>	<fctr>
SRR1039508	774.5002	control
SRR1039509	6258.7915	treated
SRR1039512	1100.2741	control
SRR1039513	6093.0324	treated
SRR1039516	736.9483	control
SRR1039517	2742.1908	treated

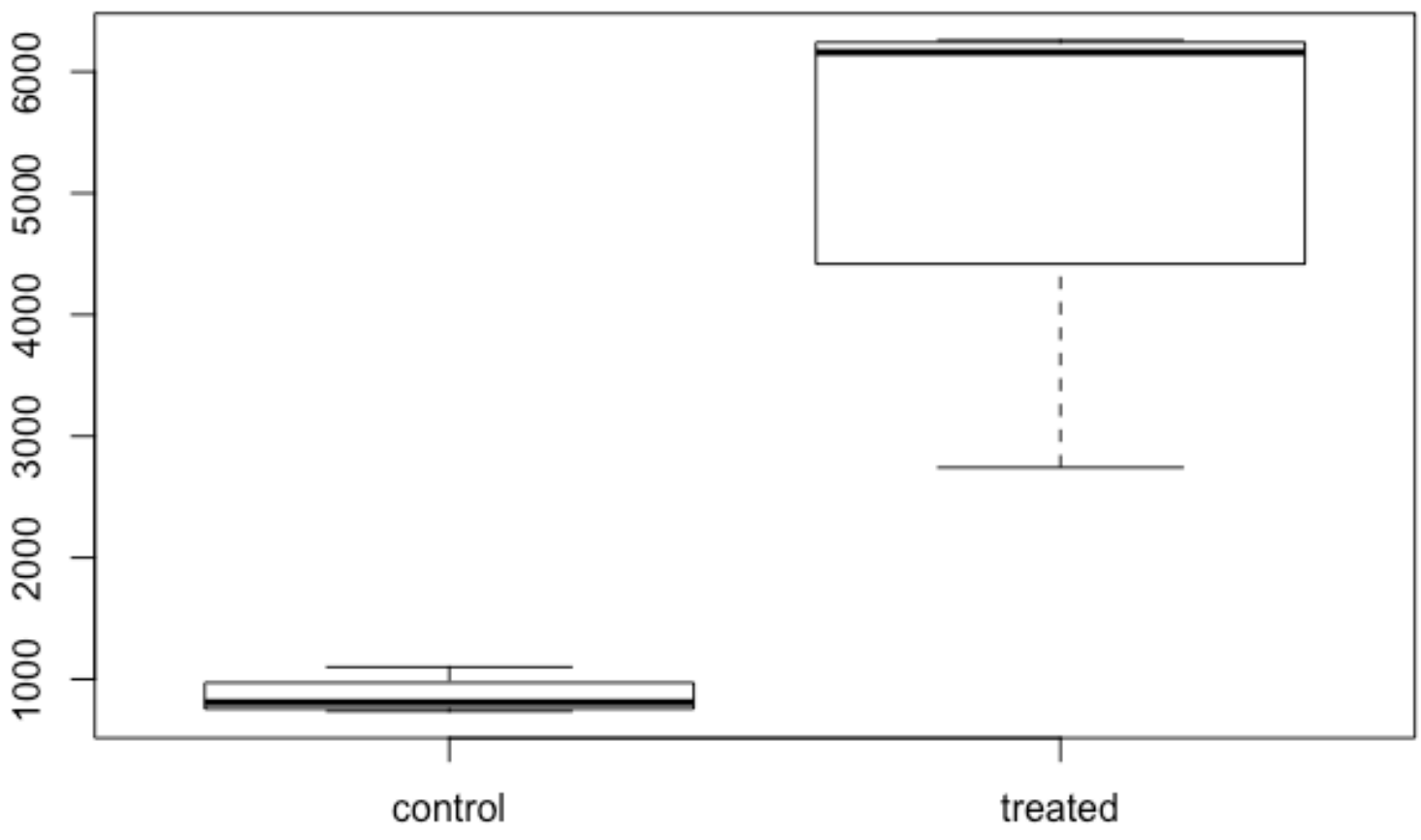
6 rows

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

Hide

Hide

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



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

Hide

Hide

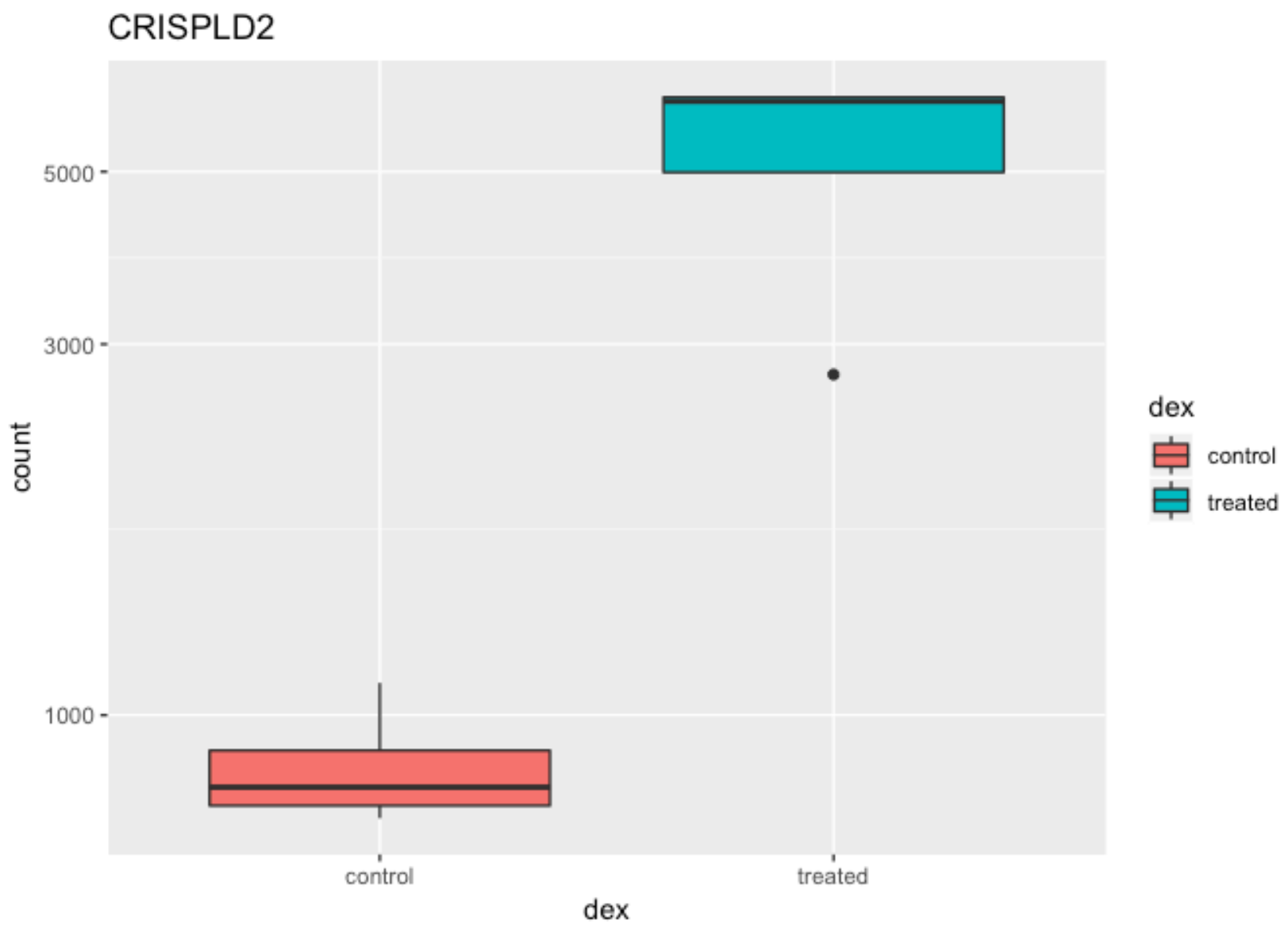
```
library(ggplot2)
```

```
## Warning: package 'ggplot2' was built under R version 3.5.2
```

Hide

Hide

```
ggplot(d, aes(dex, count)) + geom_boxplot(aes(fill=dex)) + scale_y_log1
0() + ggtitle("CRISPLD2")
```



Which plot do you prefer? Maybe time to learn ggplot via the DataCamp course ;-)

Volcano plots

Let's make another commonly produced visualization from this data, namely so-called Volcano plots (https://en.wikipedia.org/wiki/Volcano_plot_(statistics)). These summary figures are frequently used to highlight the proportion of genes that are both significantly regulated and display a high fold change.

