Code

Transcriptome Data Analysis - hands-on session

MSE - Module Genetic Epidemiology

(https://www.unimedizin-mainz.de/imbei/startseite/mse/)

Federico Marini ((https://federicomarini.github.io)marinif@uni-mainz.de (mailto:marinif@uni-mainz.de))
IMBEI, University Medical Center Mainz (https://www.unimedizin-mainz.de/imbei/)
`@FedeBioinfo` (https://twitter.com/FedeBioinfo)

Alicia Schulze (alpoplaw@uni-mainz.de (mailto:alpoplaw@uni-mainz.de))
IMBEI, University Medical Center Mainz (https://www.unimedizin-mainz.de/imbei/)

2023/07/05-06

1 Introduction

In this tutorial we walk through a gene-level RNA-seq differential expression analysis using Bioconductor packages. We start from the gene-vs-sample count matrix, and thus assume that the raw reads have already been quality controlled and that the gene expression has been quantified (either using alignment and counting, or by applying an alignment-free quantification tool). We perform exploratory data analysis (EDA) for quality assessment and to explore the relationship between samples, then perform differential gene expression analysis, and visually explore the results.

Bioconductor has many packages supporting analysis of high-throughput sequence data, including RNA-seq. The packages that we will use in this tutorial include core packages maintained by the Bioconductor core team (https://www.bioconductor.org/about/core-team/) for importing and processing raw sequencing data and loading gene annotations. We will also use contributed packages for statistical analysis and visualization of sequencing data. Through scheduled releases every 6 months, the Bioconductor project ensures that all the packages within a release will work together in harmony (hence the “conductor” metaphor). The packages used in this tutorial are loaded with the *library* function and can be installed by following the Bioconductor package installation instructions (http://bioconductor.org/install/#install-bioconductor-packages). If you use the results from an R package in published research, you can find the proper citation for the software by typing `citation("pkgName")`, where you would substitute the name of the package for `pkgName`. Citing methods papers helps to support and reward the individuals who put time into open source software for genomic data analysis.

Many parts of this tutorial are based on a published RNA-seq workflow available via F1000Research (http://f1000research.com/articles/4-1070) (Love et al. 2015) and as a Bioconductor package (https://www.bioconductor.org/packages/release/workflows/html/rnaseqGene.html).

Along this notebook that you can render yourself - we will show you how during the workshop - you will encounter some small quiz sections to keep up with your progress throughout the entire workflow.

1.1 Setup recommendation

We recommend you use the following setup for an effective learning experience

- Clone or download the workshop repo, available at https://github.com/imbeimainz/RNAseqFrankfurt2023 (https://github.com/imbeimainz/RNAseqFrankfurt2023)
- (Make sure you have R/RStudio/Bioconductor all setup, as expected)

- Work in RStudio, ideally clicking on the RStudio project file
- Open up the `RNAseq_practical.Rmd` file, inside RStudio
- Execute the code chunks/read the information provided

Have fun learning - together!

1.2 Experimental data

The data used in this workflow comes from an RNA-seq experiment (Alasoo et al. 2018), in which the authors identified shared quantitative trait loci (QTLs) for chromatin accessibility and gene expression in human macrophages exposed to IFNgamma, Salmonella and IFNgamma plus Salmonella. Processed data from a subset of 24 samples from this experiment (six female donors, with four treatments each) is available via the *macrophage (https://bioconductor.org/packages/3.17/macrophage)* R/Bioconductor package. In particular, the package contains output from *Salmon* (Patro et al. 2017), as well as a metadata file. More information about how the raw data was processed is available from the package vignette (http://bioconductor.org/packages/release/data/experiment/vignettes/macrophage/inst/doc/macrophage.html).

We start by setting the path to the folder containing the quantifications (the output folders from *Salmon*). Since these are provided with an R package, we will point to the `extdata` subfolder of the installed package. For a typical analysis of your own data, you would point directly to a folder on your storage system (i.e., not using `system.file()`).

Hide

```
library(macrophage)
dir <- system.file("extdata", package = "macrophage")
list.files(dir)
# [1] "coldata.csv"                   "errs"
# [3] "gencode.v29_salmon_0.12.0"     "gencode.v29.annotation.gtf.gz"
# [5] "PRJEB18997.txt"                "quants"
# [7] "supp_table_1.csv"              "supp_table_7.csv"
```

TODO: open up that directory to “better see what is in it”? Idea: it can give them a sense for the *what do I expect to get there?*

2 Reading the metadata

First, we will read the metadata for the experiment. The main annotations of interest for this tutorial are `condition_name` , which represents the treatment of the sample (naive, IFN gamma, Salmonella, IFN gamma+Salmonella) and `line_id` , which represents the donor ID. The sample identifier is given by the `names` column, and will be used to match the metadata table to the quantifications.

Hide

```
coldata <- read.csv(file.path(dir, "coldata.csv"))[, c(1, 2, 3, 5)]
dim(coldata)
# [1] 24  4
coldata
#             names sample_id line_id condition_name
# 1  SAMEA103885102    diku_A  diku_1          naive
# 2  SAMEA103885347    diku_B  diku_1           IFNg
# 3  SAMEA103885043    diku_C  diku_1         SL1344
# 4  SAMEA103885392    diku_D  diku_1    IFNg_SL1344
# 5  SAMEA103885182    eiwy_A  eiwy_1          naive
# 6  SAMEA103885136    eiwy_B  eiwy_1           IFNg
# 7  SAMEA103885413    eiwy_C  eiwy_1         SL1344
# 8  SAMEA103884967    eiwy_D  eiwy_1    IFNg_SL1344
# 9  SAMEA103885368    fikt_A  fikt_3          naive
# 10 SAMEA103885218    fikt_B  fikt_3           IFNg
# 11 SAMEA103885319    fikt_C  fikt_3         SL1344
# 12 SAMEA103885004    fikt_D  fikt_3    IFNg_SL1344
# 13 SAMEA103885284    ieki_A  ieki_2          naive
# 14 SAMEA103885059    ieki_B  ieki_2           IFNg
# 15 SAMEA103884898    ieki_C  ieki_2         SL1344
# 16 SAMEA103885157    ieki_D  ieki_2    IFNg_SL1344
# 17 SAMEA103885111    podx_A  podx_1          naive
# 18 SAMEA103884919    podx_B  podx_1           IFNg
# 19 SAMEA103885276    podx_C  podx_1         SL1344
# 20 SAMEA103885021    podx_D  podx_1    IFNg_SL1344
# 21 SAMEA103885262    qaqx_A  qaqx_1          naive
# 22 SAMEA103885228    qaqx_B  qaqx_1           IFNg
# 23 SAMEA103885308    qaqx_C  qaqx_1         SL1344
# 24 SAMEA103884949    qaqx_D  qaqx_1    IFNg_SL1344
```

In addition to the `names` column, *tximeta (https://bioconductor.org/packages/3.17/tximeta)*, which we will use to read the quantification data, requires that `coldata` has a column named `files` , pointing to the *Salmon* output (the `quant.sf` file) for the respective samples.

Hide

```
coldata$files <- file.path(dir, "quants", coldata$names, "quant.sf.gz")
head(coldata)
#            names sample_id line_id condition_name
# 1 SAMEA103885102    diku_A  diku_1          naive
# 2 SAMEA103885347    diku_B  diku_1           IFNg
# 3 SAMEA103885043    diku_C  diku_1         SL1344
# 4 SAMEA103885392    diku_D  diku_1    IFNg_SL1344
# 5 SAMEA103885182    eiwy_A  eiwy_1          naive
# 6 SAMEA103885136    eiwy_B  eiwy_1           IFNg
#
files
# 1 /Library/Frameworks/R.framework/Versions/4.3-x86_64/Resources/library/macrophage/extdata/quants/SAMEA103885102/quant.
sf.gz
# 2 /Library/Frameworks/R.framework/Versions/4.3-x86_64/Resources/library/macrophage/extdata/quants/SAMEA103885347/quant.
sf.gz
# 3 /Library/Frameworks/R.framework/Versions/4.3-x86_64/Resources/library/macrophage/extdata/quants/SAMEA103885043/quant.
sf.gz
# 4 /Library/Frameworks/R.framework/Versions/4.3-x86_64/Resources/library/macrophage/extdata/quants/SAMEA103885392/quant.
sf.gz
# 5 /Library/Frameworks/R.framework/Versions/4.3-x86_64/Resources/library/macrophage/extdata/quants/SAMEA103885182/quant.
sf.gz
# 6 /Library/Frameworks/R.framework/Versions/4.3-x86_64/Resources/library/macrophage/extdata/quants/SAMEA103885136/quant.
sf.gz
all(file.exists(coldata$files))
# [1] TRUE
```

Now we have everything we need, and can import the quantifications with *tximeta*. In this process, we will see that *tximeta* automatically identifies the source and version of the transcriptome reference that was used for the quantification, and adds some metadata. The imported data will be stored in a *SummarizedExperiment* container.

3 Importing quantifications into R

We will next read the *Salmon* quantifications provided in the *macrophage (https://bioconductor.org/packages/3.17/macrophage)* package into R and summarize the expected counts on the gene level. A simple way to import results from a variety of transcript abundance estimation tools into R is provided by the *tximport (https://bioconductor.org/packages/3.17/tximport)* and *tximeta (https://bioconductor.org/packages/3.17/tximeta)* packages. Here, *tximport* reads the quantifications into a list of matrices, while *tximeta* instead aggregates the information into a *SummarizedExperiment* object, and also automatically adds additional annotations for the features. Both packages can return quantifications on the transcript level or aggregate them on the gene level. They also calculate average transcript lengths for each gene and each sample, which can be used as offsets to improve the differential expression analysis by accounting for differential isoform usage across samples (Soneson, Love, and Robinson 2015).

The code below imports the *Salmon* quantifications into R using the *tximeta* package. Note how the transcriptome that was used for the quantification is automatically recognized and used to annotate the resulting data object. In order for this to work, *tximeta* requires that the output folder structure from Salmon is retained, since it reads information from the associated log files in addition to the quantified abundances themselves.

Hide

```
suppressPackageStartupMessages({
    library(tximeta)
    library(DESeq2)
    library(org.Hs.eg.db)
    library(SummarizedExperiment)
})

## Import quantifications on the transcript level
st <- tximeta(coldata = coldata, type = "salmon", dropInfReps = TRUE)

st
# class: RangedSummarizedExperiment
# dim: 205870 24
# metadata(6): tximetaInfo quantInfo ... txomeInfo txdbInfo
# assays(3): counts abundance length
# rownames(205870): ENST00000456328.2 ENST00000450305.2 ...
#   ENST00000387460.2 ENST00000387461.2
# rowData names(3): tx_id gene_id tx_name
# colnames(24): SAMEA103885102 SAMEA103885347 ... SAMEA103885308
#   SAMEA103884949
# colData names(4): names sample_id line_id condition_name
```

We see that *tximeta* has identified the transcriptome used for the quantification as `GENCODE - Homo sapiens - release 29`. How did this happen? In fact, the output directory from *Salmon* contains much more information than just the `quant.sf` file! (as mentioned above, this means that it is not advisable to move files out of the folder, or to share only the `quant.sf` file, since the context is lost):

Hide

```
list.files(file.path(dir, "quants", coldata$names[1]), recursive = TRUE)
#  [1] "aux_info/ambig_info.tsv.gz"       "aux_info/bootstrap/bootstraps.gz"
#  [3] "aux_info/bootstrap/names.tsv.gz"  "aux_info/exp_gc.gz"
#  [5] "aux_info/expected_bias.gz"        "aux_info/fld.gz"
#  [7] "aux_info/meta_info.json"          "aux_info/obs_gc.gz"
#  [9] "aux_info/observed_bias_3p.gz"     "aux_info/observed_bias.gz"
# [11] "cmd_info.json"                    "lib_format_counts.json"
# [13] "libParams/flenDist.txt"           "logs/salmon_quant.log.txt"
# [15] "quant.sf.gz"
```

In particular, the `meta_info.json` file contains a hash checksum, which is derived from the set of transcripts used as reference during the quantification and which lets *tximeta* identify the reference source (by comparing to a table of these hash checksums for commonly used references).

Hide

```
rjson::fromJSON(file = file.path(dir, "quants", coldata$names[1],
                                 "aux_info", "meta_info.json"))
# $salmon_version
# [1] "0.12.0"
#
# $samp_type
# [1] "gibbs"
#
# $opt_type
# [1] "vb"
#
# $quant_errors
# list()
#
# $num_libraries
# [1] 1
#
# $library_types
# [1] "ISR"
#
# $frag_dist_length
# [1] 1001
#
# $seq_bias_correct
# [1] FALSE
#
# $gc_bias_correct
# [1] TRUE
#
# $num_bias_bins
# [1] 4096
#
# $mapping_type
# [1] "mapping"
#
# $num_targets
# [1] 205870
#
# $serialized_eq_classes
# [1] FALSE
#
# $eq_class_properties
# list()
#
# $length_classes
# [1]    520    669   1065   2328 205012
#
# $index_seq_hash
# [1] "40849ed828ea7d6a94af54a5c40e5d87eb0ce0fc1e9513208a5cffe59d442292"
#
# $index_name_hash
# [1] "77aca5545a0626421efb4730dd7b95482c77da261f9bdef70d36e25ee68bb7ef"
#
# $index_seq_hash512
# [1] "f37ae2a7412efd8518d68c22fc3fcc2478b59833809382abd5d9055475505e516730c70914343af34c9232af92fe22832e70afde6b38c26381
097068e1e82551"
#
# $index_name_hash512
# [1] "67f286e5ec2895c10aa6e3a8feed04cb3a80747de7b3afb620298078481b303d2f979407fc104d4f3e8af0e82be2acb2f294ee5f2f68c00087
f2855120d9f4ed"
#
# $num_bootstraps
# [1] 20
#
# $num_processed
# [1] 45141218
#
# $num_mapped
# [1] 40075160
#
# $percent_mapped
# [1] 88.77731
#
```

```
# $call
# [1] "quant"
#
# $start_time
# [1] "Tue Jan 15 21:21:22 2019"
#
# $end_time
# [1] "Tue Jan 15 21:29:00 2019"
```

Looking at the size of the *SummarizedExperiment* object (205,870 rows!) as well as the row names, we see that this object contains transcript-level information. The assays are created by directly importing the values from the `quant.sf` files and combining this information across the 24 samples:

- counts - `NumReads` column
- abundance - `TPM` column
- length - `EffectiveLength` column

We can access any of the assays via the `assay` function:

Hide

```
head(assay(st, "counts"), 3)
#                   SAMEA103885102 SAMEA103885347 SAMEA103885043 SAMEA103885392
# ENST00000456328.2         22.058         12.404           5.44          0.000
# ENST00000450305.2          0.000          0.000           0.00          0.000
# ENST00000488147.1        119.092        180.069         161.55         93.747
#                   SAMEA103885182 SAMEA103885136 SAMEA103885413 SAMEA103884967
# ENST00000456328.2            0.0         10.833          5.119          6.708
# ENST00000450305.2            0.0          0.000          0.000          0.000
# ENST00000488147.1          145.5        141.607        189.152         98.019
#                   SAMEA103885368 SAMEA103885218 SAMEA103885319 SAMEA103885004
# ENST00000456328.2          0.000          4.484          0.000          0.000
# ENST00000450305.2          0.000          0.000          0.000          0.000
# ENST00000488147.1        132.243         88.429         96.871         66.813
#                   SAMEA103885284 SAMEA103885059 SAMEA103884898 SAMEA103885157
# ENST00000456328.2         27.337         12.401          0.000          6.107
# ENST00000450305.2          0.000          0.000          0.000          0.000
# ENST00000488147.1        250.127        211.475        144.212        134.842
#                   SAMEA103885111 SAMEA103884919 SAMEA103885276 SAMEA103885021
# ENST00000456328.2         23.333         11.794          4.670          0.000
# ENST00000450305.2          0.000          0.000          0.000          0.000
# ENST00000488147.1        205.167        151.599        134.082         69.402
#                   SAMEA103885262 SAMEA103885228 SAMEA103885308 SAMEA103884949
# ENST00000456328.2          7.629          3.907          9.195          0.000
# ENST00000450305.2          0.000          0.000          0.000          0.000
# ENST00000488147.1        154.885        125.882        183.322         53.233
```

You may have noted that `st` is in fact a *RangedSummarizedExperiment* object (rather than "just" a *SummarizedExperiment* object). What does this mean? Let's look at the information we have about the rows (transcripts) in the object:

