A minimal quantitative RNAseq pipeline

Dan MacLean

2020-01-30

Contents

Chapter 1

About this course

In this short course we'll look at a method for getting quantitative estimates of gene expression from RNAseq data. The course assumes that you will already have performed a read alignment so is *not* a 'read to results' course. The course is very brief and will show you how to use a perform a common pipeline centered around `DESeq` in `R` and `RStudio`

I acknowledge that there are lots of other programs and methods - this course is *not* meant to be comprehensive, it is meant to get you being productive. Seek out further advice if you need to run other programs or systems. Do be encouraged though, lots of what you learn here will be applicable to other pipelines for the same job (they all run in a similar manner with similar objects) so this is a good place to start.

The course is intended to run on your 'local' machine, that is to say, your laptop or desktop computer. In general these machines will be powerful enough for most datasets though the pipeline we will learn can be easily adapted for a high performance computing environment if you need greater computational power.

1.1 Prerequisites

This course assumes that you are a little familiar with the basics of running R and R commands from the R console. You'll need to know the basics of typing in commands and getting output, not much more.

1.1.1 R and RStudio

1.1.1.1 Installing R

Follow this link and install the right version for your operating system https://www.stats.bris.ac.uk/R/

1.1.1.2 Installing RStudio

Follow this link and install the right version for your operating system https://www.rstudio.com/products/rstudio/download/

1.1.1.3 Installing R packages in RStudio.

You'll need the following R packages:

1. devtools
2. atacR
3. DESeq

For simplicity, install them in that order.

To install `devtools`:

Start RStudio and use the `Packages` tab in lower right panel. Click the install button (top left of the panel) and enter the package name `devtools`, then click install as in this picture

To install `atacR`:

Type the following into the RStudio console, `devtools::install_github("TeamMacLean/atacr")`

To install `DESeq`:

Type the following into the RStudio console, `BiocManager::install("DESeq")`

Now you are done! Everything is installed ready for you to work with. Next we need to get the sample data

1.1.2 Sample BAM file and counts files

You'll need this zip file of data: sample_data.zip which contains a lot of ifles including BAM files and counts. Download it, extract the files and put them into a folder on your machine. I suggest something like `Desktop/align_tut`. This will be the directory we'll work from in the rest of the course.

That's all you need to do the lesson. If you have any problems getting this going, then ask someone in the Bioinformatics Team and we'll help.

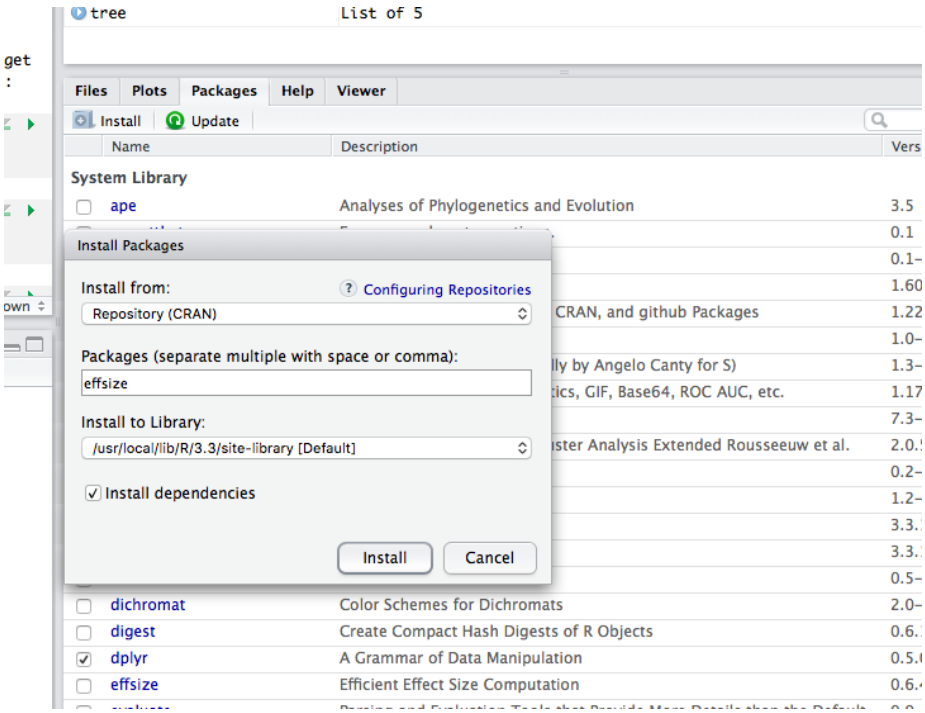


Figure 1.1: Installing Packages

Chapter 2

Counting Aligned Reads in Genomic Regions

2.1 About this chapter

1. Questions

- How do I calculate counts of reads at genes from my alignments?

