DE_analysis_edgeR_script.R

Ronan Harrington

18/8/2020

Contents

Introduction

Written for https://github.com/rnnh/bioinfo-notebook

In `DE_analysis_edgeR_script.R`, differential expression (DE) analysis is carried out on RNA-seq data, using the `R` programming language with the `{edgeR}` library. This document will provide commentary for this `R` script, and will discuss general concepts used in `R`. Note that each DE analysis needs to be tailored to the specific research questions that need to be addressed: a one-size-fits-all approach does not apply to DE analysis scripts.

This page features the `R` code from the DE analysis edgeR script broken into different chunks (sections of code) with commentary. It also features the output displayed in the `R` console when each section is run: sections with code output feature two hash signs (`##`).

This script is used to analyse data from an RNA-seq experiment, featuring reads from *Saccharomyces cerevisiae* grown in chemostats with different stress conditions. These stresses are: high temperature, low pH, anaerobic stress, and osmotic pressure (KCl). A standard chemostat was used as a control. This experiment was carried out as part of the CHASSY project, and the data is available through NCBI BioProject PRJNA531619.

The aim of this script is to determine which genes are most differentially expressed under each stress condition when compared to the control sample. Before doing this, we want to create a function that will test the relative standard deviation between replicates in each condition. This is done to eliminate genes with large differences between replicates from the analysis.

Gene count tables were generated from these data using the `fastq-dump_to_featureCounts.sh` script. These gene count tables were then combined into featCounts_S_cere_20200331.csv using the `combining_featCount_tables.sh` script. The experimental design is summarised in design_table.csv.

Loading and formatting data

The `R` packages that are required for this script are loaded at the beginning using the `library()` function. An `R` package or library is a collection of functions, complied code and sample data. In this case, the packages loaded are edgeR and limma. Commands from the `{limma}` package are not used in this script, it is loaded as it is a dependency for `{edgeR}`.

```
# Loading required libraries
library(limma)
library(edgeR)
```

Once the packages are loaded, an `if (){} else {}` statement is used to set the working directory for the script to its current location. This is done so that the data can be loaded using relative file pathways.

```
# Changing working directory
# Setting the working directory to the directory which contains this script
if (exists("RStudio.Version")){
  setwd(dirname(rstudioapi::getActiveDocumentContext()$path))
} else {
  setwd(getSrcDirectory()[1])
}
```

After setting the working directory to the script location, the feature count table can be loaded using the `read.csv()` command. The command reads a given comma-separated values (CSV) file. The arrow operator (`<-`) is used to assign the output from this command to *object* `counts.df`. An object can contain different types of data, and can be manipulated with different functions/commands within `R`. Once the feature count table is assigned to the `counts.df` object, the `head()` command is used on this object to print its first six rows to the console.

```
# Reading in the feature count file as "counts.df"
counts.df <- read.csv("../data/featCounts_S_cere_20200331.csv")

# Printing the start of the counts.df object in R...
head(counts.df)
```

```
##      Geneid SRR8933532 SRR8933534 SRR8933509 SRR8933530 SRR8933511 SRR8933533
## 1   YAL068C          7          6          8          7         24          4
## 2 YAL067W-A          0          0          0          0          0          0
## 3   YAL067C          0          0          0          0          0          0
## 4   YAL065C          0          0          1          1          1          1
## 5 YAL064W-B         23         28         24         19         32         24
## 6 YAL064C-A          0          0          0          0          0          0
##   SRR8933537 SRR8933506 SRR8933531 SRR8933538 SRR8933512 SRR8933510 SRR8933535
## 1          6         11         12          3         28          5          5
## 2          0          0          0          0          0          0          0
## 3          0          0          0          0          0          0          0
## 4          3          3          2          3          0          0          2
## 5         11         17         11         19         18         25         17
## 6          0          0          0          0          0          0          0
##   SRR8933536 SRR8933539
## 1          6          5
## 2          0          0
## 3          0          0
## 4          1          1
## 5         15         15
## 6          0          0
```