First, let's add a column called `sig` to our full `res` results that evaluates to TRUE only if padj<0.05 and the absolute log2FoldChange>2, FALSE if not, and NA if padj is also NA.

Hide

Hide

```
res$sig <- res$padj<0.05 & abs(res$log2FoldChange)>2

# How many of each?
table(res$sig)
```

```
##
## FALSE  TRUE
## 24282   167
```

Hide

Hide

```
sum(is.na(res$sig))
```

```
## [1] 14245
```

A volcano plot 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).

Here we first make a volcano plot with base graphics and color by our `res$sig+1` (we add 1 so we dont have 0 as a color and end up with white points):

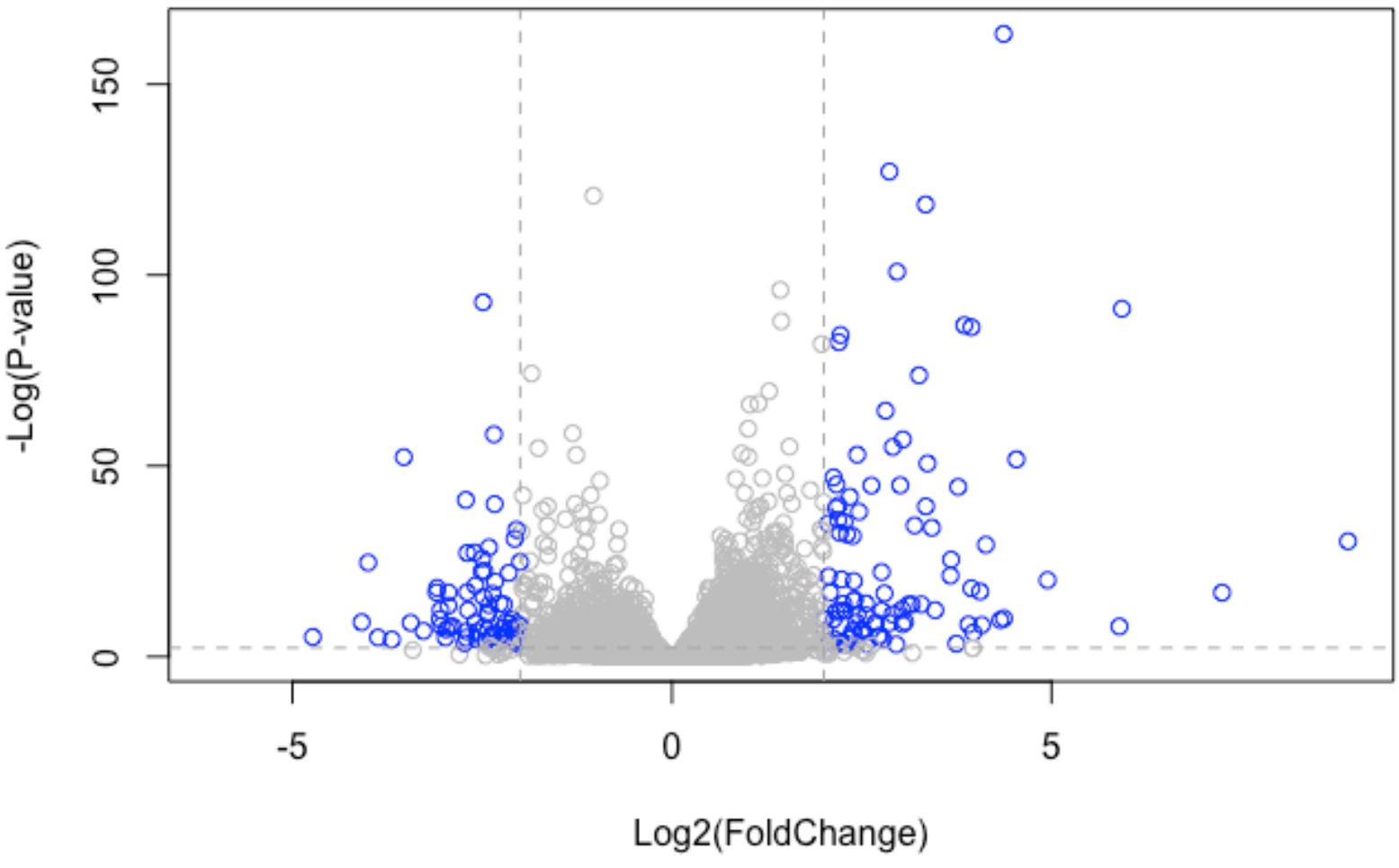
Hide

Hide

```
# Set the color palette for our plot
palette( c("gray","blue") )

plot( res$log2FoldChange,  -log(res$padj),
 col=res$sig+1, ylab="-Log(P-value)", xlab="Log2(FoldChange)")

# Add some cut-off lines
abline(v=c(-2,2), col="darkgray", lty=2)
abline(h=-log(0.1), col="darkgray", lty=2)
```



Hide

Hide

```
# Reset the color palette
palette("default")
```

We could also setup a custom color vector indicating transcripts with large fold change and significant differences between conditions:

Hide

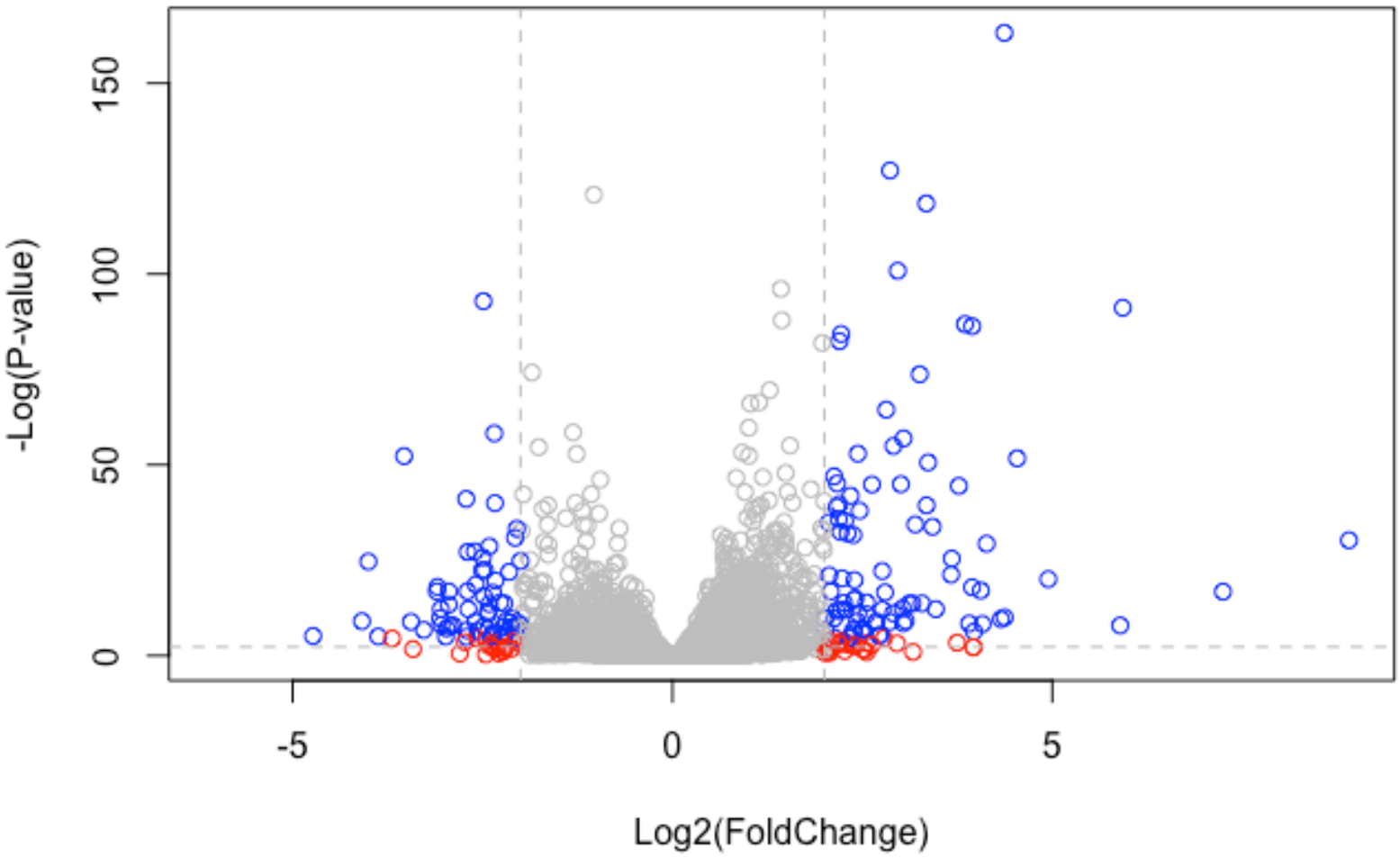
Hide

```
# 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)" )