2. Objectives

- Understand the basis for the gene region and read counting technique
- Understand what the count matrix represents
- Use the `make_counts()` function to make a count matrix

3. Keypoints

- Gene regions are designated by coordinates in GFF files
- A count matrix is a table-like object of reads that are found in a given genomic region
- The count matrix is the main object in a DESeq analysis

In this chapter we'll look at the fundamentals of read counting from a BAM file of aligned reads.

2.2 Counting the number of reads that have aligned to gene regions

The basis of quantitative RNAseq is working out how many of our sequence reads have aligned to each gene. In broad terms this is done by taking the

genomic coordinates of all the aligned reads (the start and end positions of the read's alignment on the reference genome) and cross-referencing them with the positions of the genes from a gene file. The resulting table is called a count matrix. See the figure below for a representation.

It is our aim in this section to create a count matrix from BAM files.

2.2.1 atacR

`atacR` was initially designed to help with the analysis of ATAC-Cap-seq data, a quite differen sort of data to RNAseq, but as with many bioinformatics pipelines, the first steps are quite common so we can make use of the neat way `atacR` handles the count matrix creation in the helpful function `make_counts()`

2.3 Preparing the input

We needs three things to work: the BAM files, a GFF file and a file of sample information.

2.3.1 The GFF file

GFF files are one way among many of describing the positions of genes on a genome. Here's a quick look at one.

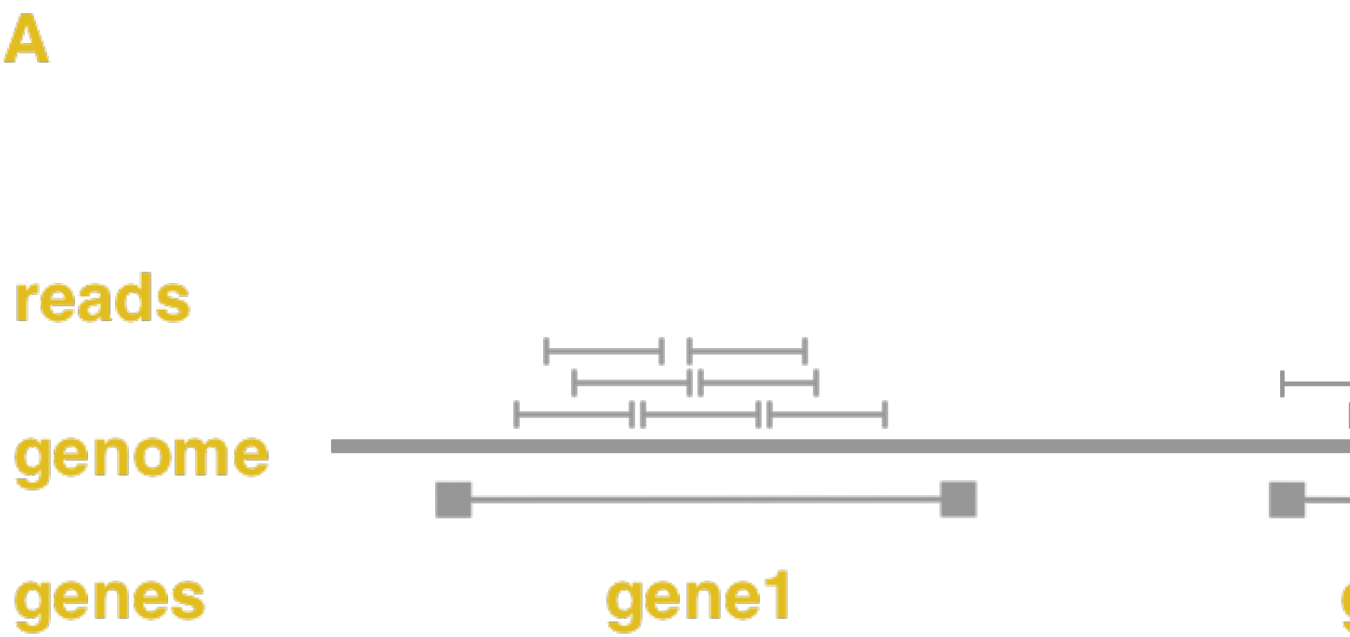
```
chr123  .  gene  1300  1500  .  +  .  ID=gene1
chr123  .  gene  1050  1500  .  +  .  ID=gene2
```