Hide

```
rowRanges(st)
# GRanges object with 205870 ranges and 3 metadata columns:
#                     seqnames      ranges strand |     tx_id           gene_id
#                        <Rle>   <IRanges>  <Rle> | <integer>   <CharacterList>
#   ENST00000456328.2     chr1 11869-14409      + |         1 ENSG00000223972.5
#   ENST00000450305.2     chr1 12010-13670      + |         2 ENSG00000223972.5
#   ENST00000488147.1     chr1 14404-29570      - |      9483 ENSG00000227232.5
#   ENST00000619216.1     chr1 17369-17436      - |      9484 ENSG00000278267.1
#   ENST00000473358.1     chr1 29554-31097      + |         3 ENSG00000243485.5
#                  ...      ...         ...    ... .       ...               ...
#   ENST00000361681.2     chrM 14149-14673      - |    206692 ENSG00000198695.2
#   ENST00000387459.1     chrM 14674-14742      - |    206693 ENSG00000210194.1
#   ENST00000361789.2     chrM 14747-15887      + |    206684 ENSG00000198727.2
#   ENST00000387460.2     chrM 15888-15953      + |    206685 ENSG00000210195.2
#   ENST00000387461.2     chrM 15956-16023      - |    206694 ENSG00000210196.2
#                           tx_name
#                         <character>
#   ENST00000456328.2 ENST00000456328.2
#   ENST00000450305.2 ENST00000450305.2
#   ENST00000488147.1 ENST00000488147.1
#   ENST00000619216.1 ENST00000619216.1
#   ENST00000473358.1 ENST00000473358.1
#                 ...               ...
#   ENST00000361681.2 ENST00000361681.2
#   ENST00000387459.1 ENST00000387459.1
#   ENST00000361789.2 ENST00000361789.2
#   ENST00000387460.2 ENST00000387460.2
#   ENST00000387461.2 ENST00000387461.2
#   -------
#   seqinfo: 25 sequences (1 circular) from hg38 genome
```

By knowing the source and version of the reference used for the quantification, *tximeta* was able to retrieve the annotation files and decorate the object with information about the transcripts, such as the chromosome and position, and the corresponding gene ID. Importantly, *Salmon* did not use (or know about) any of this during the quantification! It needs only the transcript sequences. If we just want the annotation columns, without the ranges, we can get those with the `rowData` accessor:

Hide

```
rowData(st)
# DataFrame with 205870 rows and 3 columns
#                       tx_id           gene_id           tx_name
#                   <integer>   <CharacterList>       <character>
# ENST00000456328.2         1 ENSG00000223972.5 ENST00000456328.2
# ENST00000450305.2         2 ENSG00000223972.5 ENST00000450305.2
# ENST00000488147.1      9483 ENSG00000227232.5 ENST00000488147.1
# ENST00000619216.1      9484 ENSG00000278267.1 ENST00000619216.1
# ENST00000473358.1         3 ENSG00000243485.5 ENST00000473358.1
# ...                     ...               ...               ...
# ENST00000361681.2    206692 ENSG00000198695.2 ENST00000361681.2
# ENST00000387459.1    206693 ENSG00000210194.1 ENST00000387459.1
# ENST00000361789.2    206684 ENSG00000198727.2 ENST00000361789.2
# ENST00000387460.2    206685 ENSG00000210195.2 ENST00000387460.2
# ENST00000387461.2    206694 ENSG00000210196.2 ENST00000387461.2
```

Similar to the row annotations in `rowData`, the *SummarizedExperiment* object contains sample annotations in the `colData` slot.

Hide

```
colData(st)
# DataFrame with 24 rows and 4 columns
#                         names   sample_id     line_id condition_name
#                   <character> <character> <character>    <character>
# SAMEA103885102 SAMEA103885102      diku_A      diku_1          naive
# SAMEA103885347 SAMEA103885347      diku_B      diku_1           IFNg
# SAMEA103885043 SAMEA103885043      diku_C      diku_1         SL1344
# SAMEA103885392 SAMEA103885392      diku_D      diku_1    IFNg_SL1344
# SAMEA103885182 SAMEA103885182      eiwy_A      eiwy_1          naive
# ...                       ...         ...         ...            ...
# SAMEA103885021 SAMEA103885021      podx_D      podx_1    IFNg_SL1344
# SAMEA103885262 SAMEA103885262      qaqx_A      qaqx_1          naive
# SAMEA103885228 SAMEA103885228      qaqx_B      qaqx_1           IFNg
# SAMEA103885308 SAMEA103885308      qaqx_C      qaqx_1         SL1344
# SAMEA103884949 SAMEA103884949      qaqx_D      qaqx_1    IFNg_SL1344
```

Quiz

- How many samples are there in this `macrophage` dataset? How many are there in your (typical) RNA-seq experiment?
- Is the quantification done at the transcript level or at the gene level? How can you go from one to the other? And vice versa?
- What is "better"? Think of DGE, DTU, DTE… and of your experimental setting

4 Summarizing on the gene level

As we saw, the features in the *SummarizedExperiment* object above are individual transcripts, rather than genes. Often, however, we want to do analysis on the gene level, since the gene-level abundances are more robust and sometimes more interpretable than transcript-level abundances. The `rowData` contains the information about the corresponding gene for each transcript, in the `gene_id` column, and *tximeta* provides a function to summarize on the gene level:

- Counts are added up
- TPMs are added up
- Transcript lengths are added up after weighting by the respective transcript TPMs

Hide

```
## Summarize quantifications on the gene level
sg <- tximeta::summarizeToGene(st)
sg
# class: RangedSummarizedExperiment
# dim: 58294 24
# metadata(6): tximetaInfo quantInfo ... txomeInfo txdbInfo
# assays(3): counts abundance length
# rownames(58294): ENSG00000000003.14 ENSG00000000005.5 ...
#   ENSG00000285993.1 ENSG00000285994.1
# rowData names(2): gene_id tx_ids
# colnames(24): SAMEA103885102 SAMEA103885347 ... SAMEA103885308
#   SAMEA103884949
# colData names(4): names sample_id line_id condition_name

# compare e.g. to
st
# class: RangedSummarizedExperiment
# dim: 205870 24
# metadata(6): tximetaInfo quantInfo ... txomeInfo txdbInfo
# assays(3): counts abundance length
# rownames(205870): ENST00000456328.2 ENST00000450305.2 ...
#   ENST00000387460.2 ENST00000387461.2
# rowData names(3): tx_id gene_id tx_name
# colnames(24): SAMEA103885102 SAMEA103885347 ... SAMEA103885308
#   SAMEA103884949
# colData names(4): names sample_id line_id condition_name
```

Now we have a new `RangedSummarizedExperiment` object, with one row per gene. The row ranges have been summarized as well, and can be used for subsetting and interpretation just as for the transcripts.

At this point, the only information we have about the genes in our data set, apart from their genomic location and the associated transcript IDs, is the Ensembl ID. Often we need additional annotations, such as gene symbols. Bioconductor provides a range of annotation packages:

- `OrgDb` packages, providing gene-based annotations for a given organism
- `TxDb` and `EnsDb` packages, providing transcript ranges for a given genome build
- `BSgenome` packages, providing the genome sequence for a given genome build

For our purposes here, the appropriate `OrgDb` package is the most suitable, since it contains gene-centric ID conversion tables. Since this is human data, we will use the *org.Hs.eg.db (https://bioconductor.org/packages/3.17/org.Hs.eg.db)* package.

Hide

```
## Add gene symbols
sg <- tximeta::addIds(sg, "SYMBOL", gene = TRUE)
sg
# class: RangedSummarizedExperiment
# dim: 58294 24
# metadata(6): tximetaInfo quantInfo ... txomeInfo txdbInfo
# assays(3): counts abundance length
# rownames(58294): ENSG00000000003.14 ENSG00000000005.5 ...
#   ENSG00000285993.1 ENSG00000285994.1
# rowData names(3): gene_id tx_ids SYMBOL
# colnames(24): SAMEA103885102 SAMEA103885347 ... SAMEA103885308
#   SAMEA103884949
# colData names(4): names sample_id line_id condition_name
head(rowData(sg))
# DataFrame with 6 rows and 3 columns
#                             gene_id
#                         <character>
# ENSG00000000003.14 ENSG00000000003.14
# ENSG00000000005.5   ENSG00000000005.5
# ENSG00000000419.12 ENSG00000000419.12
# ENSG00000000457.13 ENSG00000000457.13
# ENSG00000000460.16 ENSG00000000460.16
# ENSG00000000938.12 ENSG00000000938.12
#                                                                   tx_ids
#                                                             <CharacterList>
# ENSG00000000003.14  ENST00000612152.4,ENST00000373020.8,ENST00000614008.4,...
# ENSG00000000005.5                         ENST00000373031.4,ENST00000485971.1
# ENSG00000000419.12  ENST00000371588.9,ENST00000466152.5,ENST00000371582.8,...
# ENSG00000000457.13 ENST00000367771.10,ENST00000367770.5,ENST00000367772.8,...
# ENSG00000000460.16  ENST00000498289.5,ENST00000472795.5,ENST00000496973.5,...
# ENSG00000000938.12  ENST00000374005.7,ENST00000399173.5,ENST00000374004.5,...
#                        SYMBOL
#                    <character>
# ENSG00000000003.14      TSPAN6
# ENSG00000000005.5         TNMD
# ENSG00000000419.12        DPM1
# ENSG00000000457.13       SCYL3
# ENSG00000000460.16       FIRRM
# ENSG00000000938.12         FGR
```

To see a list of the possible columns, use the `columns` function from the *AnnotationDbi (https://bioconductor.org/packages/3.17/AnnotationDbi)* package:

Hide

```
AnnotationDbi::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"
```

We can even add annotations where we expect (and would like to retain) multiple mapping values, e.g., associated GO terms:

Hide

```
sg <- addIds(sg, "GO", multiVals = "list", gene = TRUE)
head(rowData(sg))
# DataFrame with 6 rows and 4 columns
#                             gene_id
#                           <character>
# ENSG00000000003.14 ENSG00000000003.14
# ENSG00000000005.5   ENSG00000000005.5
# ENSG00000000419.12 ENSG00000000419.12
# ENSG00000000457.13 ENSG00000000457.13
# ENSG00000000460.16 ENSG00000000460.16
# ENSG00000000938.12 ENSG00000000938.12
#                                                                   tx_ids
#                                                             <CharacterList>
# ENSG00000000003.14  ENST00000612152.4,ENST00000373020.8,ENST00000614008.4,...
# ENSG00000000005.5                     ENST00000373031.4,ENST00000485971.1
# ENSG00000000419.12  ENST00000371588.9,ENST00000466152.5,ENST00000371582.8,...
# ENSG00000000457.13 ENST00000367771.10,ENST00000367770.5,ENST00000367772.8,...
# ENSG00000000460.16  ENST00000498289.5,ENST00000472795.5,ENST00000496973.5,...
# ENSG00000000938.12  ENST00000374005.7,ENST00000399173.5,ENST00000374004.5,...
#                        SYMBOL                                       GO
#                     <character>                                <list>
# ENSG00000000003.14      TSPAN6 GO:0005515,GO:0005886,GO:0039532,...
# ENSG00000000005.5         TNMD GO:0001886,GO:0001937,GO:0005515,...
# ENSG00000000419.12        DPM1 GO:0004169,GO:0004582,GO:0004582,...
# ENSG00000000457.13       SCYL3 GO:0000139,GO:0005515,GO:0005524,...
# ENSG00000000460.16       FIRRM                           GO:0005515
# ENSG00000000938.12         FGR GO:0001784,GO:0001784,GO:0001819,...
```

Note that *Salmon* returns *estimated* or *expected* counts, which are not necessarily integers. They may need to be rounded before they are passed to count-based statistical methods. To obtain consistent results with different pipelines, we round the estimated counts here (note that in practice, *DESeq2 (https://bioconductor.org/packages/3.17/DESeq2)* will automatically round the counts, while *edgeR (https://bioconductor.org/packages/3.17/edgeR)* will work well also with the non-integer values).

Hide

```
assay(sg, "counts") <- round(assay(sg, "counts"))
```

Quiz

- Can you start your DE analysis with normalized values such as TPMs?
- Where can I store additional information about the samples?
- Where can I store additional information about the features? (What can your features be?)
- Can you think of a way to read in the required information if you start (AAAAAAAH) from Excel files?

5 Representing counts for differential expression packages

At this point, we have a gene-level count matrix, contained in our *SummarizedExperiment* object. This is a branching point where we could use a variety of Bioconductor packages for exploration and differential expression of the count matrix, including *edgeR (https://bioconductor.org/packages/3.17/edgeR)* (Robinson, McCarthy, and Smyth 2009), *DESeq2 (https://bioconductor.org/packages/3.17/DESeq2)* (Love, Huber, and Anders 2014), *limma (https://bioconductor.org/packages/3.17/limma)* with the voom method (Law et al. 2014), *DSS (https://bioconductor.org/packages/3.17/DSS)* (Wu, Wang, and Wu 2013), *EBSeq (https://bioconductor.org/packages/3.17/EBSeq)* (Leng et al. 2013) and *BaySeq (https://bioconductor.org/packages/3.17/BaySeq)* (Hardcastle and Kelly 2010). We will continue using *DESeq2* and *edgeR*.

Bioconductor software packages often define and use a custom class for storing data that makes sure that all the needed data slots are consistently provided and fulfill any requirements. In addition, Bioconductor has general data classes (such as the *SummarizedExperiment*) that can be used to move data between packages. The *DEFormats (https://bioconductor.org/packages/3.17/DEFormats)* package can be useful for converting between different classes. The core Bioconductor classes also provide useful functionality: for example, subsetting or reordering the rows or columns of a *SummarizedExperiment* automatically subsets or reorders the associated *rowRanges* and *colData*, which can help to prevent accidental sample swaps that would otherwise lead to spurious results. With *SummarizedExperiment* this is all taken care of behind the scenes.

Each of the packages we will use for differential expression has a specific class of object used to store the summarization of the RNA-seq experiment and the intermediate quantities that are calculated during the statistical analysis of the data. *DESeq2* uses a *DESeqDataSet* and *edgeR* uses a *DGEList*.

5.1 The *DESeqDataSet*, sample information, and the design formula

In *DESeq2*, the custom class is called *DESeqDataSet*. It is built on top of the *SummarizedExperiment* class, and it is easy to convert *SummarizedExperiment* objects into *DESeqDataSet* objects. One of the two main differences compared to a *SummarizedExperiment* object is that the `assay` slot can be accessed using the `counts` accessor function, and the *DESeqDataSet* class enforces that the values in this matrix are non-negative integers.

A second difference is that the *DESeqDataSet* has an associated *design formula*. The experimental design is specified at the beginning of the analysis, as it will inform many of the *DESeq2* functions how to treat the samples in the analysis (one exception is the size factor estimation, i.e., the adjustment for differing library sizes, which does not depend on the design formula). The design formula tells which columns in the sample information table (`colData`) specify the experimental design and how these factors should be used in the analysis.

Let's remind ourselves of the design of our experiment:

Hide

```
colData(sg)
# DataFrame with 24 rows and 4 columns
#                         names   sample_id     line_id condition_name
#                   <character> <character> <character>    <character>
# SAMEA103885102 SAMEA103885102      diku_A      diku_1          naive
# SAMEA103885347 SAMEA103885347      diku_B      diku_1           IFNg
# SAMEA103885043 SAMEA103885043      diku_C      diku_1         SL1344
# SAMEA103885392 SAMEA103885392      diku_D      diku_1    IFNg_SL1344
# SAMEA103885182 SAMEA103885182      eiwy_A      eiwy_1          naive
# ...                         ...         ...         ...            ...
# SAMEA103885021 SAMEA103885021      podx_D      podx_1    IFNg_SL1344
# SAMEA103885262 SAMEA103885262      qaqx_A      qaqx_1          naive
# SAMEA103885228 SAMEA103885228      qaqx_B      qaqx_1           IFNg
# SAMEA103885308 SAMEA103885308      qaqx_C      qaqx_1         SL1344
# SAMEA103884949 SAMEA103884949      qaqx_D      qaqx_1    IFNg_SL1344
```

We have samples from four different conditions, and six donors:

Hide

```
table(colData(sg)$condition_name)
#
#        IFNg IFNg_SL1344       naive      SL1344
#           6           6           6           6
table(colData(sg)$line_id)
#
# diku_1 eiwy_1 fikt_3 ieki_2 podx_1 qaqx_1
#      4      4      4      4      4      4

# possible to use a shortcut
table(sg$line_id)
#
# diku_1 eiwy_1 fikt_3 ieki_2 podx_1 qaqx_1
#      4      4      4      4      4      4
```

We want to find the changes in gene expression that can be associated with the different treatments, but we also want to control for differences between the donors. The design which accomplishes this is obtained by writing `~ line_id + condition_name` . By including `line_id` , terms will be added to the model which account for differences across donors, and by adding `condition_name` we get terms representing the different treatment effects.