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



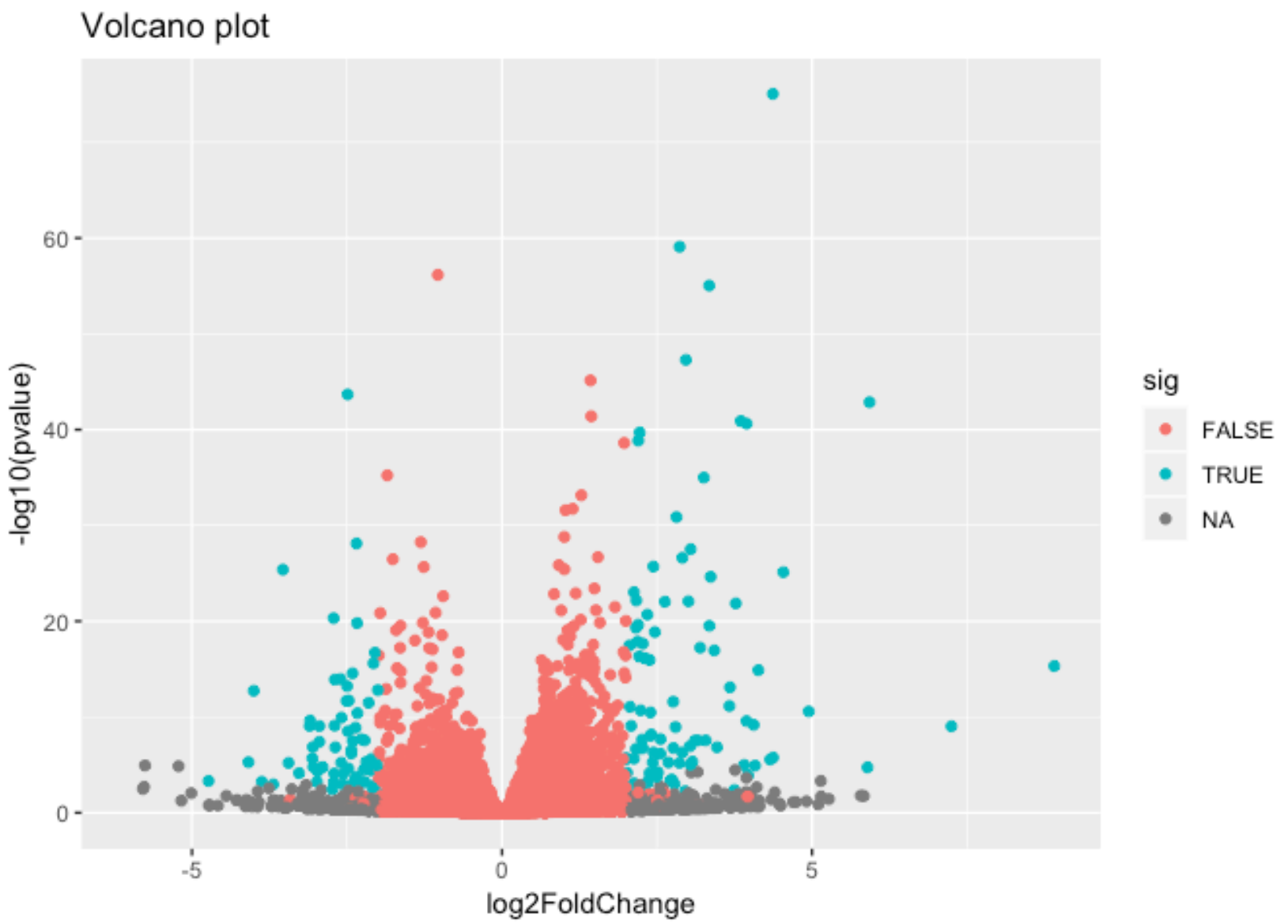
and do the same plot with **ggplot**…

Hide

Hide