As you can see, it's a simple file with a gene represented on each line, by its chromosome (`chr123`), its start and end and its strand. The best thing about GFF files is that usually we can just download them from the relevant genome website. They tend to be freely available.

2.3.2 The Sample Information file

This file is a really simple file that references the BAM file of the alignment with the sample and replicate information. It has three columns: `sample_name`, `bam_file_path` and `treatment`. Here is an example.

```
## Parsed with column specification:
## cols(
##   treatment = col_character(),
##   sample_name = col_character(),
##   bam_file_path = col_character()
## )
```



B

	read count
gene1	7
gene2	4

Figure 2.1: A) Graphic of read alignment and gene position showing reads within genes. B) The equivalent count matrix that comes from this alignment

treatment	sample_name	bam_file_path
control	control_rep1	sample_data/control1/alignedSorted.bam
control	control_rep2	sample_data/control2/alignedSorted.bam
control	control_rep3	sample_data/control3/alignedSorted.bam
treatment	treatment_rep1	sample_data/treatment1/alignedSorted.bam
treatment	treatment_rep2	sample_data/treatment2/alignedSorted.bam
treatment	treatment_rep3	sample_data/treatment3/alignedSorted.bam

The `sample_name` column describes the treatment and replicate performed, the `bam_file_path` describes the place in which the BAM file for that sample is saved and `treatment` is the general name for the treatment that was used; this column is usually not unique when you have replicates.

2.3.3 The BAM files

The BAM files all come from a previously done alignment. The sample information file describes the place where they are kept and the sample they represent.

2.3.4 Sample files for this chapter

All the files are provided for you in the sample data you downloaded as `50_genes.gff` and `sample_information.csv` and in the folders containing BAM files. Feel free to examine them and look at how they relate to each other.

Once we have these files prepared, we can go on to use the `atacR` package to make the count matrix.

2.4 Running `make_counts()`

First we must load in `atacR`. Type the following into the R console.

```
library(atacr)
```

Now we can do the counting with `make_counts()`. Here's how to do it. Remember to properly describe the path to the files. The paths given here are correct if the files are in a folder called `sample_data` in the current working directory.

```
count_information <- make_counts("sample_data/50_genes.gff",
                                 "sample_data/sample_information.csv",
                                 is_rnaseq = TRUE
                                 )
```

The function should run and give no output. Note that it is important to set `is_rnaseq` to `TRUE` to tell the function to count appropriately. The results are saved in the `count_information` object.

2.5 Summaries and Diagnostic plots

With the counts computed we can do some diagnosis on the quality of the experiment.