Note: it will be helpful for us if the first level of a factor is the reference level (e.g. control, or untreated samples). The reason is that by specifying this, functions further in the pipeline can be used and will give comparisons such as 'treatment vs control', without needing to specify additional arguments.

We can *relevel* the `condition_name` factor like so:

Hide

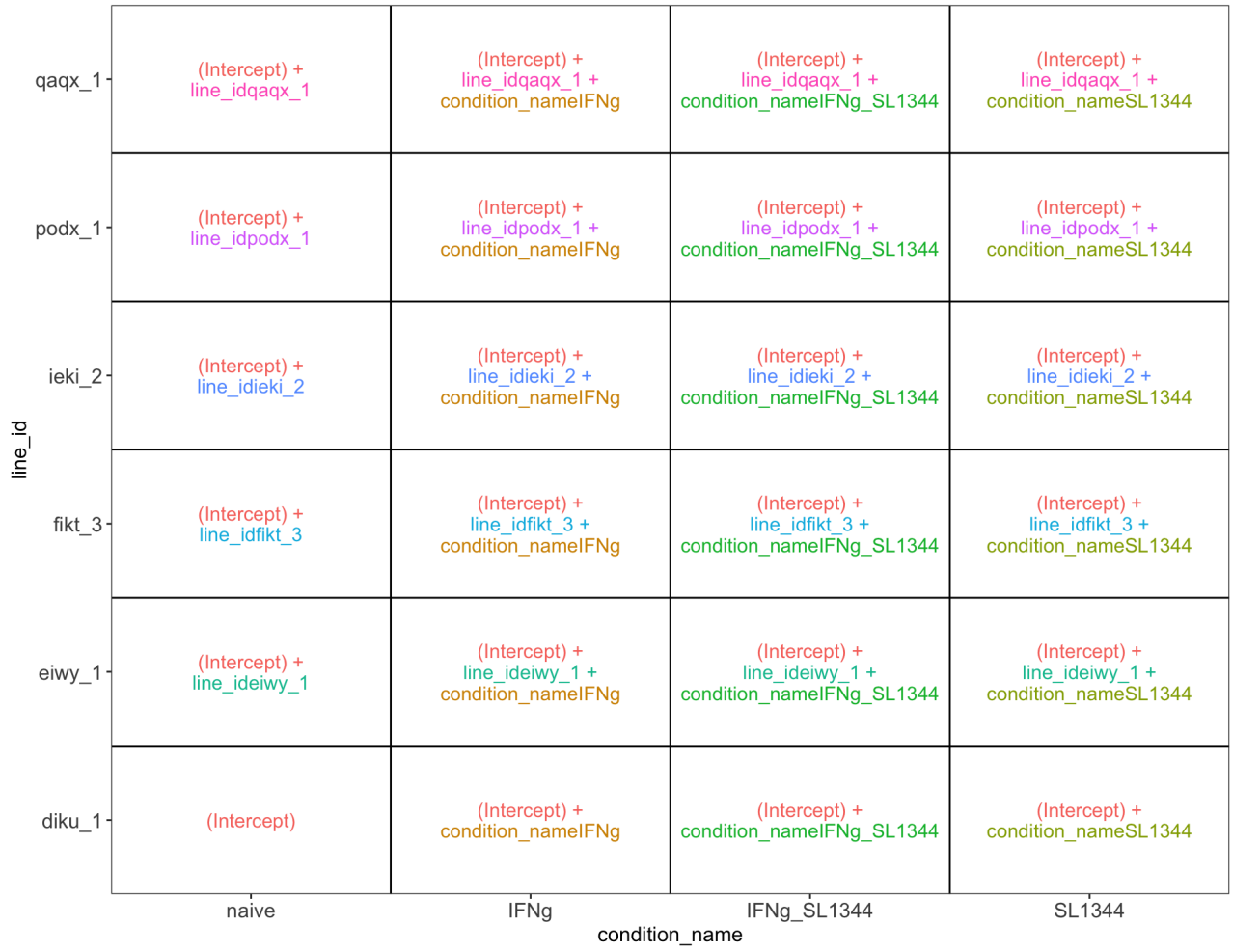
```
colData(sg)$condition_name <- factor(colData(sg)$condition_name)
colData(sg)$condition_name <- relevel(colData(sg)$condition_name, ref = "naive")
colData(sg)$condition_name
#  [1] naive       IFNg        SL1344      IFNg_SL1344 naive       IFNg
#  [7] SL1344      IFNg_SL1344 naive       IFNg        SL1344      IFNg_SL1344
# [13] naive       IFNg        SL1344      IFNg_SL1344 naive       IFNg
# [19] SL1344      IFNg_SL1344 naive       IFNg        SL1344      IFNg_SL1344
# Levels: naive IFNg IFNg_SL1344 SL1344
```

You can use R's formula notation to express any fixed-effects experimental design for *edgeR* or *DESeq2*. Note that these packages use the same formula notation as, for instance, the `lm` function of base R.

Using the *ExploreModelMatrix (https://bioconductor.org/packages/3.17/ExploreModelMatrix)* R/Bioconductor package, we can represent our design in a graphical way:

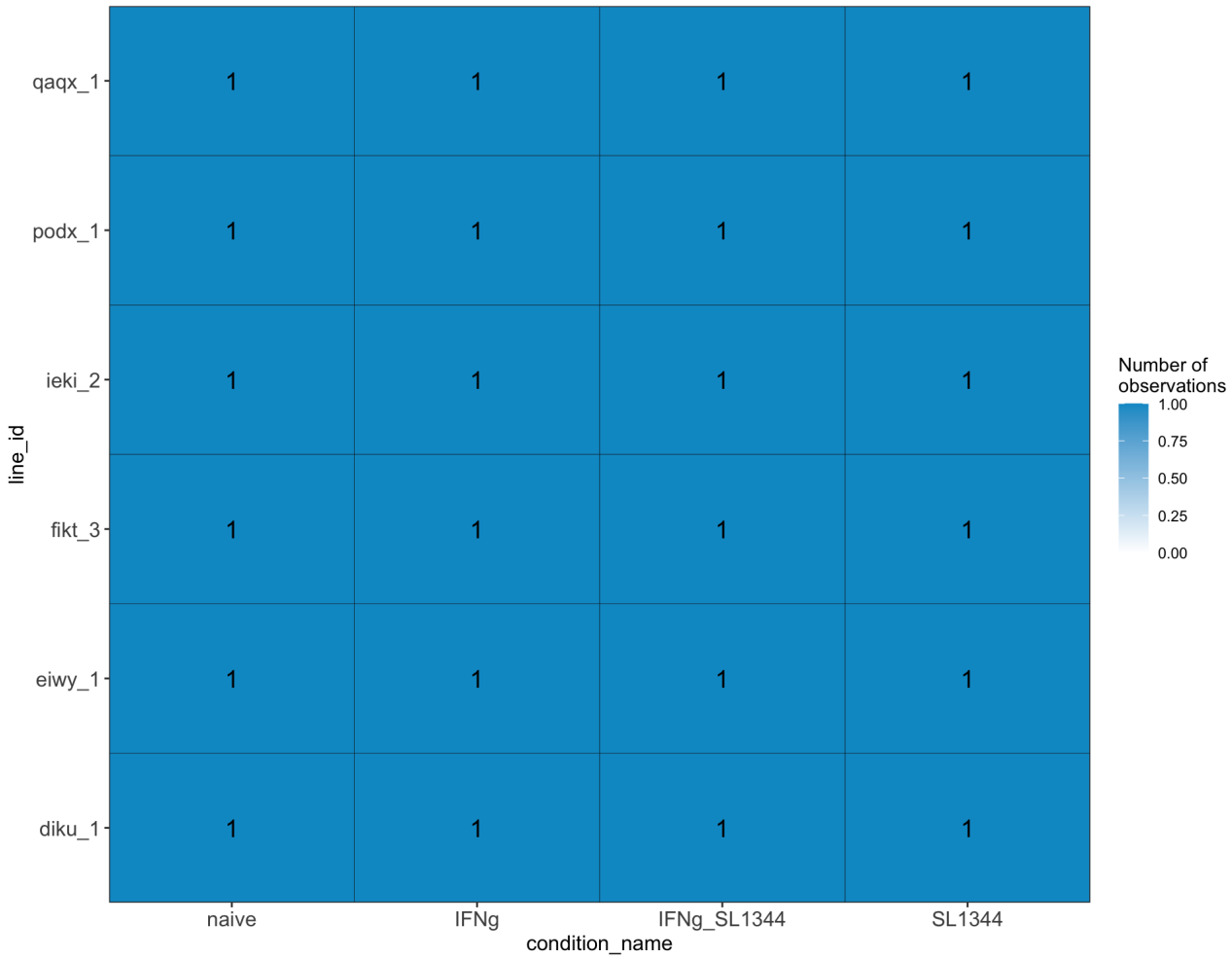
Hide

```
library(ExploreModelMatrix)
vd <- VisualizeDesign(sampleData = colData(sg),
                      designFormula = ~ line_id + condition_name,
                      textSizeFitted = 4)
vd$plotlist
# [[1]]
```



Hide

```
vd$cooccurrenceplots
# [[1]]
```



We can also open the interactive interface to explore our design further:

Hide

```
ExploreModelMatrix(sampleData = colData(sg),
                   designFormula = ~ line_id + condition_name)
```

To generate a *DESeqDataSet* object from a *SummarizedExperiment* object, we only need to additionally provide the experimental design in terms of a formula.

Hide

```
dds <- DESeqDataSet(sg, design = ~ line_id + condition_name)
```

We can also create a *DESeqDataSet* directly from a count matrix, a data frame with sample information and a design formula (see the `DESeqDataSetFromMatrix` function).

5.2 The *DGEList*

As mentioned above, the *edgeR* package uses another type of data container, namely a *DGEList* object. *tximeta* provides a convenient wrapper function to generate a *DGEList* from the gene-level *SummarizedExperiment* object:

Hide

```
library(edgeR)

dge <- tximeta::makeDGEList(sg)
names(dge)
# [1] "counts"  "samples" "genes"   "offset"
head(dge$samples)
#                group lib.size norm.factors          names sample_id line_id
# SAMEA103885102     1 40074879            1 SAMEA103885102    diku_A  diku_1
# SAMEA103885347     1 40467661            1 SAMEA103885347    diku_B  diku_1
# SAMEA103885043     1 41832780            1 SAMEA103885043    diku_C  diku_1
# SAMEA103885392     1 42535180            1 SAMEA103885392    diku_D  diku_1
# SAMEA103885182     1 40738502            1 SAMEA103885182    eiwy_A  eiwy_1
# SAMEA103885136     1 39701890            1 SAMEA103885136    eiwy_B  eiwy_1
#                condition_name
# SAMEA103885102          naive
# SAMEA103885347           IFNg
# SAMEA103885043         SL1344
# SAMEA103885392    IFNg_SL1344
# SAMEA103885182          naive
# SAMEA103885136           IFNg
```

As for the *DESeqDataSet*, a *DGEList* can also be generated directly from a count matrix and sample metadata (see the `DGEList()` constructor function). Just like the *SummarizedExperiment* and the *DESeqDataSet*, the *DGEList* contains all the information we need: the count matrix, information about the samples (the columns of the count matrix), and information about the genes (the rows of the count matrix). One difference compared to the *DESeqDataSet* is that the experimental design is not defined when creating the *DGEList*, but later in the workflow.

Quiz

- How important is the definition of the design?
- How do I quantify the effect size? “In which direction” is this to be interpreted?
- Is it possible to change the reference in the comparison?
- If you have multiple experimental factors, how should you proceed? Think of 2 condition-2 tissues/cell types

6 Filtering

It is often helpful to filter out lowly expressed genes before continuing with the analysis, to remove features that have nearly no information, increase the speed of the analysis and reduce the size of the data. At the very least we exclude genes with zero counts across all samples.

Hide

```
nrow(dds)
# [1] 58294
table(rowSums(assay(dds, "counts")) == 0)
#
# FALSE  TRUE
# 38829 19465
```

Here, we additionally remove genes that have a single read across the samples.

Hide

```
keep <- rowSums(counts(dds)) > 1
dds <- dds[keep, ]
dge <- dge[match(rownames(dds), rownames(dge)), ]
dim(dds)
# [1] 35683    24
dim(dge)
# [1] 35683    24
```

Importantly, the group information should *not* be used to define the filtering criterion, since that can interfere with the validity of the p-values downstream.

Quiz

- Removing unexpressed features might influence what you do when integrating different analyses. How could we proceed in such cases?
- What if I download publicly available data and some genes are missing? What can be some of the possible reasons for this?

7 Exploratory analysis and visualization

There are two separate analysis paths in this tutorial:

1. *visual exploration* of sample relationships, in which we will discuss transformation of the counts for computing distances or making plots
2. *statistical testing* for differences attributable to treatment, controlling for donor effects

Importantly, the statistical testing methods rely on original count data (not scaled or transformed) for calculating the precision of measurements. However, for visualization and exploratory analysis, transformed counts are typically more suitable. Thus, it is critical to separate the two workflows and use the appropriate input data for each of them.

7.1 Transformations

Many common statistical methods for exploratory analysis of multidimensional data, for example clustering and *principal components analysis* (PCA), work best for data that generally has the same range of variance at different ranges of the mean values. When the expected amount of variance is approximately the same across different mean values, the data is said to be *homoskedastic*. For RNA-seq raw counts, however, the variance grows with the mean. For example, if one performs PCA directly on a matrix of size-factor-normalized read counts, the result typically depends only on the few most strongly expressed genes because they show the largest absolute differences between samples. A simple and often used strategy to avoid this is to take the logarithm of the normalized count values plus a small pseudocount; however, now the genes with the very lowest counts will tend to dominate the results because, due to the strong Poisson noise inherent to small count values, and the fact that the logarithm amplifies differences for the smallest values, these low count genes will show the strongest relative differences between samples.

As a solution, *DESeq2* offers transformations for count data that stabilize the variance across the mean: the *regularized logarithm* (rlog) and the *variance stabilizing transformation* (VST). These have slightly different implementations, discussed a bit in the *DESeq2* paper and in the vignette, but a similar goal of stabilizing the variance across the range of values. Both produce log2-like values for high counts. Here we will use the variance stabilizing transformation implemented with the `vst` function.

Hide

```
vsd <- DESeq2::vst(dds)
```

This returns a *DESeqTransform* object…

Hide

```
class(vsd)
# [1] "DESeqTransform"
# attr(,"package")
# [1] "DESeq2"
```

…which retains all the column metadata that was attached to the *DESeqDataSet*:

Hide

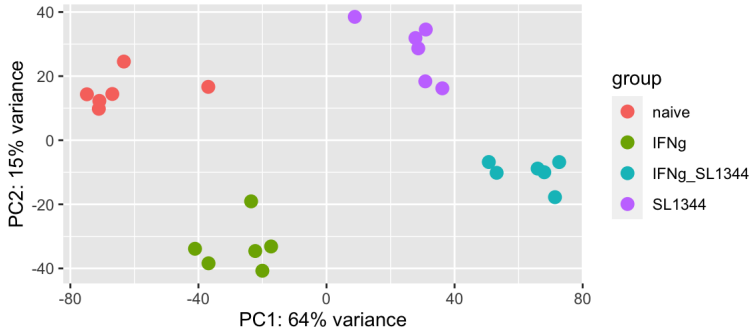
```
head(colData(vsd), 3)
# DataFrame with 3 rows and 4 columns
#                        names   sample_id  line_id condition_name
#                  <character> <character> <factor>       <factor>
# SAMEA103885102 SAMEA103885102      diku_A   diku_1          naive
# SAMEA103885347 SAMEA103885347      diku_B   diku_1          IFNg
# SAMEA103885043 SAMEA103885043      diku_C   diku_1          SL1344
```

7.2 PCA plot

One way to visualize sample-to-sample distances is a principal components analysis (PCA). In this ordination method, the data points (here, the samples) are projected onto the 2D plane such that they spread out in the two directions that explain most of the differences (Figure below). The x-axis (the first principal component, or *PC1*) is the direction that separates the data points the most (i.e., the direction with the largest variance). The y-axis (the second principal component, or *PC2*) represents the direction with largest variance subject to the constraint that it must be *orthogonal* to the first direction. The percent of the total variance that is contained in the direction is printed in the axis label. Note that these percentages do not sum to 100%, because there are more dimensions that contain the remaining variance (although each of these remaining dimensions will explain less than the two that we see).