```
ggplot(as.data.frame(res), aes(log2FoldChange, -log10(pvalue), col=sig)
) +
    geom_point() +
    ggtitle("Volcano plot")
```

```
## Warning: Removed 13578 rows containing missing values (geom_point).
```



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

Hide

Hide

```
biocLite("EnhancedVolcano")
```

Hide

Hide

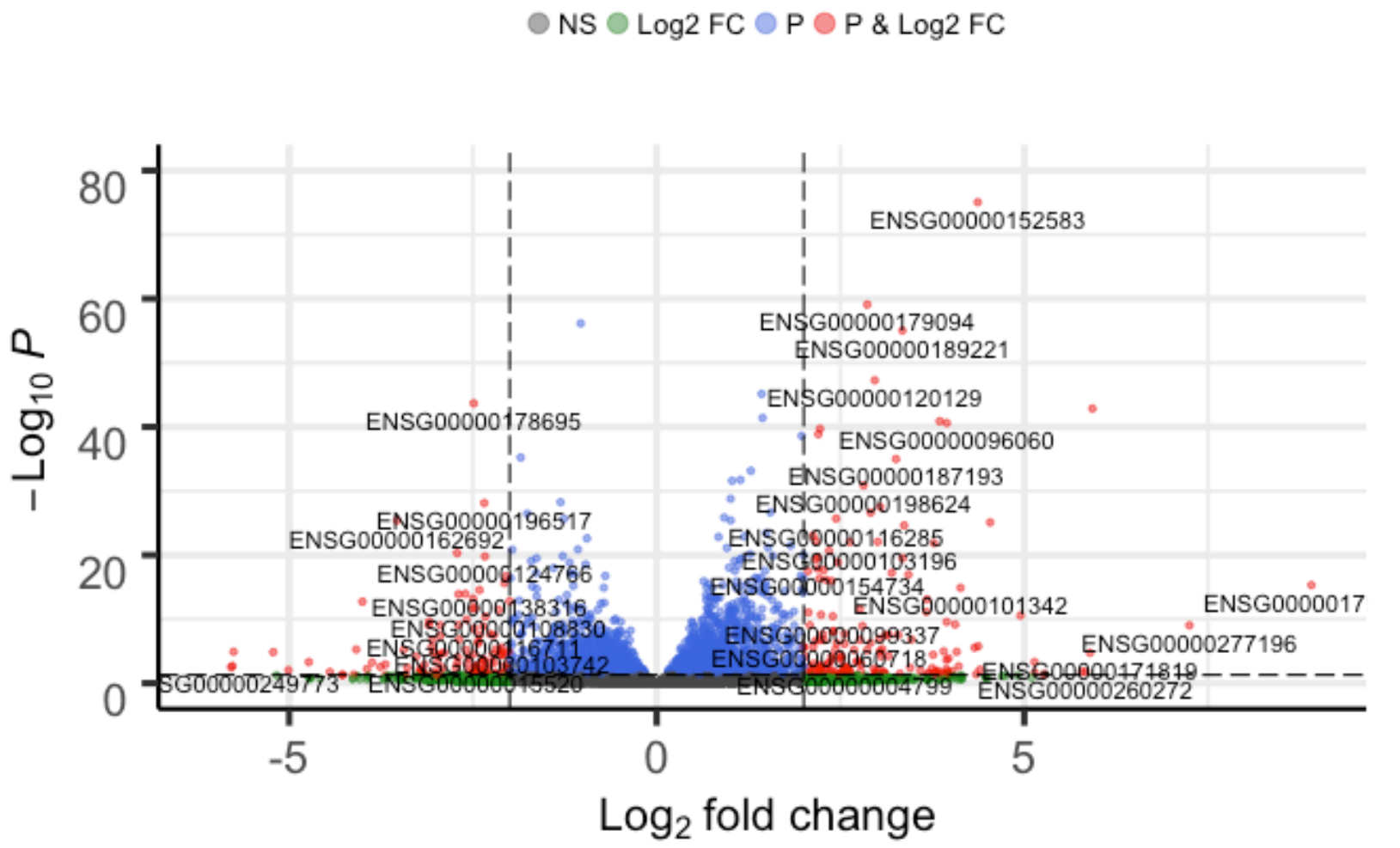
```
library(EnhancedVolcano)
```

Hide

Hide