We can see summary information with the `summary()` function

```
summary(count_information)
```

```
## ATAC-seq experiment of 2 treatments in 6 samples
##  Treatments: control,treatment
##  Samples: control_rep1,control_rep2,control_rep3,treatment_rep1,treatment_rep2,treatment_rep3
##  Bait regions used: 50
##  Total Windows: 99
##
##  On/Off target read counts:
##            sample off_target on_target percent_on_target
## 1   control_rep1          0     57733               100
## 2   control_rep2          0     66155               100
## 3   control_rep3          0     66122               100
## 4 treatment_rep1          0    100547               100
## 5 treatment_rep2          0    120325               100
## 6 treatment_rep3          0    107611               100
##  Quantiles:
##  $bait_windows
##     control_rep1 control_rep2 control_rep3 treatment_rep1 treatment_rep2
## 1%        149.48       294.60       241.12         228.70         102.98
## 5%        386.35       437.75       340.50         328.30         193.90
## 95%      2335.20      2438.20      2927.10        4445.90        6940.20
## 99%      3054.18      2752.19      3291.34        5234.33        9423.95
##     treatment_rep3
## 1%          116.50
## 5%          324.00
## 95%        4438.75
## 99%        6948.15
##
## $non_bait_windows
##     control_rep1 control_rep2 control_rep3 treatment_rep1 treatment_rep2
## 1%             0            0            0              0              0
## 5%             0            0            0              0              0
## 95%            0            0            0              0              0
```

```
## 99%            0            0            0              0              0
##     treatment_rep3
## 1%               0
## 5%               0
## 95%              0
## 99%              0
##
##  Read depths:
##             sample off_target on_target
## 1   control_rep1          0   1154.66
## 2   control_rep2          0   1323.10
## 3   control_rep3          0   1322.44
## 4 treatment_rep1          0   2010.94
## 5 treatment_rep2          0   2406.50
## 6 treatment_rep3          0   2152.22
```

It is long, but actually quite helpful. The first thing to note is that the words relate to ATAC-Cap-Seq, but in our context 'bait regions' just mean gene regions and non-bait just means intergenic regions. The 'on_targets' are read hits to genes, the 'off_targets' are read hits to intergenic regions.

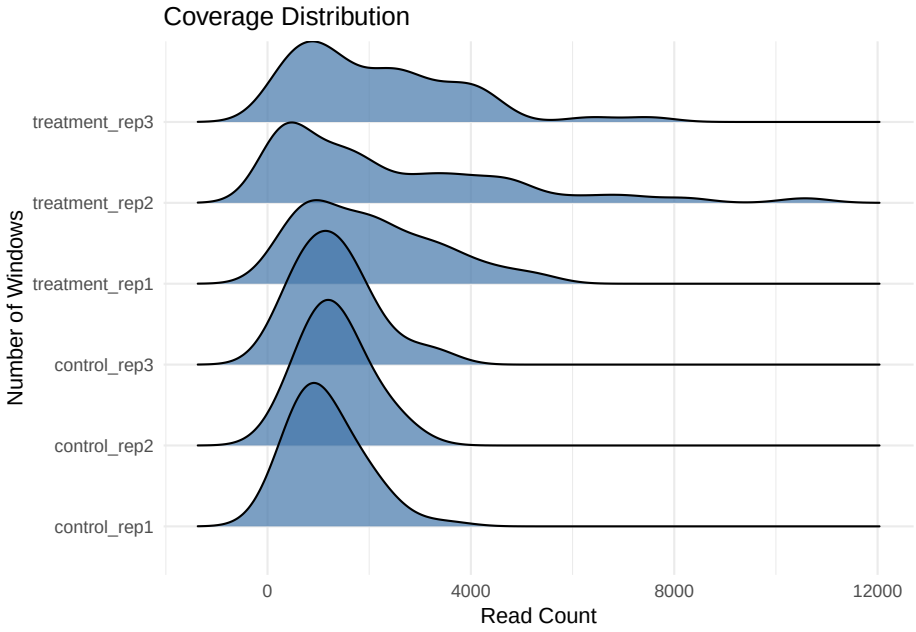
We can see that all the reads have hit in gene regions; that the read depth distribution of genes from the quantiles section give depths in the 1000 - 2000 range. This sort of summary is helpful when you're trying to work out whether the RNAseq is useful, lots of reads 'off target' is bad, as is low depth.

2.5.1 Gene Count Plots

We can see the distribution of depths over genes as a plot using the `plot_counts()` function

```
plot_counts(count_information, log10 = FALSE)
```

```
## Picking joint bandwidth of 488
```