Hide

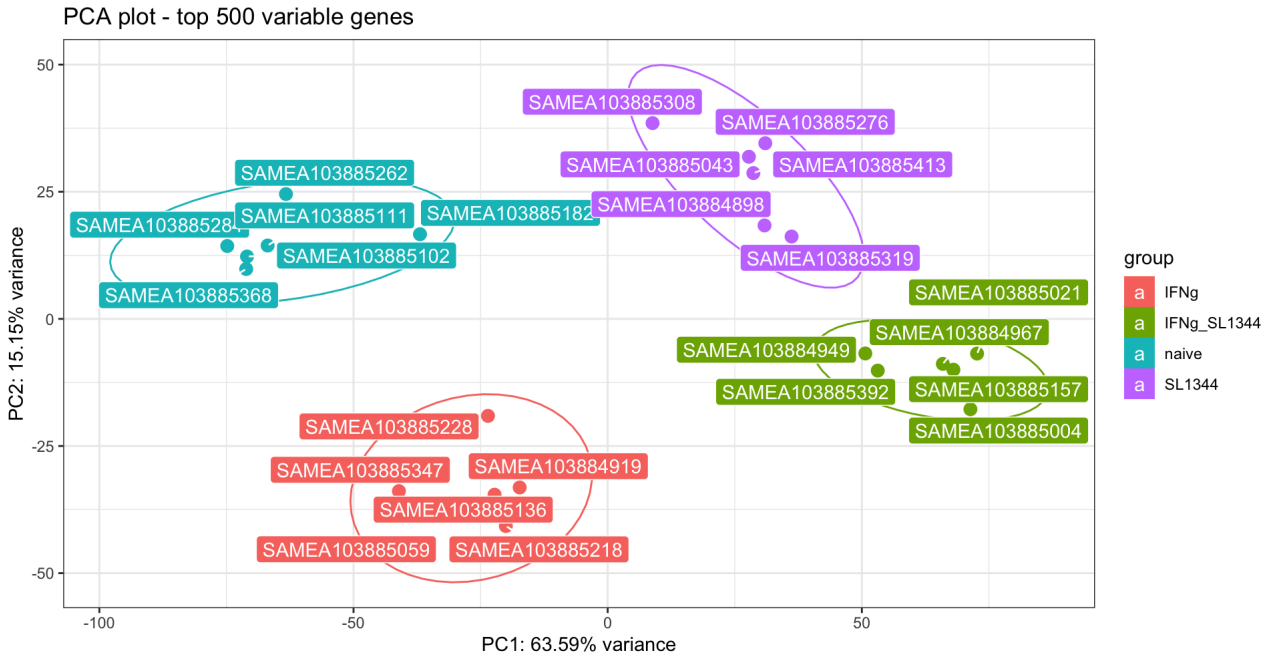
```
DESeq2::plotPCA(vsd, intgroup = "condition_name")
```



Additionally, the *pcaExplorer* package has some functionality on top to explore datasets from the point of view of Principal Components - including also a functional interpretation of it with the `pca2go()` function.

Hide

```
library(pcaExplorer)
pcaplot(vsd, intgroup = "condition_name", ellipse = TRUE)
```



7.3 MDS plot

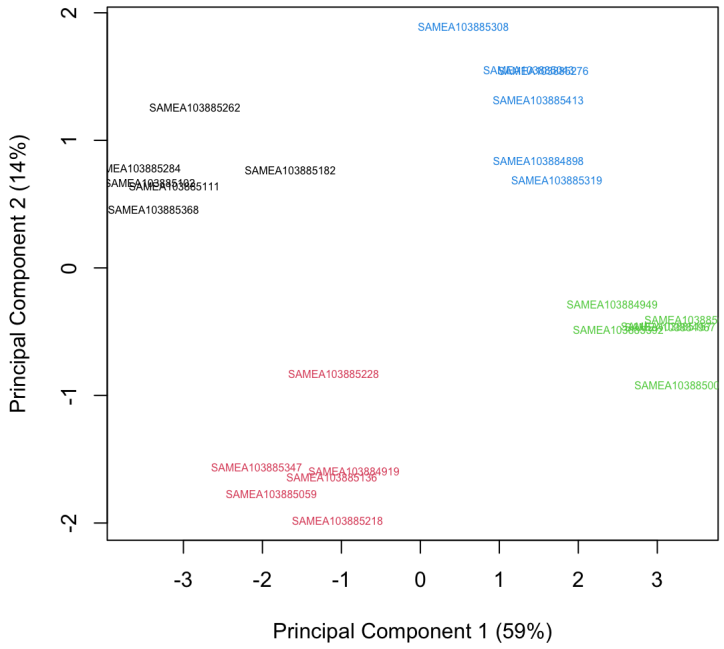
Another way to reduce dimensionality, which is in many ways similar to PCA, is *multidimensional scaling* (MDS). For MDS, we first have to calculate all pairwise distances between our objects (samples in this case), and then create a (typically) two-dimensional representation where these pre-calculated distances are represented as accurately as possible. This means that depending on how the pairwise sample distances are defined, the two-dimensional plot can be very different, and it is important to choose a distance that is suitable for the type of data at hand.

edgeR contains a function `plotMDS`, which operates on a *DGEList* object and generates a two-dimensional MDS representation of the samples. The default distance between two samples can be interpreted as the "typical" log fold change between the two samples, for the genes that are most different between them (by default, the top 500 genes, but this can be modified). We generate an MDS plot from the *DGEList* object `dge`, coloring by the treatment and using different plot symbols for different donors.

Note: Since the *DGEList* was created using the `makeDGEList` function, the average transcript length offsets have been incorporated in the object and will be used as offsets in downstream analysis. If this is not the case, we need to estimate TMM normalization factors before performing further analysis.

Hide

```
# dge <- edgeR::calcNormFactors(dge)
plotMDS(dge, top = 500, labels = NULL,
        col = as.numeric(dge$samples$condition_name),
        cex = 0.5, gene.selection = "common")
```



Quiz

- What if I do not see a clear separation in my PCA plot? Is it still ok to proceed?
- What should I do if I detect/am aware of a batch effect?
- Raw data vs normalized data vs transformed data: Which one should I use in "all the cases" I could encounter?
- "I have seen people using tSNE/UMAP in single cell data, why shouldn't we do the same here?"

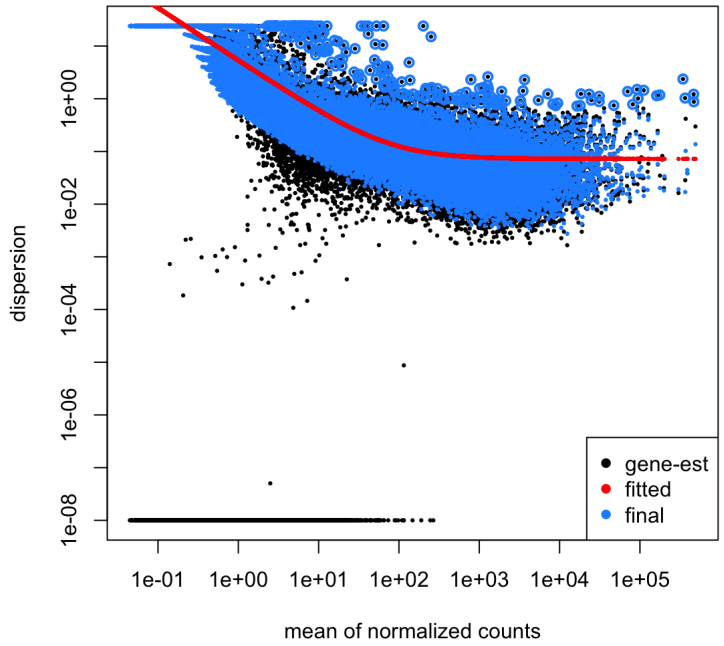
8 Differential expression analysis

8.1 Performing differential expression testing with *DESeq2*

As we have already specified an experimental design when we created the *DESeqDataSet*, we can run the differential expression pipeline on the raw counts with a single call to the function `DESeq`. We can also plot the estimated dispersions.

Hide

```
dds <- DESeq2::DESeq(dds)
DESeq2::plotDispEsts(dds)
```



The `DESeq` function will print out a message for the various steps it performs. These are described in more detail in the manual page, which can be accessed by typing `?DESeq`. Briefly these are: the estimation of size factors (controlling for differences in the sequencing depth of the samples), the estimation of dispersion values for each gene, and fitting a generalized linear model.

A *DESeqDataSet* is returned that contains all the fitted parameters within it, and the following section describes how to extract out results tables of interest from this object.

Calling the `results` function without any arguments will extract the estimated log2 fold changes and *p* values for the last variable in the design formula. If there are more than 2 levels for this variable, `results` will extract the results table for a comparison of the last level over the first level. This comparison is printed at the top of the output: `condition name SL1344 vs naive`. Other comparisons can be performed via the `contrast` argument. For example, we will focus on comparing the IFN gamma treatment to the naive group.

Hide

```
## Default - SL1344 vs naive
res <- DESeq2::results(dds)
head(res)
# log2 fold change (MLE): condition name SL1344 vs naive
# Wald test p-value: condition name SL1344 vs naive
# DataFrame with 6 rows and 6 columns
#                    baseMean log2FoldChange     lfcSE      stat      pvalue
#                   <numeric>      <numeric> <numeric> <numeric>   <numeric>
# ENSG00000000003.14   171.782      0.1171248 0.3008327  0.389335 6.97028e-01
# ENSG00000000419.12   967.527      0.0886824 0.0860008  1.031181 3.02456e-01
# ENSG00000000457.13   681.637      0.7109442 0.1973877  3.601766 3.16062e-04
# ENSG00000000460.16   263.282     -1.0347169 0.2179499 -4.747499 2.05947e-06
# ENSG00000000938.12  2646.887      1.6453083 0.2348000  7.007275 2.43005e-12
# ENSG00000000971.15  3045.742      0.7794411 0.4980265  1.565059 1.17569e-01
#                          padj
#                     <numeric>
# ENSG00000000003.14 8.11132e-01
# ENSG00000000419.12 4.55113e-01
# ENSG00000000457.13 1.22290e-03
# ENSG00000000460.16 1.18399e-05
# ENSG00000000938.12 3.14455e-11
# ENSG00000000971.15 2.20491e-01

## We'll instead focus on IFNgamma vs naive
res <- DESeq2::results(dds, contrast = c("condition_name", "IFNg", "naive"))
head(res)
# log2 fold change (MLE): condition_name IFNg vs naive
# Wald test p-value: condition name IFNg vs naive
# DataFrame with 6 rows and 6 columns
#                    baseMean log2FoldChange     lfcSE      stat      pvalue
#                   <numeric>      <numeric> <numeric> <numeric>   <numeric>
# ENSG00000000003.14   171.782     -0.2829860 0.3010930 -0.939862 3.47288e-01
# ENSG00000000419.12   967.527      0.0383933 0.0856623  0.448194 6.54013e-01
# ENSG00000000457.13   681.637      1.2838945 0.1966270  6.529593 6.59486e-11
# ENSG00000000460.16   263.282     -1.4725128 0.2183088 -6.745092 1.52930e-11
# ENSG00000000938.12  2646.887      0.6747921 0.2351631  2.869464 4.11168e-03
# ENSG00000000971.15  3045.742      4.9869519 0.4966828 10.040518 1.01142e-23
#                          padj
#                     <numeric>
# ENSG00000000003.14 5.77116e-01
# ENSG00000000419.12 8.25573e-01
# ENSG00000000457.13 1.62287e-09
# ENSG00000000460.16 4.10702e-10
# ENSG00000000938.12 1.89738e-02
# ENSG00000000971.15 1.13504e-21
```

As `res` is a *DataFrame* object, it carries metadata with information on the meaning of the columns:

Hide

```
mcols(res, use.names = TRUE)
# DataFrame with 6 rows and 2 columns
#                        type            description
#                 <character>            <character>
# baseMean       intermediate mean of normalized c..
# log2FoldChange      results log2 fold change (ML..
# lfcSE               results standard error: cond..
# stat                results Wald statistic: cond..
# pvalue              results Wald test p-value: c..
# padj                results   BH adjusted p-values
```

The first column, `baseMean`, is a just the average of the normalized count values, dividing by size factors, taken over all samples in the *DESeqDataSet*. The remaining four columns refer to a specific contrast, namely the comparison of the `IFNg` level over the `naive` level for the factor variable `condition_name`.

The column `log2FoldChange` is the effect size estimate. It tells us how much the gene's expression seems to have changed due to infection with IFN gamma in comparison to naive samples. This value is reported on a logarithmic scale to base 2: for example, a log2 fold change of 1.5 means that the gene's expression is increased by a multiplicative factor of 2^1.5.

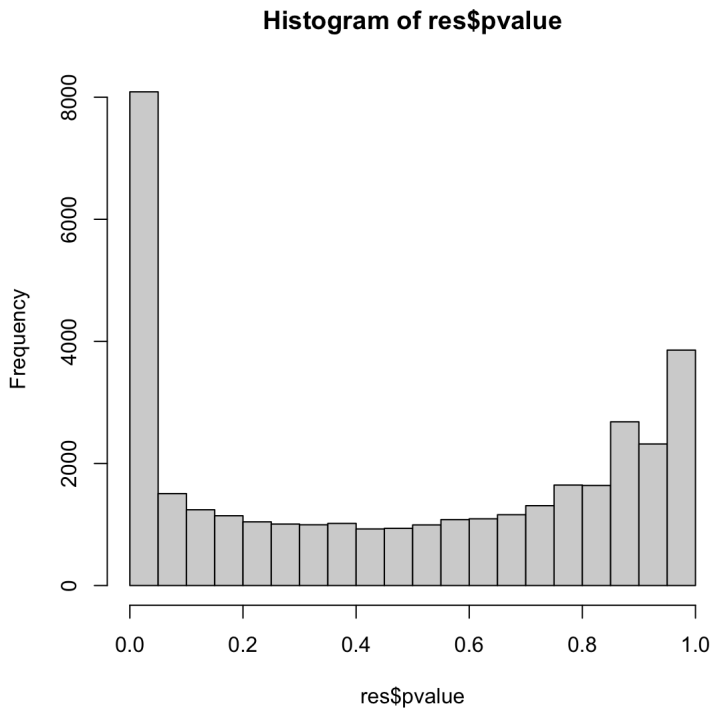
Of course, this estimate has an uncertainty associated with it, which is available in the column `lfcSE`, the standard error estimate for the log2 fold change estimate. We can also express the uncertainty of a particular effect size estimate as the result of a statistical test. The purpose of a test for differential expression is to test whether the data provides sufficient evidence to conclude that this value is really different from zero. *DESeq2* performs for each gene a *hypothesis test* to see whether evidence is sufficient to decide against the *null hypothesis* that there is zero effect of the treatment on the gene and that the observed difference between treatment

and control was merely caused by experimental variability (i.e., the type of variability that you can expect between different samples in the same treatment group). As usual in statistics, the result of this test is reported as a *p* value, and it is found in the column `pvalue`. Remember that a *p* value indicates the probability that an effect as strong as the observed one, or even stronger, would be seen under the situation described by the null hypothesis.

We can also summarize the results with the following line of code, which reports some additional information, that will be covered in later sections.