```
EnhancedVolcano(res,
    lab = rownames(res),
    x = 'log2FoldChange',
    y = 'pvalue')
```

```
## Warning: Removed 13578 rows containing missing values (geom_point).
```



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

Session Information

The `sessionInfo()` function prints version information about R and any attached packages. It is 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.

Hide

Hide

```
sessionInfo()
```

```
## R version 3.5.1 (2018-07-02)
## Platform: x86_64-apple-darwin15.6.0 (64-bit)
## Running under: macOS High Sierra 10.13.6
##
## Matrix products: default
## BLAS: /Library/Frameworks/R.framework/Versions/3.5/Resources/lib/lib
Rblas.0.dylib
## LAPACK: /Library/Frameworks/R.framework/Versions/3.5/Resources/lib/l
ibRlapack.dylib
##
## locale:
## [1] en_US.UTF-8/en_US.UTF-8/en_US.UTF-8/C/en_US.UTF-8/en_US.UTF-8
##
## attached base packages:
## [1] parallel  stats4    stats     graphics  grDevices utils     data
sets
## [8] methods   base
##
## other attached packages:
##  [1] EnhancedVolcano_1.0.1       ggrepel_0.8.1
##  [3] ggplot2_3.1.1               org.Hs.eg.db_3.7.0
##  [5] AnnotationDbi_1.44.0        DESeq2_1.22.2
##  [7] SummarizedExperiment_1.12.0 DelayedArray_0.8.0
##  [9] BiocParallel_1.16.6         matrixStats_0.54.0
## [11] Biobase_2.42.0              GenomicRanges_1.34.0
## [13] GenomeInfoDb_1.18.2         IRanges_2.16.0
## [15] S4Vectors_0.20.1            BiocGenerics_0.28.0
## [17] BiocManager_1.30.4
##
## loaded via a namespace (and not attached):
##  [1] jsonlite_1.6           bit64_0.9-7            splines_3.5.1
##  [4] Formula_1.2-3          assertthat_0.2.1       latticeExtra_0.6-
28
##  [7] blob_1.1.1             GenomeInfoDbData_1.2.0 yaml_2.2.0
## [10] pillar_1.4.0           RSQLite_2.1.1          backports_1.1.4
## [13] lattice_0.20-38        glue_1.3.1             digest_0.6.18
## [16] RColorBrewer_1.1-2     XVector_0.22.0         checkmate_1.9.3
## [19] colorspace_1.4-1       htmltools_0.3.6        Matrix_1.2-17
## [22] plyr_1.8.4             XML_3.98-1.19          pkgconfig_2.0.2
## [25] genefilter_1.64.0      zlibbioc_1.28.0        purrr_0.3.2
## [28] xtable_1.8-4           scales_1.0.0           htmlTable_1.13.1
## [31] tibble_2.1.1           annotate_1.60.1        withr_2.1.2
```

```
## [34] nnet_7.3-12              lazyeval_0.2.2           survival_2.44-1.1
## [37] magrittr_1.5             crayon_1.3.4             memoise_1.1.0
## [40] evaluate_0.13            foreign_0.8-71           tools_3.5.1
## [43] data.table_1.12.2        stringr_1.4.0            locfit_1.5-9.1
## [46] munsell_0.5.0            cluster_2.0.9            compiler_3.5.1
## [49] rlang_0.3.4              grid_3.5.1               RCurl_1.95-4.12
## [52] rstudioapi_0.10          htmlwidgets_1.3          labeling_0.3
## [55] bitops_1.0-6             base64enc_0.1-3          rmarkdown_1.12
## [58] gtable_0.3.0             DBI_1.0.0                R6_2.4.0
## [61] gridExtra_2.3            knitr_1.22               dplyr_0.8.1
## [64] bit_1.1-14               Hmisc_4.2-0              stringi_1.4.3
## [67] Rcpp_1.0.1               geneplotter_1.60.0       rpart_4.1-15
## [70] acepack_1.4.1            tidyselect_0.2.5         xfun_0.7
```