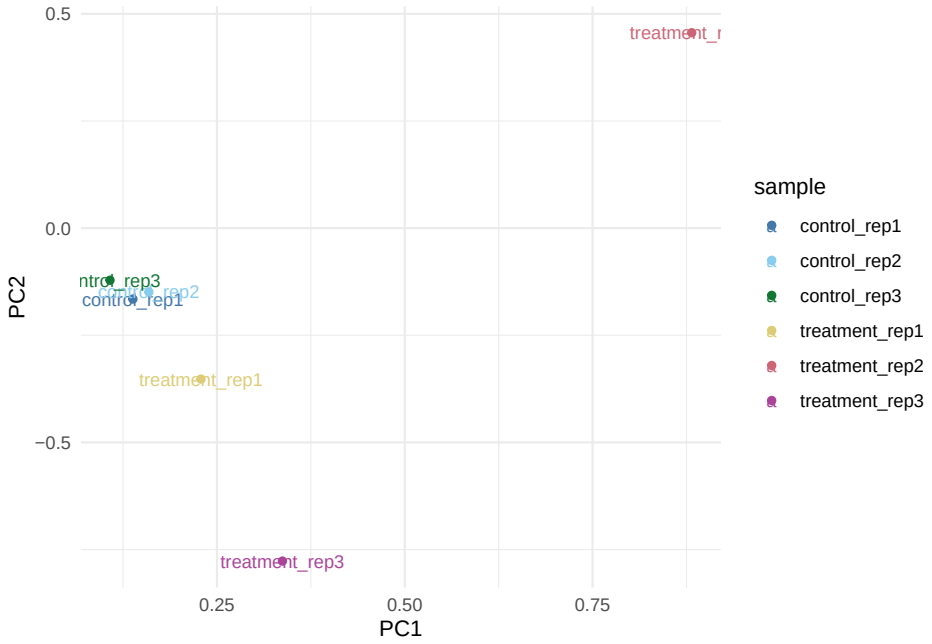
We can see that the mean count per gene (windows in `atacR`) is about 1000. The distributions in the treatment are bit more skewed than the controls.

2.5.2 Comparing Samples with PCA

It is common to examine the similarity of the samples to each other before moving on with analysis, ideally the similar samples will cluster together.

With `atacR` it is easy to perform a quick PCA analysis.

```
sample_pca_plot(count_information)
```



Here we can see that the control samples all cluster together, but the treatment samples are a bit more variable. We might want to normalise these counts later as a consequence.

2.6 Extracting and saving the count matrix

We now want to extract out the actual counts hiding inside the `count_information` object, we can do this with the `assay()` extractor function from the `SummarizedExperiment` package.

```
library(SummarizedExperiment)
raw_counts <- assay(count_information$bait_windows)

head(raw_counts)
```

```
##                          control_rep1 control_rep2 control_rep3
## Chr1:245989-249141                670          784          548
## Chr2:2195797-2200134             1104         1266          976
## Chr3:2454387-2458244              703          922          198
## Chr4:6650421-6657260             1865         1654         3207
## Chr5:11798344-11805414           1482         1266         1646
## Chr1:12893748-12901885           1186         1416         1458
##                          treatment_rep1 treatment_rep2 treatment_rep3
## Chr1:245989-249141                 1784           2558            368
```

```
## Chr2:2195797-2200134              358          1186          4436
## Chr3:2454387-2458244             1373          1167          1726
## Chr4:6650421-6657260             3533           703          2427
## Chr5:11798344-11805414           1258          1690          1864
## Chr1:12893748-12901885            834           594          2684
```

We can see the counts for each gene in each sample. Because `atacR` works on windows, the gene coordinates are given. We can replace the coordinates with gene names if we wish as follows

```
gene_names <- readr::read_csv("sample_data/gene_names.txt", col_names = FALSE )$X1
```

```
## Parsed with column specification:
## cols(
##   X1 = col_character()
## )
```

```
rownames(raw_counts) <- gene_names
head(raw_counts)
```

```
##           control_rep1 control_rep2 control_rep3 treatment_rep1
## AT1G01680          670          784          548           1784
## AT1G07160         1104         1266          976            358
## AT1G07920          703          922          198           1373
## AT1G19250         1865         1654         3207           3533
## AT1G32640         1482         1266         1646           1258
## AT1G35210         1186         1416         1458            834
##           treatment_rep2 treatment_rep3
## AT1G01680           2558            368
## AT1G07160           1186           4436
## AT1G07920           1167           1726
## AT1G19250            703           2427
## AT1G32640           1690           1864
## AT1G35210            594           2684
```

In this code chunk we load in the gene names from a file `gene_names.txt` using the `readr` package. Then we use the `rownames()` function to set the row names of `raw_counts`. This *is* a little cumbersome. Often you'll come across fiddly little things like this in bioinformatics analysis. If you ever get stuck feel free to come and chat to us in the bioinformatics team.