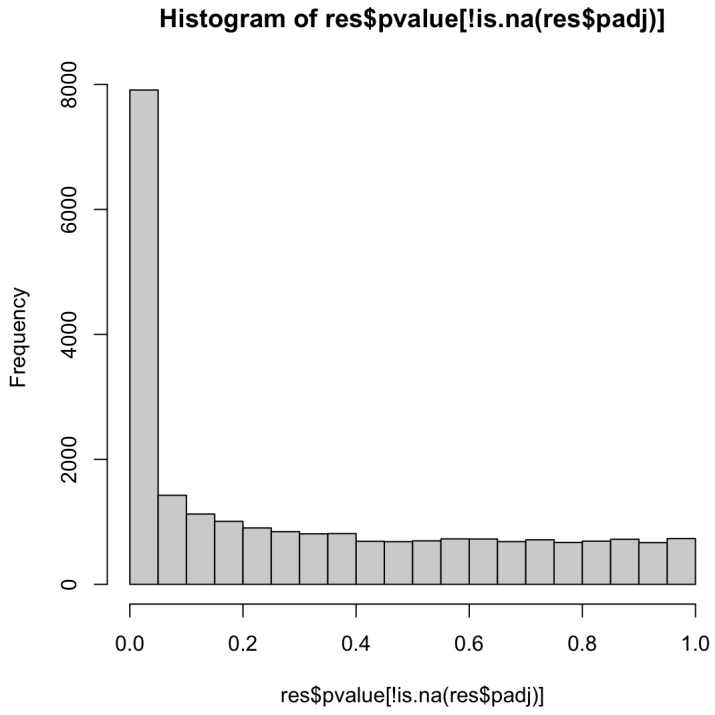
Hide

```
summary(res)
#
# out of 35683 with nonzero total read count
# adjusted p-value < 0.1
# LFC > 0 (up)       : 3718, 10%
# LFC < 0 (down)     : 3437, 9.6%
# outliers [1]       : 0, 0%
# low counts [2]     : 12453, 35%
# (mean count < 2)
# [1] see 'cooksCutoff' argument of ?results
# [2] see 'independentFiltering' argument of ?results
hist(res$pvalue)
```



Hide

```
## Remove the genes that were filtered out in the independent filtering
hist(res$pvalue[!is.na(res$padj)])
```



Hide

```
## We also add a couple of extra columns that will be useful for the interactive
## visualization later
rowData(dds)$log10Dispersion <- log10(rowData(dds)$dispersion)

restmp <- DataFrame(res)
restmp$log10BaseMean <- log10(restmp$baseMean)
restmp$mlog10PValue <- -log10(restmp$pvalue)
colnames(restmp) <- paste0("DESeq2_IFNg_vs_naive_", colnames(restmp))
rowData(dds) <- cbind(rowData(dds), restmp)
```

Note that there are many genes with differential expression due to IFN gamma treatment at the FDR level of 10%. There are two ways to be more strict about which set of genes are considered significant:

- lower the false discovery rate threshold (the threshold on `padj` in the results table)
- raise the log2 fold change threshold from 0 using the `lfcThreshold` argument of *results*

If we lower the false discovery rate threshold, we should also tell this value to `results()`, so that the function will use an alternative threshold for the optimal independent filtering step:

Hide

```
res.05 <- results(dds, alpha = 0.05,
                  contrast = c("condition_name", "IFNg", "naive"))
table(res.05$padj < 0.05)
#
# FALSE  TRUE
# 16423  6115
```

If we want to raise the log2 fold change threshold, so that we test for genes that show more substantial changes due to treatment, we simply supply a value on the log2 scale. For example, by specifying `lfcThreshold = 1`, we look for genes that show significant effects of treatment on gene counts more than doubling or less than halving, because 2^1 = 2.

Hide

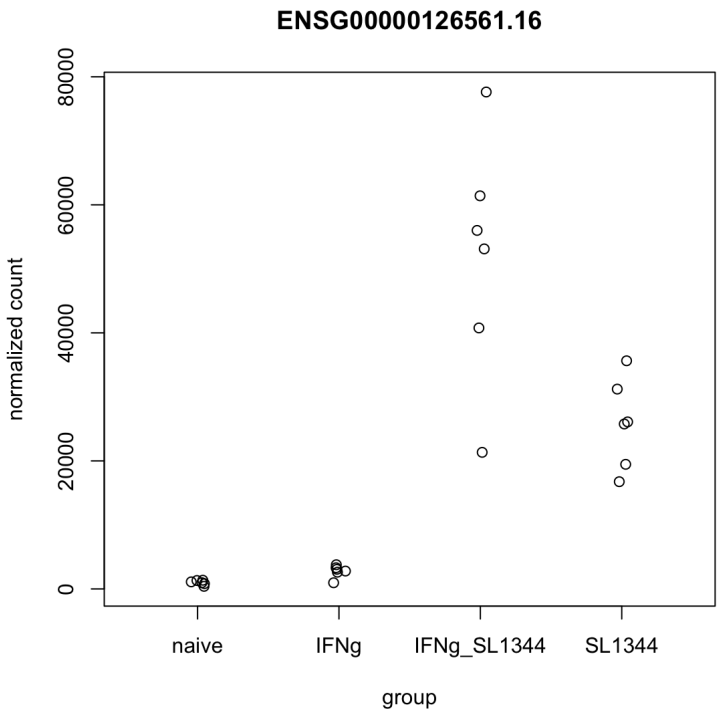
```
resLFC1 <- results(dds, lfcThreshold = 1,
                   contrast = c("condition_name", "IFNg", "naive"))
summary(resLFC1)
#
# out of 35683 with nonzero total read count
# adjusted p-value < 0.1
# LFC > 1.00 (up)    : 687, 1.9%
# LFC < -1.00 (down) : 419, 1.2%
# outliers [1]       : 0, 0%
# low counts [2]     : 0, 0%
# (mean count < 0)
# [1] see 'cooksCutoff' argument of ?results
# [2] see 'independentFiltering' argument of ?results
table(resLFC1$padj < 0.1)
#
# FALSE  TRUE
# 34577  1106
```

Sometimes a subset of the *p* values in `res` will be `NA` ("not available"). This is *DESeq*'s way of reporting that all counts for this gene were zero, and hence no test was applied. In addition, *p* values can be assigned `NA` if the gene was excluded from analysis because it contained an extreme count outlier. For more information, see the outlier detection section of the *DESeq2* vignette.

With *DESeq2*, there is also an easy way to plot the (normalized, transformed) counts for specific genes, using the `plotCounts` function:

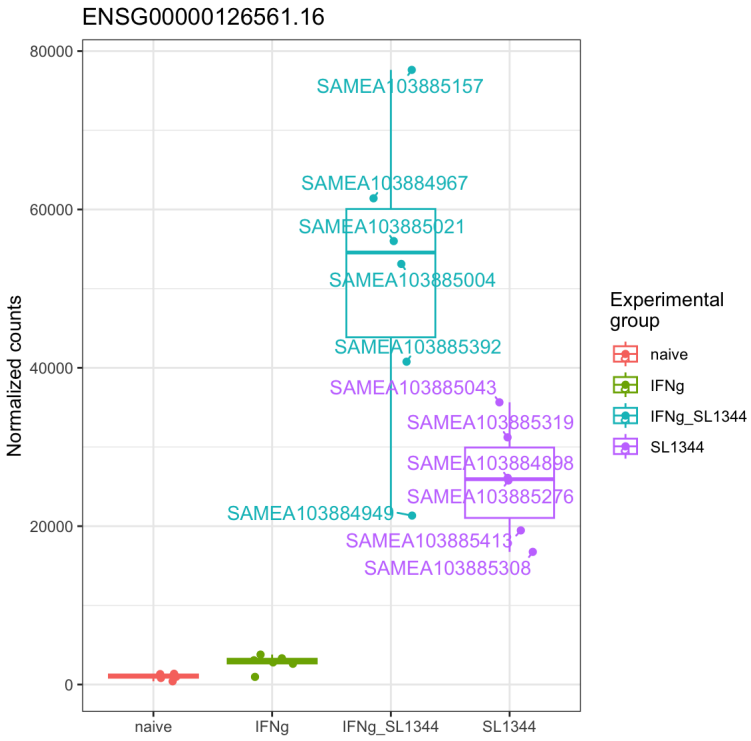
Hide

```
plotCounts(dds, gene = "ENSG00000126561.16", intgroup = "condition_name",
           normalized = TRUE, transform = FALSE)
```



Hide

```
GeneTonic::gene_plot(dds, gene = "ENSG00000126561.16", intgroup = "condition_name",
           normalized = TRUE, transform = FALSE)
```



8.2 Performing differential expression testing with *edgeR*

Next we will show how to perform differential expression analysis with *edgeR*. Recall that we have a *DGEList* `dge` , containing all the necessary information:

Hide

```
names(dge)
# [1] "counts"  "samples" "genes"   "offset"
```

We first define a design matrix, using the same formula syntax as for *DESeq2* above.

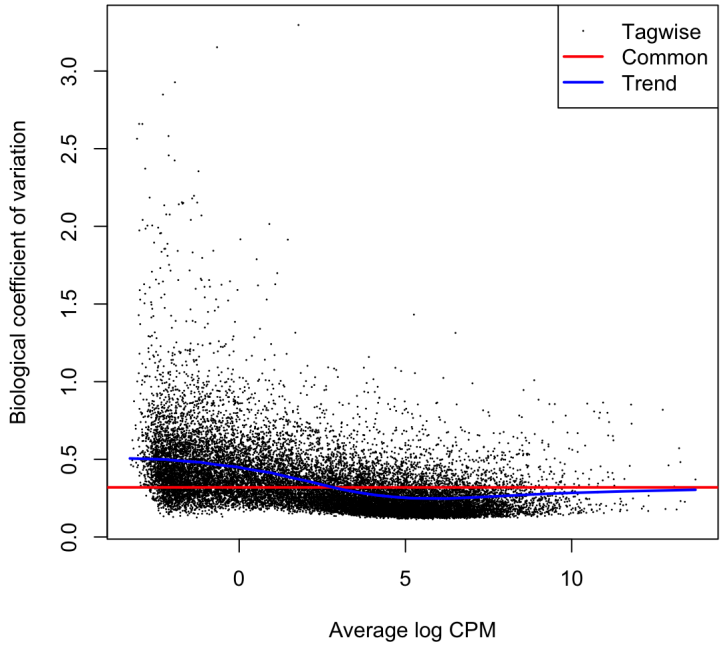
Hide

```
design <- model.matrix(~ line_id + condition_name, data = dge$samples)
head(design)
#                (Intercept) line_ideiwy_1 line_idfikt_3 line_idieki_2
# SAMEA103885102           1             0             0             0
# SAMEA103885347           1             0             0             0
# SAMEA103885043           1             0             0             0
# SAMEA103885392           1             0             0             0
# SAMEA103885182           1             1             0             0
# SAMEA103885136           1             1             0             0
#                line_idpodx_1 line_idqaqx_1 condition_nameIFNg
# SAMEA103885102             0             0                 0
# SAMEA103885347             0             0                 1
# SAMEA103885043             0             0                 0
# SAMEA103885392             0             0                 0
# SAMEA103885182             0             0                 0
# SAMEA103885136             0             0                 1
#                condition_nameIFNg_SL1344 condition_nameSL1344
# SAMEA103885102                         0                    0
# SAMEA103885347                         0                    0
# SAMEA103885043                         0                    1
# SAMEA103885392                         1                    0
# SAMEA103885182                         0                    0
# SAMEA103885136                         0                    0
```

While *DESeq2* performs independent filtering of lowly expressed genes internally, this is done by the user before applying *edgeR*. Here, we filter out lowly expressed genes using the `filterByExpr()` function, and then estimate the dispersion for each gene. Note that it is important that we specify the design in the dispersion calculation (it will be used to determine a suitable number of samples to require a gene to be expressed in). Afterwards, we plot the estimated dispersions.

Hide

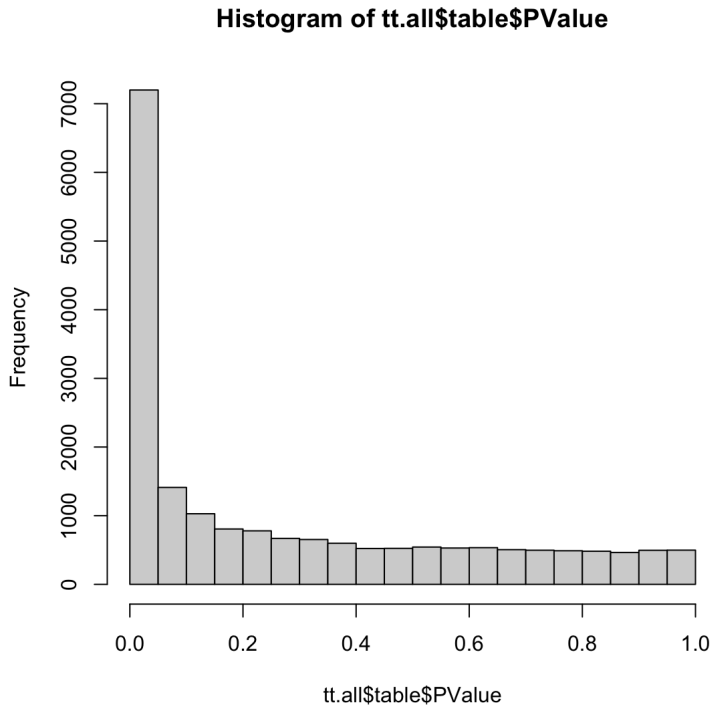
```
keep <- edgeR::filterByExpr(dge, design)
dge <- dge[keep, ]
dge <- edgeR::estimateDisp(dge, design)
edgeR::plotBCV(dge)
```



Finally, we fit the generalized linear model and perform the test. In the `glmQLFTest` function, we indicate which coefficient (which column in the design matrix) that we would like to test for. It is possible to test more general contrasts as well, and the user guide contains many examples on how to do this. The `topTags` function extracts the top-ranked genes. You can indicate the adjusted p-value cutoff, and/or the number of genes to keep.

Hide

```
fit <- edgeR::glmQLFit(dge, design)
qlf <- edgeR::glmQLFTest(fit, coef = "condition_nameIFNg")
tt.all <- edgeR::topTags(qlf, n = nrow(dge), sort.by = "none") # all genes
hist(tt.all$table$PValue)
```



Hide

```
tt <- edgeR::topTags(qlf, n = nrow(dge), p.value = 0.1) # genes with adj.p<0.1
tt10 <- edgeR::topTags(qlf) # just the top 10 by default
tt10
# Coefficient:  condition_nameIFNg
#                              gene_id       tx_ids  SYMBOL           GO
# ENSG00000111181.12 ENSG00000111181.12 ENST0000.... SLC6A12 GO:00033....
# ENSG00000125347.13 ENSG00000125347.13 ENST0000....    IRF1 GO:00007....
# ENSG00000137496.17 ENSG00000137496.17 ENST0000....  IL18BP GO:00055....
# ENSG00000204257.14 ENSG00000204257.14 ENST0000.... HLA-DMA GO:00022....
# ENSG00000162645.12 ENSG00000162645.12 ENST0000....    GBP2 GO:00001....
# ENSG00000145365.10 ENSG00000145365.10 ENST0000....    TIFA GO:00027....
# ENSG00000174944.8   ENSG00000174944.8 ENST0000....  P2RY14 GO:00058....
# ENSG00000204267.13 ENSG00000204267.13 ENST0000....    TAP2 GO:00019....
# ENSG00000134470.20 ENSG00000134470.20 ENST0000....  IL15RA GO:00001....
# ENSG00000100911.15 ENSG00000100911.15 ENST0000....   PSME2 GO:00005....
#                        logFC   logCPM        F       PValue          FDR
# ENSG00000111181.12 4.705110 4.271224 505.8824 4.181499e-15 8.040187e-11
# ENSG00000125347.13 5.552412 9.415299 462.2690 9.509923e-15 9.142840e-11
# ENSG00000137496.17 4.045715 7.356263 404.7655 3.174984e-14 2.034953e-10
# ENSG00000204257.14 4.062854 5.544983 378.5629 5.813342e-14 2.794473e-10
# ENSG00000162645.12 6.663163 9.603736 354.0817 1.061790e-13 3.575685e-10
# ENSG00000145365.10 5.188439 6.703715 352.1349 1.115774e-13 3.575685e-10
# ENSG00000174944.8  9.807319 5.276214 338.5682 1.588138e-13 4.104064e-10
# ENSG00000204267.13 3.452324 8.021179 332.2798 1.879065e-13 4.104064e-10
# ENSG00000134470.20 4.293508 6.548007 331.4633 1.920979e-13 4.104064e-10
# ENSG00000100911.15 3.354760 8.025720 310.9602 3.402208e-13 6.541765e-10
```

The columns in the *edgeR* result data frame are similar to the ones output by *DESeq2*. *edgeR* represents the overall expression level on the log-CPM scale rather than on the normalized count scale that *DESeq2* uses. The `F` column contains the test statistic, and the `FDR` column contains the Benjamini-Hochberg adjusted p-values.