Now we can save the matrix to a file for re-use and importing into other programs. We'll do it in two ways 1) to a native R binary file that we can load straight in, 2) to a CSV file we can examine in programs including Excel.

2.6.1 Saving to an R RDS file

To save as a native R object, use `saveRDS()`, passing the filename you wish to save to.

```
saveRDS(raw_counts, "sample_data/raw_counts.RDS")
```

To save as a csv file use `write.table()`, again passing the filename you wish to save to.

```
write.csv( raw_counts, "sample_data/raw_counts.csv")
```

Now we can move on to using `DESeq`.

Chapter 3

Running DESeq2

3.1 About this chapter

1. Questions

- How do I work out which genes are differentially regulated?

2. Objectives

- Build a `DESeqDataSet` and `group` factor
- Run `DESeq`

3. Keypoints

- `DESeq2` is a package for estimating differential expression
- `DESeq2` needs you to describe the experiment in order to work

In this chapter we'll look at how to take our count matrix through `DESeq2` to estimate differential expression of genes.

3.2 Getting the count matrix and describing the experiment for DESeq2

3.2.1 The count matrix

The object we created in the previous chapter `raw_counts` is already in the format we need. If you carried straight into this chapter from the last one, then you already have what you need. If not, you can load in the saved version (there's a copy in the sample data) as follows

```
raw_counts <- readRDS("sample_data/raw_counts.RDS")
head(raw_counts)
```

```
##           control_rep1 control_rep2 control_rep3 treatment_rep1
## AT1G01680          670          784          548           1784
## AT1G07160         1104         1266          976            358
## AT1G07920          703          922          198           1373
## AT1G19250         1865         1654         3207           3533
## AT1G32640         1482         1266         1646           1258
## AT1G35210         1186         1416         1458            834
##           treatment_rep2 treatment_rep3
## AT1G01680           2558            368
## AT1G07160           1186           4436
## AT1G07920           1167           1726
## AT1G19250            703           2427
## AT1G32640           1690           1864
## AT1G35210            594           2684
```

3.3 The 'grouping' object

As R is a very powerful statistical programming language, it can support analysis of some very complicated experimental designs. `DESeq` supports this behaviour and as a result we have to describe our experiment in the appropriate manner.

We need to create a `data.frame` object that states which group each column is in. A `data.frame` is basically an R analogue of an Excel sheet. We just need to work out the right order of sample types in the matrix column.

Our experiment names are in the column names of the count matrix, we can see that with the `colnames()` function.

```
colnames(raw_counts)
```

```
## [1] "control_rep1"   "control_rep2"   "control_rep3"   "treatment_rep1"
## [5] "treatment_rep2" "treatment_rep3"
```

The controls are all in columns 1 to 3 and the treatments are in columns 4 to 6. To make the groupings we can just type in the sample types in the appropriate order and put them in a column of a `data.frame`. That looks like this

```
grouping <- data.frame(sample_type =  c("control", "control", "control", "treatment", "
grouping
```

```
##   sample_type
## 1     control
## 2     control
## 3     control
```

```
## 4   treatment
## 5   treatment
## 6   treatment
```

3.4 Running DESeq2

Now we have everything we need to run `DESeq2`. First, we must load in the library.

```
library(DESeq2)
```

Next, we can prepare the `DESeqDataSet` object that combines all the information `DESeq2` needs to work. We run `DESeqDataSetFromMatrix()` to do this.

```
dds <- DESeqDataSetFromMatrix(
  countData = raw_counts,
  colData = grouping,
  design = ~ sample_type)
```

Here we set the arguments

1. `countData` which is the actual data, so gets our `raw_counts`
2. `colData` which tells the group each data column is in so gets `grouping`
3. `design` is an R-ish way of describing the experiment design, for a standard exoeriment like this you use the `~` and the name of the `grouping` column

Don't worry too much about whether the `design` argument makes sense at this stage, its a bit out of scope to discuss the way R expects experimental designs for now. Follow the pattern you see here until you have a really complex design and have motivation to come back to it.

Finally, we can do the `DESeq` analysis. We have a single function for this and all it needs is our prepared data.

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

And now we can extract the results with the helpful `results()` function. This needs the `contrast` to be described, basically the column name and the types.

The types are ordered so that the first mentioned is the measurement of interest (ie the `treatment`) and the second is the baseline to which it is compared (here `control`). If you get the two the wrong way round, your up-regulated genes will look down-regulated and vice-versa, so take time to check.

```
results_data <- results(de_seq_analysed, contrast = c('sample_type', 'treatment', 'control'))
head(results_data)
```

```
## log2 fold change (MLE): sample_type treatment vs control
## Wald test p-value: sample type treatment vs control
```

```
## DataFrame with 6 rows and 6 columns
##                   baseMean     log2FoldChange             lfcSE
##                  <numeric>          <numeric>         <numeric>
## AT1G01680 1022.39953137642  0.638556501008842 0.752757958605879
## AT1G07160 1522.95786602724  0.354926012092298 0.785708629982376
## AT1G07920 946.524794250111  0.671894996505363 0.742814441576441
## AT1G19250 2230.10488838027 -0.591880977166901 0.603665159459885
## AT1G32640 1539.40321179978 -0.420258753438091 0.537635950705001
## AT1G35210 1384.96737134404 -0.485617346461465 0.690715741846472
##                       stat            pvalue              padj
##                  <numeric>         <numeric>         <numeric>
## AT1G01680  0.848289272412955 0.396276890458589 0.747206163566895
## AT1G07160   0.45172726701533 0.651465472435186 0.796576241413894
## AT1G07920  0.904526028168531 0.365716539229272 0.747206163566895
## AT1G19250 -0.980478942492676 0.326849759032812 0.747206163566895
## AT1G32640 -0.781679039296026 0.434403223096372 0.755158226458792
## AT1G35210 -0.703063962554663 0.482015889229017 0.755158226458792
```

We get a lot of information back in this column. We can see in amongst all that the important log fold change estimates and the adjusted p-value. Effectively, our analysis is done, we have our differential expression estimates, though we do need to do more to answer questions of interest. That's what we'll do in the next chapter

3.5 Saving the results

As a final step, we can save the results to a CSV file. As in the earlier chapter we can do this with `write.csv()`

```
write.csv(results_data, "sample_data/results.csv")
```

Chapter 4

Next Steps

4.1 About this chapter

1. Questions

- How can I filter out the 'significant' genes?
- How can I find the functions of these genes?

2. Objectives

- Filter the genes by p value
- Find the gene annotations on an external service

3. Keypoints

- The p value we need must be corrected for the large number of genes

In this chapter we'll look at how to take our results table and get the significantly differentially expressed genes out of it.

4.2 The results data frame

The object we created in the previous chapter `results_data` is already in the general format we need. Im going to load a version with different gene names (from Magnaporthe, not Arabidopsis) so we can work on a more familiar genome.

```
results_data <- read.csv("sample_data/results_mo.csv")
head(results_data)
```

```
##        gene  baseMean log2FoldChange     lfcSE       stat    pvalue
## 1 MGG_00865 1022.3995      0.6385565 0.7527580  0.8482893 0.3962769
## 2 MGG_08134 1522.9579      0.3549260 0.7857086  0.4517273 0.6514655
```

```
## 3 MGG_01588  946.5248      0.6718950 0.7428144  0.9045260 0.3657165
## 4 MGG_13806 2230.1049     -0.5918810 0.6036652 -0.9804789 0.3268498
## 5 MGG_06121 1539.4032     -0.4202588 0.5376360 -0.7816790 0.4344032
## 6 MGG_06504 1384.9674     -0.4856173 0.6907157 -0.7030640 0.4820159
##        padj
## 1 0.7472062
## 2 0.7965762
## 3 0.7472062
## 4 0.7472062
## 5 0.7551582
## 6 0.7551582
```

4.2.1 Which *p* value?

Note that two columns in this data.frame have p value information in them - `pvalue` and `padj`. Which is the correct one? We need the adjusted p value in `padj`.