We can compare the sets of significantly differentially expressed genes to see how the results from the two packages overlap:

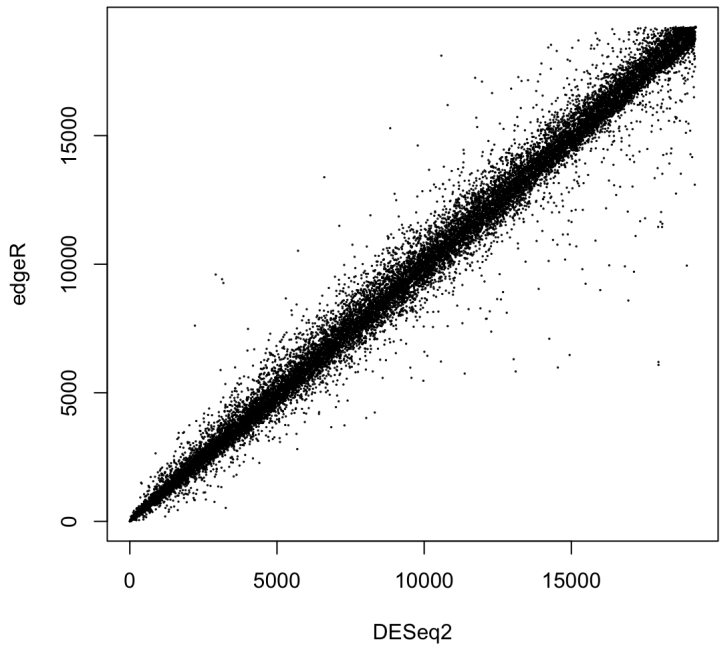
Hide

```
shared <- intersect(rownames(res), rownames(tt.all$table))
table(DESeq2 = res$padj[match(shared, rownames(res))] < 0.1,
      edgeR = tt.all$table$FDR[match(shared, rownames(tt.all$table))] < 0.1)
#        edgeR
# DESeq2  FALSE  TRUE
#   FALSE 12175    98
#   TRUE    434  6520
```

We can also compare the two result lists by the ranks:

Hide

```
plot(rank(res$pvalue[match(shared, rownames(res))]),
     rank(tt.all$table$PValue[match(shared, rownames(tt.all$table))]),
     cex = 0.1, xlab = "DESeq2", ylab = "edgeR")
```



Also with *edgeR* we can test for significance relative to a fold-change threshold, using the function `glmTreat` . Below we set the log fold-change threshold to 1 (i.e., fold change threshold equal to 2), as for *DESeq2* above.

Hide

```
treatres <- edgeR::glmTreat(fit, coef = "condition_nameIFNg", lfc = 1)
tt.treat <- edgeR::topTags(treatres, n = nrow(dge), sort.by = "none")
```

Quiz

- A feature being called DE: “Why is my expected shortlisted gene not showing up?” What possible factors can play a role in the feature being above or below the significance threshold?
- Shall I subset my DE results to the genes only detected as DE? Why?
- The results between DESeq2 and edgeR might differ. Which one is “correct”? How can you better appreciate the existing commonalities and differences in the DE analysis?
- The thought above is valid also for the process of integrating different DE analyses (i.e. using the same method, but “comparing different groups”). What can you think of to represent these results?

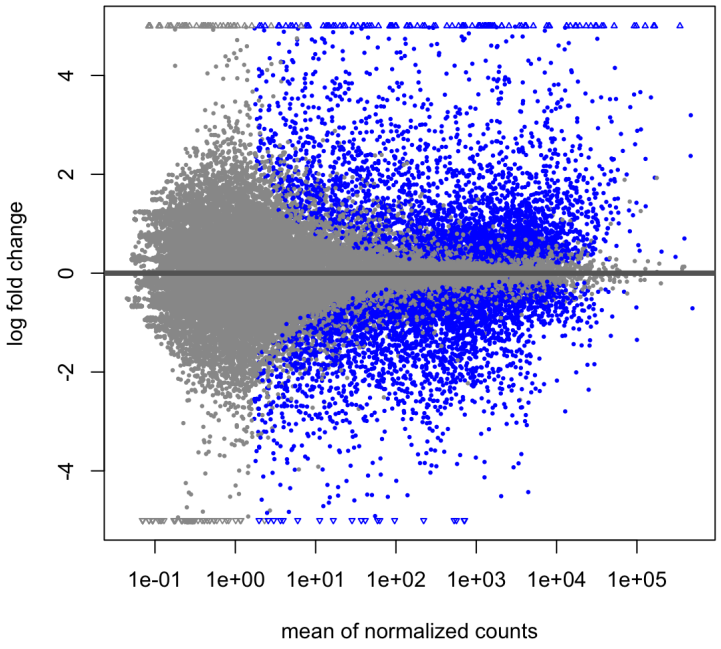
9 Plotting results

9.1 MA plot with DESeq2

An *MA-plot* (Dudoit et al. 2002) provides a useful overview for an experiment with a two-group comparison. The log2 fold change for a particular comparison is plotted on the y-axis and the average of the counts normalized by size factor is shown on the x-axis (“M” for minus, because a log ratio is equal to log minus log, and “A” for average). Each gene is represented with a dot. Genes with an adjusted *p* value below a threshold (here 0.1, the default with *DESeq2*) are shown in color

Hide

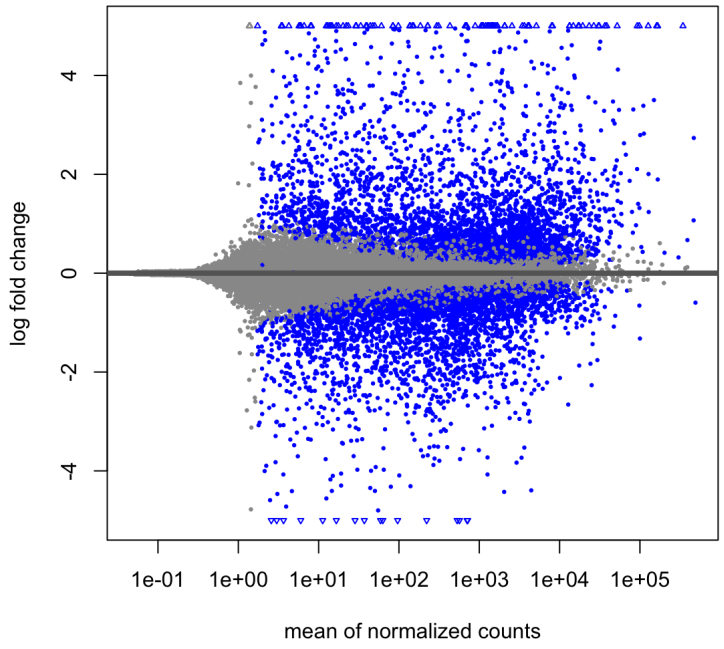
```
DESeq2::plotMA(res, ylim = c(-5, 5))
```



We see that there are many genes with low expression levels that nevertheless have large fold changes (since we are, effectively, dividing by a small number). To get more interpretable log fold changes (e.g., for ranking genes), we use the `lfcShrink` function to shrink the log2 fold changes for the comparison of IFN gamma-treated vs naive samples. There are three types of shrinkage estimators in *DESeq2*, which are covered in the vignette. Here we specify the *apeglm* method for shrinking coefficients, which is good for shrinking the noisy LFC estimates while giving low bias LFC estimates for true large differences (Zhu, Ibrahim, and Love 2019). To use apeglm we specify a coefficient from the model to shrink, either by name or number as the coefficient appears in `resultsNames(dds)` .

Hide

```
library(apeglm)
DESeq2::resultsNames(dds)
#  [1] "Intercept"                          "line_id_eiwy_1_vs_diku_1"
#  [3] "line_id_fikt_3_vs_diku_1"           "line_id_ieki_2_vs_diku_1"
#  [5] "line_id_podx_1_vs_diku_1"           "line_id_qaqx_1_vs_diku_1"
#  [7] "condition_name_IFNg_vs_naive"       "condition_name_IFNg_SL1344_vs_naive"
#  [9] "condition_name_SL1344_vs_naive"     "DESeq2_IFNg_vs_naive_log2FoldChange"
resape <- DESeq2::lfcShrink(dds, coef = "condition_name_IFNg_vs_naive", type = "apeglm")
DESeq2::plotMA(resape, ylim = c(-5, 5))
```

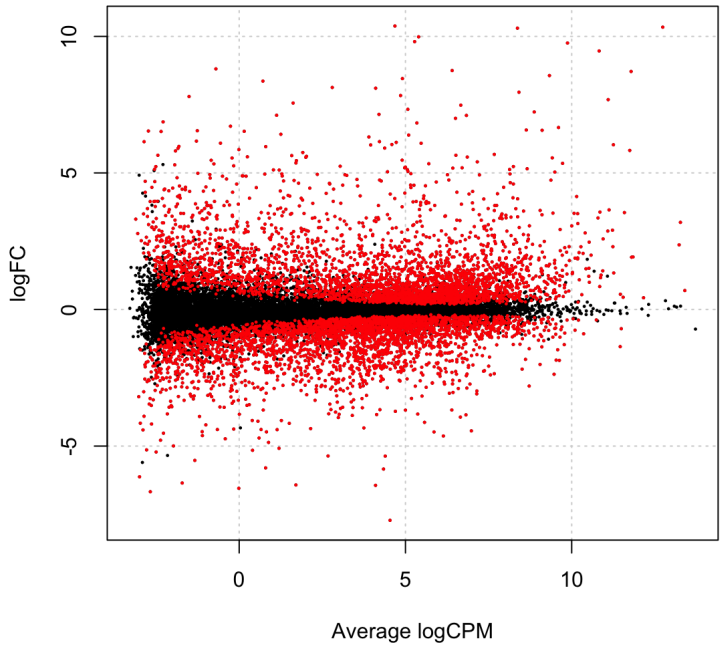


9.2 MA / Smear plot with edgeR

In *edgeR*, the MA plot is obtained via the `plotSmear` function.

Hide

```
edgeR::plotSmear(qlf, de.tags = rownames(tt$table))
```



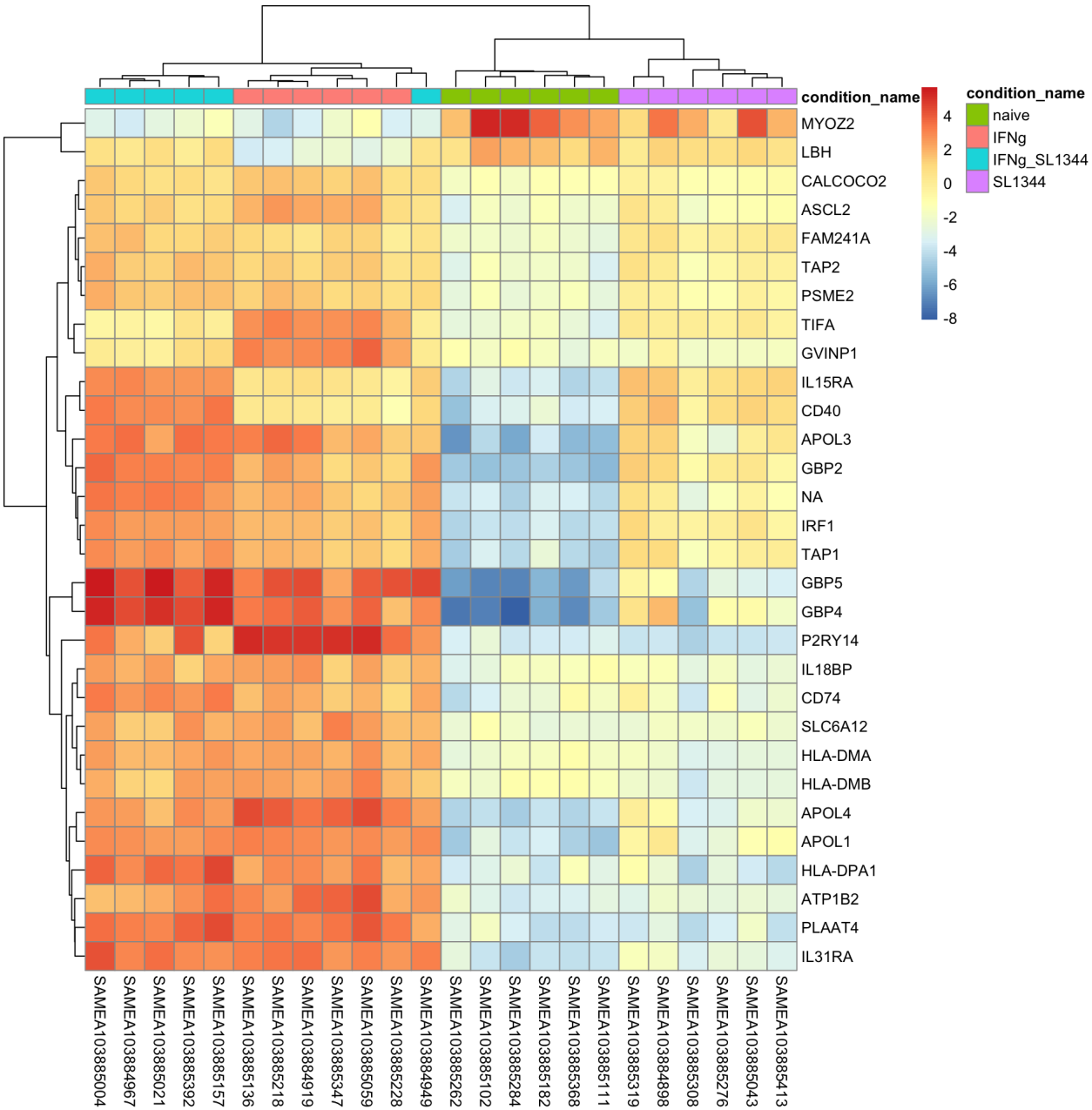
9.3 Heatmap of the most significant genes

Another way of representing the results of a differential expression analysis is to construct a heatmap of the top differentially expressed genes. A heatmap is a “color coded expression matrix”, where the rows and columns are clustered using hierarchical clustering. Typically, it should not be applied to counts, but works better with transformed values. Here we show how it can be applied to the variance-stabilized values generated above. We would expect the contrasted sample groups to cluster

separately ("by construction", since the genes were selected to be most discriminative between the groups). The heatmap will allow us to display, e.g., the variability within the groups of the differentially expressed genes. We choose the top 30 differentially expressed genes. There are many functions in R that can generate heatmaps, here we show the one from the *pheatmap (https://CRAN.R-project.org/package=pheatmap)* package.

Hide

```
library(pheatmap)
stopifnot(rownames(vsd) == rownames(res))
mat <- assay(vsd)
rownames(mat) <- rowData(vsd)$SYMBOL
mat <- mat[head(order(res$padj), 30), ]
mat <- mat - rowMeans(mat)
df <- as.data.frame(colData(vsd)[, c("condition_name"), drop = FALSE])
pheatmap(mat, annotation_col = df)
```



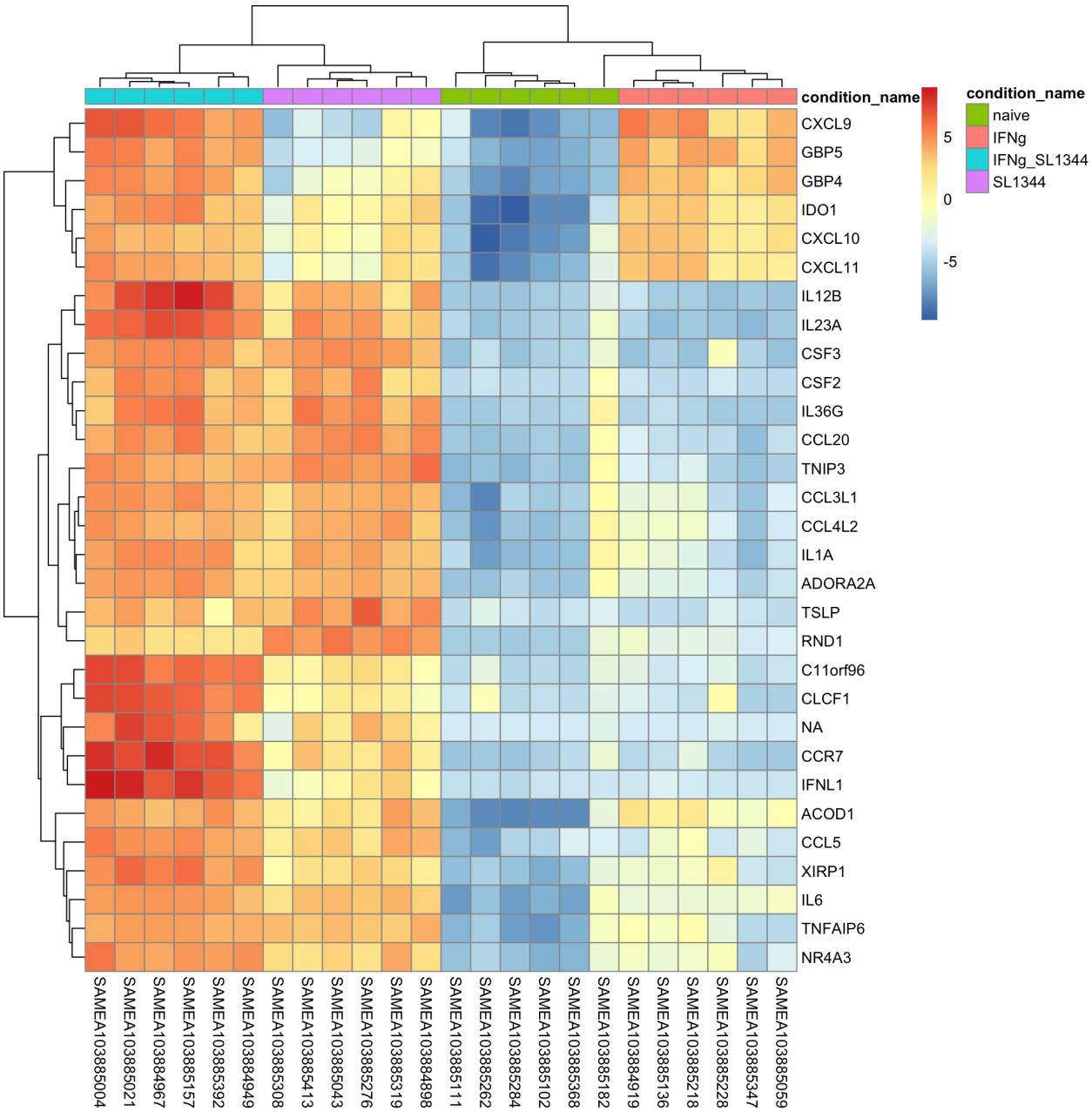
We can of course also create heatmaps for other sets of genes - for example, the collection of genes with the highest overall variance (which may or may not indicate a difference between the groups - in this particular case most of the highly variable genes show a clear difference between the groups).