The reason for this is that a separate p value was calculated in a separate test for each gene. As p values have a built in error expectation (IE $p = 0.05$ means the test will be wrong 5 percent of the time on average) then repeating the test means that using a gene level p value means we get lots of errors. So we need a whole gene set p value rather than a per gene value. Hence `DESeq` adjusts the p value for the whole set of genes.

4.3 Filtering rows with significant *p* values

To filter the rows in the data frame we can use `tidyverse` tools like `dplyr` (which we study in a separate training course). Let's keep rows from the `results_data` data frame with a `padj` lower than `0.05`

```
library(dplyr)
significant_genes <- filter(results_data, padj < 0.05)
significant_genes
```

```
##        gene  baseMean log2FoldChange     lfcSE     stat pvalue  padj
## 1 MGG_12738  552.4170     -1.2700540 1.1960414 1.061881  0.005 0.030
## 2 MGG_01482 1912.4502      1.1010845 0.8063151 1.365576  0.005 0.004
## 3 MGG_17878  988.8704      0.9095213 0.9142498 0.994828  0.002 0.010
```

The `filter()` function works by taking the dataframe and the condition and column to filter with.

4.3.1 Filtering UP and DOWN genes

An elaboration of this is to find the up or down genes. To do this we need to build a filter on the `log2FoldChange` column. As the fold changes are encoded in a log scale, up regulated genes will have a positive value, down regulated genes will have a negative value

```
up_genes <- filter(results_data, padj < 0.05, log2FoldChange > 0)
up_genes
```

```
##        gene  baseMean log2FoldChange     lfcSE     stat pvalue  padj
## 1 MGG_01482 1912.4502      1.1010845 0.8063151 1.365576  0.005 0.004
## 2 MGG_17878  988.8704      0.9095213 0.9142498 0.994828  0.002 0.010
```

```
down_genes <- filter(results_data, padj < 0.05, log2FoldChange < 0)
down_genes
```

```
##        gene baseMean log2FoldChange    lfcSE     stat pvalue padj
## 1 MGG_12738  552.417      -1.270054 1.196041 1.061881  0.005 0.03
```

Note that if you want to find values that are two fold up or down regulated, then you'll need to change the `log2FoldChange` values to `1` and `-1` (as `log2(2) = 1` and `log2(0.5) = -1`).

You can export each of these tables to files with `write.csv()` as previously.

4.4 Finding gene annotations

A common question is 'which pathways and functional categories do my genes belong to?'. Answering this requires quite an involved process, and doing it entirely in R is out of scope for this 'minimal' RNAseq tutorial. Instead of avoiding the question completely, we'll look at how to achieve a basic annotation using webtools. Specifically, the Ensembl BioMart service.

4.4.1 BioMart

BioMart is a data warehouse for genomic information that can be queried through a web interface. Not all genome projects provide such a service, but the ones on Ensembl generally do. We'll work with the Magnaporthe one here, available at https://fungi.ensembl.org/Magnaporthe_oryzae/Info/Index

To access it we need to click `BioMart` from the top menu, and be patient, it can take a little while to load.

Then we need to follow this procedure to get a gene list annotated

1. From the `Choose Database` drop-down select `Ensembl Fungi Genes`

2. From the `Choose Dataset` drop down select `Magnaporthe oryzae genes (MG8)`
3. Select the `Attributes` page and on the `External` tab tick `GO Term Name`, `GO Term Definition` and `KEGG Pathway and Enzyme ID`. These attributes are the things you will retrieve from the BioMart.
4. Select the `Filters` page and click on the `External` tab, paste in the gene IDs of interest into the box or upload a file.
5. Click `Results` button at the top

After a wait, the screen should fill with the annotations that you asked for. You can save this to a file using the `Export` options at the top.

This is all you need to make an annotated gene list.

4.5 Further Questions

Of course, this isn't all you might want to do with your RNAseq data and gene lists. We've achieved our overall goal of getting a minimal RNAseq analysis done. What happens next will be quite different for every experiment. For example, you might want to look at seeing whether a GO Term or enzymatic pathway is enriched. Pretty much everything will be a separate analysis in itself and will require some design and planning. Please feel free to talk to the bioinformatics team when you find yourself at this stage, we'll be extremely happy to work with you!