Hide

```
mat <- assay(vsd)
rownames(mat) <- rowData(vsd)$SYMBOL

topVarGenes <- head(order(rowVars(mat), decreasing = TRUE), 30)

mat <- mat[topVarGenes, ]
mat <- mat - rowMeans(mat)
df <- as.data.frame(colData(vsd)[, c("condition_name"), drop = FALSE])
pheatmap(mat, annotation_col = df)
```



TODO: see the “stripes” in some genes over all the samples

9.4 Interactive visualization with iSEE

iSEE (https://bioconductor.org/packages/3.17/iSEE) is a Bioconductor package that allows interactive exploration of any data stored in a *SummarizedExperiment* container, or any class extending this (such as, e.g., the *DESeqDataSet* class, or the *SingleCellExperiment* for single-cell data). By calling the `iSEE()` function with the object as the first argument, an interactive application will be opened, in which all observed values as well as metadata columns (`rowData` and `colData`) can be explored.

Hide

```
library(iSEE)
library(iSEEu)
dds <- iSEEu::registerAveAbPatterns(dds, "log10BaseMean")
dds <- iSEEu::registerLogFCPatterns(dds, "log2FoldChange")
dds <- iSEEu::registerPValuePatterns(dds, "pvalue")
app <- iSEE(dds, initial = list(MAPlot(), VolcanoPlot(),
                                RowDataTable(), FeatureAssayPlot()))
## shiny::runApp(app)
```

9.5 Exporting results to CSV file

You can easily save the results table in a CSV file that you can then share or load with a spreadsheet program such as Excel (note, however, that Excel sometimes does funny things to gene identifiers (Zeeberg et al. 2004; Ziemann, Eren, and El-Osta 2016)). The call to *as.data.frame* is necessary to convert the *DataFrame* object to a *data.frame* object that can be processed by `write.csv`. Here, we first show how to add gene symbols to the output table, and then export just the top 100 genes for demonstration.

Hide

```
stopifnot(all(rownames(res) == rownames(dds)))
res$symbol <- rowData(dds)$SYMBOL

resOrdered <- res[order(res$padj), ]
head(resOrdered)
# log2 fold change (MLE): condition_name IFNg vs naive
# Wald test p-value: condition name IFNg vs naive
# DataFrame with 6 rows and 7 columns
#                     baseMean log2FoldChange     lfcSE      stat       pvalue
#                    <numeric>      <numeric> <numeric> <numeric>    <numeric>
# ENSG00000125347.13 30487.254        5.55915  0.218390   25.4551 6.19761e-143
# ENSG00000111181.12   687.519        4.70999  0.195911   24.0415 1.02514e-127
# ENSG00000162645.12 36639.987        6.66498  0.286603   23.2551 1.26442e-119
# ENSG00000137496.17  7118.885        4.05787  0.177049   22.9195 2.97302e-116
# ENSG00000145365.10  3642.657        5.19246  0.237141   21.8961 2.82829e-106
# ENSG00000204257.14  1906.261        4.07091  0.190575   21.3612 3.06785e-101
#                           padj      symbol
#                      <numeric> <character>
# ENSG00000125347.13 1.43971e-138        IRF1
# ENSG00000111181.12 1.19070e-123     SLC6A12
# ENSG00000162645.12 9.79081e-116        GBP2
# ENSG00000137496.17 1.72658e-112      IL18BP
# ENSG00000145365.10 1.31402e-102        TIFA
# ENSG00000204257.14  1.18777e-97     HLA-DMA

resOrderedDF <- as.data.frame(resOrdered)[seq_len(100), ]
write.table(cbind(id = rownames(resOrderedDF), resOrderedDF),
            file = "results.txt", quote = FALSE, sep = "\t",
            row.names = FALSE)
```

Quiz

- Is visualization of data and results *really that important*?
- Volcano plot or MA plot, what is “better”? Think of the information every plot type displays
- Can I always plot expression values for a single gene, or for a set of genes (i.e. independently of their DE status)?
- What can I do if I want to know more on one “novel” gene, possibly relevant with my experiment?

10 Functional analysis

In order to interpret the differential expression analysis results in terms of known gene sets, we can also apply a functional enrichment test. There are many alternatives to perform enrichment analysis in the context of R and Bioconductor. Among these, *topGO* and *clusterProfiler* are two popular options.

We can perform the analysis by following the instructions in the next chunk. In each case, the gene sets that are tested for enrichment are obtained from the Gene Ontology (http://geneontology.org/) ‘biological process’ catalog.

Hide

```
library(GeneTonic)

res$symbol <- rowData(dds)$SYMBOL

de_symbols_IFNg_vs_naive <- deseqresult2df(res, FDR = 0.05)$symbol
bg_ids <- rowData(dds)$SYMBOL[rowSums(counts(dds)) > 0]

library(topGO)
topgo_DE_macrophage_IFNg_vs_naive <- pcaExplorer::topGOtable(
    DEgenes = de_symbols_IFNg_vs_naive,
    BGgenes = bg_ids,
    ontology = "BP",
    mapping = "org.Hs.eg.db",
    geneID = "symbol",
    topTablerows = 500
)

library(clusterProfiler)
clupro_DE_macrophage_IFNg_vs_naive <- clusterProfiler::enrichGO(
    gene = de_symbols_IFNg_vs_naive,
    universe = bg_ids,
    keyType = "SYMBOL",
    OrgDb = org.Hs.eg.db,
    ont = "BP",
    pAdjustMethod = "BH",
    pvalueCutoff = 0.01,
    qvalueCutoff = 0.05,
    readable = FALSE
)
```

Hide

```
DT::datatable(topgo_DE_macrophage_IFNg_vs_naive[1:10,])
```

Show 10 entries Search:

	GO.ID	Term	Annotated	Significant	Expected	Rank in p.value_classic	p.value_elim	p.value_classic	genes
1	GO:0002181	cytoplasmic translation	158	92	45.47	58	7.7e-15	8.8e-15	DHX36,DNAJC24,DPH
2	GO:0006954	inflammatory response	768	324	221.02	50	1e-9	3e-16	A2M,ABCF1,ABHD12,A E,HMGB1,HMGB2,HNF
3	GO:0051607	defense response to virus	290	137	83.46	104	1.4e-9	1.5e-11	ACOD1,ADAR,AGBL5,A
4	GO:0045087	innate immune response	817	386	235.12	11	2.8e-8	1.6e-30	A2M,ACOD1,ACTG1,AC G,HMGB1,HMGB2,HM
5	GO:0002503	peptide antigen assembly with MHC class II protein complex	13	13	3.74	219	8.9e-8	9.2e-8	B2M,HLA-DMA,HLA-DI
6	GO:0032760	positive regulation of tumor necrosis factor production	101	54	29.07	230	1.5e-7	1.6e-7	AKAP12,ARID5A,BTK,C
7	GO:0051301	cell division	616	252	177.27	110	2.8e-7	3.4e-11	AATF,ABRAXAS2,ACTR 1,NR3C1,NUF2,NUP37

	GO.ID	Term	Annotated	Significant	Expected	Rank in p.value_classic	p.value_elim	p.value_classic	genes
8	GO:0071346	cellular response to type II interferon	112	57	32.23	293	6e-7	6.4e-7	ACOD1,ACTG1,ACTR3,
9	GO:0036297	interstrand cross-link repair	39	26	11.22	311	9.7e-7	0.000001	ATR,CENPS,CENPX,DC
10	GO:0034341	response to type II interferon	135	75	38.85	113	0.0000011	6e-11	ACOD1,ACTG1,ACTR3,

Showing 1 to 10 of 10 entries

Previous 1 Next

Hide

```
DT::datatable(as.data.frame(clupro_DE_macrophage_IFNg_vs_naive)[1:10,])
```

Show 10 **entries** **Search:**

	ID	Description	GeneRatio	BgRatio	pvalue	p.adjust	qvalue	geneID
GO:0002831	GO:0002831	regulation of response to biotic stimulus	223/4663	436/16203	2.128012987737452e-23	1.333200136817513e-19	1.094246678431311e-19	IRF1/TIFA/GBP5 G/FCN1/TRIM15
GO:0045088	GO:0045088	regulation of innate immune response	185/4663	355/16203	9.239612582401379e-21	2.525247393807805e-17	2.072639730961672e-17	IRF1/TIFA/GBP5
GO:0002460	GO:0002460	adaptive immune response based on somatic recombination of immune receptors built from immunoglobulin superfamily domains	161/4663	296/16203	1.209216629117864e-20	2.525247393807805e-17	2.072639730961672e-17	IL18BP/HLA-DM
GO:0002253	GO:0002253	activation of immune response	222/4663	455/16203	5.39268504032743e-20	8.446292944412837e-17	6.932438532105131e-17	IRF1/TIFA/CD40 DQB1/CYLD/CFE
GO:0002250	GO:0002250	adaptive immune response	218/4663	459/16203	6.374815533400103e-18	7.987643863350329e-15	6.555994501191475e-15	IRF1/IL18BP/HL G/CD46/C5/PTP
GO:0001819	GO:0001819	positive regulation of cytokine production	218/4663	460/16203	8.723966092057389e-18	8.086154182077814e-15	6.636848520090751e-15	IRF1/HLA-DPA1/ G/FCN1/CD46/IF
GO:0031349	GO:0031349	positive regulation of defense response	201/4663	415/16203	9.034809141986385e-18	8.086154182077814e-15	6.636848520090751e-15	IRF1/TIFA/GBP5 G/FCN1/TRIM15
GO:0002833	GO:0002833	positive regulation of response to biotic stimulus	159/4663	309/16203	2.392837527501052e-17	1.873890888724261e-14	1.53802780550561e-14	IRF1/TIFA/GBP5

	ID	Description	GeneRatio	BgRatio	pvalue	p.adjust	qvalue	geneID
GO:0045089	GO:0045089	positive regulation of innate immune response	149/4663	286/16203	5.886038925248471e-17	3.947873433720629e-14	3.240284239715256e-14	IRF1/TIFA/GBP5
GO:0002764	GO:0002764	immune response-regulating signaling pathway	198/4663	413/16203	6.301473956457508e-17	3.947873433720629e-14	3.240284239715256e-14	IRF1/TIFA/CD40/G/FCN1/TRIM15

Showing 1 to 10 of 10 entries

Previous 1 Next

10.1 Streamlining interpretation of results with GeneTonic

"Since it is bioinformatics", every software package can be expected to return a (slightly) different-but-similar-in-content output format. To simplify the interpretation of transcriptome datasets, the *GeneTonic* package offers an interactive application to explore in depth all the workflow results.

As a first step, we convert the output of each tool to a consolidated "standard" format, as expected by GeneTonic - this is the first step to construct a `GeneTonicList` object, as a single container to perform all operations on afterwards, be it in the app or offline by using its functionality in scripts/notebooks.

Hide

```
res_enrich_topGO <- shake_topGOtableResult(topgo_DE_macrophage_IFNg_vs_naive)
res_enrich_clupro <- shake_enrichResult(clupro_DE_macrophage_IFNg_vs_naive)

gtl_macrophage <- GeneTonicList(
    dds = dds,
    res_de = res,
    res_enrich = res_enrich_clupro,
    annotation_obj = data.frame(
        gene_id = rowData(dds)$gene_id,
        gene_name = rowData(dds)$SYMBOL
    )
)

describe_gtl(gtl_macrophage)
#  [1] "---------------------------------\n"
#  [2] "----- GeneTonicList object ------\n"
#  [3] "---------------------------------\n"
#  [4] "\n----- dds object -----\n"
#  [5] "Providing an expression object (as DESeqDataset) of 35683 features over 24 samples\n"
#  [6] "\n----- res_de object -----\n"
#  [7] "Providing a DE result object (as DESeqResults), 35683 features tested, 6072 found as DE\n"
#  [8] "Upregulated:     2848\n"
#  [9] "Downregulated:   3224\n"
# [10] "\n----- res_enrich object -----\n"
# [11] "Providing an enrichment result object, 6265 reported\n"
# [12] "\n----- annotation_obj object -----\n"
# [13] "Providing an annotation object of 35683 features with information on 2 identifier types\n"

## we can store this object as serialized file to load/share/...
saveRDS(gtl_macrophage, "gtl_macrophage.RDS")
```

After that, we would simply have to call the `GeneTonic()` function specifying the `gtl` parameter - this can also be passed at runtime

Hide

```
## and that is it!
GeneTonic(gtl = gtl_macrophage)

## or if expecting to upload at runtime... (e.g. used as a server-like app)
GeneTonic()
```

Quiz

- Genes or geneset: What should I use as a key to interpretation?
- What is the main advantage of using standardized objects vs a bunch of tables?
- "Ok, I would still like your Bioinformatics group on board for my project. What should I expect to happen?"

11 Bonus: Differential transcript expression with swish

Next, we perform a differential transcript expression analysis with *swish (https://bioconductor.org/packages/3.17/swish)*. Note that, as opposed to the workflow above, we will make use of the transcript-level abundance estimates. In addition, we need the inferential replicates. Since we ignored these when importing the data above, we will re-import it here.

Hide

```
head(coldata)
st <- tximeta(coldata = coldata, type = "salmon", dropInfReps = FALSE)
```

We can check that the inferential replicates were imported as well:

Hide

```
assayNames(st)
```

Since we are interested in comparing the `naive` and `IFNg` groups, we subset our object to these groups.

Hide

```
st <- st[, st$condition_name %in% c("naive", "IFNg")]
st$condition_name <- factor(st$condition_name, c("naive", "IFNg"))
```

Next, we run the DTE analysis with *swish*. First we'll scale the inferential replicates, followed by labeling the rows with sufficient counts for running differential expression, and then calculating the statistics.

Hide

```
library(fishpond)
st <- scaleInfReps(st, lengthCorrect = TRUE)
st <- labelKeep(st)
st <- st[mcols(st)$keep, ]
set.seed(1)
st <- swish(st, x = "condition_name", pair = "line_id", nperms = 100)
```

The results are stored in `mcols(st)`.

Hide

```
head(mcols(st))
table(mcols(st)$qvalue < 0.05)
## Most significant transcripts
tophits <- mcols(st)[order(mcols(st)$qvalue, -abs(mcols(st)$log2FC)), ]
head(tophits)
hist(mcols(st)$pvalue, col = "grey")
```

We can plot the results for some of the most significant transcripts.

Hide

```
plotInfReps(st, idx = rownames(tophits)[1],
            x = "condition_name", cov = "line_id")
plotInfReps(st, idx = rownames(tophits)[43],
            x = "condition_name", cov = "line_id")
plotInfReps(st, idx = rownames(tophits)[60],
            x = "condition_name", cov = "line_id")
plotMASwish(st, alpha = 0.05)
```

We can also use the plotting functions of *swish* to plot the inferential replicates for the top-ranked genes in the differential gene expression analysis.

Hide

```
## Summarize on the gene level
sg_inf <- summarizeToGene(st)
sg_inf <- sg_inf[, sg_inf$condition_name %in% c("naive", "IFNg")]
sg_inf$condition_name <- factor(sg_inf$condition_name, c("naive", "IFNg"))
plotInfReps(sg_inf, idx = rownames(resOrdered)[1],
            x = "condition_name", cov = "line_id")
```

12 Bonus: Differential transcript usage

Finally, we show how to run differential transcript usage analysis with *swish* and *DEXSeq (https://bioconductor.org/packages/3.17/DEXSeq)*. With *swish*, we can build upon the data imported earlier to calculate isoform proportions and perfom a permutation test based on these.

Hide

```
iso <- isoformProportions(st)
iso <- swish(iso, x = "condition_name", pair = "line_id",
             nperms = 100)
```

For *DEXSeq*, we will again re-import the data, since we would like the transcript counts to represent so called 'scaled TPM' values (similarly to what *swish* will do internally when scaling the inferential replicates, to avoid differences in transcript length being interpreted as differences in relative abundance between groups). *tximeta* can do this for us, effectively populating the 'counts' assay with TPMs, scaled up to the observed library size to be comparable in size to the actual counts. For *DEXSeq*, we further subset the transcripts to those on chromosome 18, for computational time reasons.

Hide

```
head(coldata)
st <- tximeta(coldata = coldata, type = "salmon",
              countsFromAbundance = "scaledTPM")
st <- st[, st$condition_name %in% c("naive", "IFNg")]
st$condition_name <- factor(st$condition_name, c("naive", "IFNg"))
st$sample_id <- colnames(st)
st <- st[seqnames(rowRanges(st)) == "chr18", ]

library(DEXSeq)
dxd <- DEXSeqDataSet(countData = round(assay(st, "counts")),
                     sampleData = as.data.frame(colData(st)),
                     design = ~sample + exon + condition_name:exon,
                     featureID = rowData(st)$tx_name,
                     groupID = as.character(rowData(st)$gene_id))
dxd <- estimateSizeFactors(dxd)
dxd <- estimateDispersions(dxd, quiet = TRUE)
dxd <- testForDEU(dxd, reducedModel = ~sample + exon)
dxr <- DEXSeqResults(dxd, independentFiltering = FALSE)
qval <- perGeneQValue(dxr)
dxr.g <- data.frame(gene = names(qval), qval)
head(dxr)
```

Finally, we compare the p-value ranks for the genes shared by the two analyses.

Hide

```
shared <- intersect(dxr$featureID, rownames(mcols(iso)))
plot(rank(dxr[match(shared, dxr$featureID), "pvalue"]),
     rank(mcols(iso)[shared, "pvalue"]), cex = 0.1,
     xlab = "DEXSeq", ylab = "swish")
```

Session information

It is good practice to always include a list of the software versions that were used to perform a given analysis, for reproducibility and trouble-shooting purposes. One way of achieving this is via the `sessionInfo()` function.

Hide

```
sessionInfo()
# R version 4.3.0 (2023-04-21)
# Platform: x86_64-apple-darwin20 (64-bit)
# Running under: macOS Monterey 12.6.4
#
# Matrix products: default
# BLAS:   /System/Library/Frameworks/Accelerate.framework/Versions/A/Frameworks/vecLib.framework/Versions/A/libBLAS.dylib
# LAPACK: /Library/Frameworks/R.framework/Versions/4.3-x86_64/Resources/lib/libRlapack.dylib;  LAPACK version 3.11.0
#
# 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: Europe/Berlin
# tzcode source: internal
#
# attached base packages:
# [1] stats4    stats     graphics  grDevices utils     datasets  methods
# [8] base
#
# other attached packages:
#  [1] clusterProfiler_4.8.1       topGO_2.52.0
#  [3] SparseM_1.81                GO.db_3.17.0
#  [5] graph_1.78.0                GeneTonic_2.4.0
#  [7] iSEEu_1.12.0                iSEEhex_1.2.0
#  [9] iSEE_2.12.0                 SingleCellExperiment_1.22.0
# [11] pheatmap_1.0.12             apeglm_1.22.1
# [13] pcaExplorer_2.26.1          bigmemory_4.6.1
# [15] edgeR_3.42.4                limma_3.56.2
# [17] ExploreModelMatrix_1.12.0   GenomicFeatures_1.52.1
# [19] org.Hs.eg.db_3.17.0         AnnotationDbi_1.62.1
# [21] DESeq2_1.40.2               tximeta_1.18.0
# [23] macrophage_1.16.0           knitr_1.43
# [25] BiocStyle_2.28.0            SummarizedExperiment_1.30.2
# [27] Biobase_2.60.0              GenomicRanges_1.52.0
# [29] GenomeInfoDb_1.36.1         IRanges_2.34.1
# [31] S4Vectors_0.38.1            BiocGenerics_0.46.0
# [33] MatrixGenerics_1.12.2       matrixStats_1.0.0
# [35] lubridate_1.9.2             forcats_1.0.0
# [37] stringr_1.5.0               dplyr_1.1.2
# [39] purrr_1.0.1                 readr_2.1.4
# [41] tidyr_1.3.0                 tibble_3.2.1
# [43] tidyverse_2.0.0             ggthemes_4.2.4
# [45] gapminder_1.0.0             ggplot2_3.4.2
# [47] MASS_7.3-60                 rmarkdown_2.22
#
# loaded via a namespace (and not attached):
#   [1] GSEABase_1.62.0               vroom_1.6.3
#   [3] progress_1.2.2                DT_0.28
#   [5] Biostrings_2.68.1             pagedown_0.20
#   [7] vctrs_0.6.3                   digest_0.6.32
#   [9] png_0.1-8                     shape_1.4.6
#  [11] shinyBS_0.61.1                registry_0.5-1
#  [13] ggrepel_0.9.3                 reshape2_1.4.4
#  [15] qvalue_2.32.0                 httpuv_1.6.11
#  [17] foreach_1.5.2                 withr_2.5.0
#  [19] ggfun_0.1.1                   xfun_0.39
#  [21] ellipsis_0.3.2                survival_3.5-5
#  [23] memoise_2.0.1                 xaringanExtra_0.7.0
#  [25] hexbin_1.28.3                 gson_0.1.0
#  [27] tidytree_0.4.2                GlobalOptions_0.1.2
#  [29] prettyunits_1.1.1             KEGGREST_1.40.0
#  [31] promises_1.2.0.1              httr_1.4.6
#  [33] downloader_0.4                restfulr_0.0.15
#  [35] ps_1.7.5                      rstudioapi_0.14
#  [37] shinyAce_0.4.2                DOSE_3.26.1
#  [39] miniUI_0.1.1.1                generics_0.1.3
#  [41] archive_1.1.5                 base64enc_0.1-3
#  [43] processx_3.8.1                curl_5.0.1
#  [45] zlibbioc_1.46.0               ggraph_2.1.0
#  [47] polyclip_1.10-4               ca_0.71.1
#  [49] GenomeInfoDbData_1.2.10       RBGL_1.76.0
#  [51] threejs_0.3.3                 interactiveDisplayBase_1.38.0
#  [53] xtable_1.8-4                  doParallel_1.0.17
```

```
#  [55] evaluate_0.21                S4Arrays_1.0.4
#  [57] BiocFileCache_2.8.0          hms_1.1.3
#  [59] bookdown_0.34                colorspace_2.1-0
#  [61] filelock_1.0.2               visNetwork_2.1.2
#  [63] shinyWidgets_0.7.6           magrittr_2.0.3
#  [65] Rgraphviz_2.44.0             ggtree_3.8.0
#  [67] later_1.3.1                  viridis_0.6.3
#  [69] lattice_0.21-8               NMF_0.26
#  [71] genefilter_1.82.1            shadowtext_0.1.2
#  [73] XML_3.99-0.14                cowplot_1.1.1
#  [75] pillar_1.9.0                 nlme_3.1-162
#  [77] iterators_1.0.14             gridBase_0.4-7
#  [79] compiler_4.3.0               stringi_1.7.12
#  [81] shinycssloaders_1.0.0        Category_2.66.0
#  [83] TSP_1.2-4                    dendextend_1.17.1
#  [85] GenomicAlignments_1.36.0     plyr_1.8.8
#  [87] crayon_1.5.2                 BiocIO_1.10.0
#  [89] gridGraphics_0.5-1           emdbook_1.3.12
#  [91] locfit_1.5-9.8               graphlayouts_1.0.0
#  [93] bit_4.0.5                    chromote_0.1.1
#  [95] fastmatch_1.1-3              codetools_0.2-19
#  [97] crosstalk_1.2.0              bslib_0.5.0
#  [99] GetoptLong_1.0.5             plotly_4.10.2
# [101] mime_0.12                    splines_4.3.0
# [103] circlize_0.4.15              Rcpp_1.0.10
# [105] servr_0.27                   dbplyr_2.3.2
# [107] tippy_0.1.0                  HDO.db_0.99.1
# [109] blob_1.2.4                   utf8_1.2.3
# [111] clue_0.3-64                  BiocVersion_3.17.1
# [113] AnnotationFilter_1.24.0      fs_1.6.2
# [115] backbone_2.1.2               expm_0.999-7
# [117] ggplotify_0.1.1              Matrix_1.5-4.1
# [119] tzdb_0.4.0                   tweenr_2.0.2
# [121] pkgconfig_2.0.3              tools_4.3.0
# [123] cachem_1.0.8                 RSQLite_2.3.1
# [125] viridisLite_0.4.2            DBI_1.1.3
# [127] numDeriv_2016.8-1.1          fastmap_1.1.1
# [129] scales_1.2.1                 grid_4.3.0
# [131] shinydashboard_0.7.2         Rsamtools_2.16.0
# [133] AnnotationHub_3.8.0          sass_0.4.6
# [135] patchwork_1.1.2              coda_0.19-4
# [137] BiocManager_1.30.21          fontawesome_0.5.1
# [139] farver_2.1.1                 scatterpie_0.2.1
# [141] tidygraph_1.2.3              mgcv_1.8-42
# [143] yaml_2.3.7                   renderthis_0.2.0
# [145] AnnotationForge_1.42.2       rtracklayer_1.60.0
# [147] cli_3.6.1                    webshot_0.5.5
# [149] lifecycle_1.0.3              rsconnect_0.8.29
# [151] mvtnorm_1.2-2                xaringan_0.28
# [153] tximport_1.28.0              rintrojs_0.3.2
# [155] BiocParallel_1.34.2          annotate_1.78.0
# [157] timechange_0.2.0             gtable_0.3.3
# [159] rjson_0.2.21                 ggridges_0.5.4
# [161] ape_5.7-1                    parallel_4.3.0
# [163] jsonlite_1.8.5               colourpicker_1.2.0
# [165] seriation_1.4.2              bitops_1.0-7
# [167] bigmemory.sri_0.1.6          bit64_4.0.5
# [169] assertthat_0.2.1             yulab.utils_0.0.6
# [171] heatmaply_1.4.2              bs4Dash_2.3.0
# [173] bdsmatrix_1.3-6              icons_0.2.0
# [175] jquerylib_0.1.4              highr_0.10
# [177] GOSemSim_2.26.0              shinyjs_2.1.0
# [179] lazyeval_0.2.2               shiny_1.7.4
# [181] dynamicTreeCut_1.63-1        enrichplot_1.20.0
# [183] htmltools_0.5.5              rappdirs_0.3.3
# [185] formatR_1.14                 ensembldb_2.24.0
# [187] glue_1.6.2                   XVector_0.40.0
# [189] RCurl_1.98-1.12              treeio_1.24.1
# [191] ComplexUpset_1.3.3           gridExtra_2.3
# [193] igraph_1.5.0                 R6_2.5.1
# [195] labeling_0.4.2               cluster_2.1.4
# [197] rngtools_1.5.2               bbmle_1.0.25
# [199] aplot_0.1.10                 DelayedArray_0.26.3
```

```
# [201] tidyselect_1.2.0              vipor_0.4.5
# [203] ProtGenerics_1.32.0           GOstats_2.66.0
# [205] ggforce_0.4.1                 xml2_1.3.4
# [207] emo_0.0.0.9000                munsell_0.5.0
# [209] data.table_1.14.8             websocket_1.4.1
# [211] fgsea_1.26.0                  htmlwidgets_1.6.2
# [213] ComplexHeatmap_2.16.0         RColorBrewer_1.1-3
# [215] biomaRt_2.56.1                rlang_1.1.1
# [217] uuid_1.1-0                    fansi_1.0.4
```

References

Alasoo, Kaur, Julia Rodrigues, Subhankar Mukhopadhyay, Andrew J Knights, Alice L Mann, Kousik Kundu, HIPSCI Consortium, Christine Hale, Gordon Dougan, and Daniel J Gaffney. 2018. “Shared Genetic Effects on Chromatin and Gene Expression Indicate a Role for Enhancer Priming in Immune Response.” *Nat. Genet.* 50 (3): 424–31.

Dudoit, Sandrine, Yee H. Yang, Matthew J. Callow, and Terence P. Speed. 2002. “Statistical methods for identifying differentially expressed genes in replicated cDNA microarray experiments.” In *Statistica Sinica*, 111–39. http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.117.9702 (http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.117.9702).

Hardcastle, Thomas, and Krystyna Kelly. 2010. “baySeq: Empirical Bayesian methods for identifying differential expression in sequence count data.” *BMC Bioinformatics* 11 (1): 422+. https://doi.org/10.1186/1471-2105-11-422 (https://doi.org/10.1186/1471-2105-11-422).

Law, Charity W., Yunshun Chen, Wei Shi, and Gordon K. Smyth. 2014. “Voom: precision weights unlock linear model analysis tools for RNA-seq read counts.” *Genome Biology* 15 (2): R29+. https://doi.org/10.1186/gb-2014-15-2-r29 (https://doi.org/10.1186/gb-2014-15-2-r29).

Leng, N., J. A. Dawson, J. A. Thomson, V. Ruotti, A. I. Rissman, B. M. G. Smits, J. D. Haag, M. N. Gould, R. M. Stewart, and C. Kendziorski. 2013. “EBSeq: an empirical Bayes hierarchical model for inference in RNA-seq experiments.” *Bioinformatics* 29 (8): 1035–43. https://doi.org/10.1093/bioinformatics/btt087 (https://doi.org/10.1093/bioinformatics/btt087).

Love, Michael I., Simon Anders, Vladislav Kim, and Wolfgang Huber. 2015. “RNA-Seq Workflow: Gene-Level Exploratory Analysis and Differential Expression.” *F1000Research*, October. https://doi.org/10.12688/f1000research.7035.1 (https://doi.org/10.12688/f1000research.7035.1).

Love, Michael I., Wolfgang Huber, and Simon Anders. 2014. “Moderated estimation of fold change and dispersion for RNA-seq data with DESeq2.” *Genome Biology* 15 (12): 550+. https://doi.org/10.1186/s13059-014-0550-8 (https://doi.org/10.1186/s13059-014-0550-8).

Patro, Rob, Geet Duggal, Michael I Love, Rafael A Irizarry, and Carl Kingsford. 2017. “Salmon Provides Fast and Bias-Aware Quantification of Transcript Expression.” *Nat. Methods*.

Robinson, M. D., D. J. McCarthy, and G. K. Smyth. 2009. “edgeR: a Bioconductor package for differential expression analysis of digital gene expression data.” *Bioinformatics* 26 (1): 139–40. https://doi.org/10.1093/bioinformatics/btp616 (https://doi.org/10.1093/bioinformatics/btp616).

Soneson, Charlotte, Michael I. Love, and Mark Robinson. 2015. “Differential analyses for RNA-seq: transcript-level estimates improve gene-level inferences.” *F1000Research* 4. https://doi.org/10.12688/f1000research.7563.1 (https://doi.org/10.12688/f1000research.7563.1).

Wu, Hao, Chi Wang, and Zhijin Wu. 2013. “A new shrinkage estimator for dispersion improves differential expression detection in RNA-seq data.” *Biostatistics* 14 (2): 232–43. https://doi.org/10.1093/biostatistics/kxs033 (https://doi.org/10.1093/biostatistics/kxs033).

Zeeberg, Barry R, Joseph Riss, David W Kane, Kimberly J Bussey, Edward Uchio, W Marston Linehan, J Carl Barrett, and John N Weinstein. 2004. “Mistaken Identifiers: Gene Name Errors Can Be Introduced Inadvertently When Using Excel in Bioinformatics.” *BMC Bioinformatics* 5: 80.

Zhu, Anqi, Joseph G Ibrahim, and Michael I Love. 2019. “Heavy-Tailed Prior Distributions for Sequence Count Data: Removing the Noise and Preserving Large Differences.” *Bioinformatics* 35: 2084–92.

Ziemann, Mark, Yotam Eren, and Assam El-Osta. 2016. “Gene Name Errors Are Widespread in the Scientific Literature.” *Genome Biol.* 17 (1): 1–3.