We want the names of the rows in `count.df` to be the gene names, from the `Geneid` column. The `Geneid` column from `count.df` is selected using `counts.df$Geneid`. The names of the rows in `count.df` are selected using `rowname(counts.df)`. The arrow operator (`<-`) is then used to assign `Geneid` to the row names of `counts.df`. Once this is done, `Geneid` is no longer needed as a separate column in `counts.df`. The arrow operator (`<-`) is used to assign `NULL` to `counts.df$Geneid`, which results in this column being removed from `counts.df`. The `head()` command is used again to print the start of the `counts.df` object after making these changes.

```r
# Using the "Geneid" column to set the rownames
rownames(counts.df) <- counts.df$Geneid

# Removing the "Geneid" column
counts.df$Geneid <- NULL

# Printing the start of the counts.df object in R...
head(counts.df)
```

```
##             SRR8933532 SRR8933534 SRR8933509 SRR8933530 SRR8933511 SRR8933533
## YAL068C              7          6          8          7         24          4
## YAL067W-A            0          0          0          0          0          0
## YAL067C              0          0          0          0          0          0
## YAL065C              0          0          1          1          1          1
## YAL064W-B           23         28         24         19         32         24
## YAL064C-A            0          0          0          0          0          0
##             SRR8933537 SRR8933506 SRR8933531 SRR8933538 SRR8933512 SRR8933510
## YAL068C              6         11         12          3         28          5
## YAL067W-A            0          0          0          0          0          0
## YAL067C              0          0          0          0          0          0
## YAL065C              3          3          2          3          0          0
## YAL064W-B           11         17         11         19         18         25
## YAL064C-A            0          0          0          0          0          0
##             SRR8933535 SRR8933536 SRR8933539
## YAL068C              5          6          5
## YAL067W-A            0          0          0
## YAL067C              0          0          0
## YAL065C              2          1          1
## YAL064W-B           17         15         15
## YAL064C-A            0          0          0
```

Next, the design table is loaded. This gives a summary of the design of the experiment used to generate this data. This table to assigned to the object `design.df` using the arrow operator (`<-`) with the `read.csv()` command. The start of this object is then printed using `head()`.

```r
# Reading in the design table as "design.df"
design.df <- read.csv("../data/design_table.csv", fileEncoding="UTF-8-BOM")

# Printing the start of the design.df object in R...
```

```
print(design.df)
```

```
##           run         name        condition
## 1  SRR8933532 SCEhightemp3        high_temp
## 2  SRR8933534 SCEhightemp1        high_temp
## 3  SRR8933509      SCEkcl3 osmotic_pressure
## 4  SRR8933530    SCElowPH2           low_pH
## 5  SRR8933511    SCEanaer2        anaerobic
## 6  SRR8933533 SCEhightemp2        high_temp
## 7  SRR8933537     SCEstan1         standard
## 8  SRR8933506    SCEanaer3        anaerobic
## 9  SRR8933531    SCElowPH1           low_pH
## 10 SRR8933538      SCEkcl1 osmotic_pressure
## 11 SRR8933512    SCEanaer1        anaerobic
## 12 SRR8933510      SCEkcl2 osmotic_pressure
## 13 SRR8933535     SCEstan3         standard
## 14 SRR8933536     SCEstan2         standard
## 15 SRR8933539    SCElowPH3           low_pH
```

Based on this design table, the gene counts can be subset into different conditions. This is done using square brackets after the gene counts object: `counts.df[]`. In these square brackets, conditions are given to select specific rows and columns: `counts.df[rows, columns]`. The column names in `counts.df` correspond to the run IDs in the design table: `design.df$run`. To select the samples used in the standard chemostat, the run IDs for the standard condition are copied from the design table. These are then combined into a list using the `c()` command, separated by commas: `c("SRR8933535", "SRR8933536", "SRR8933537")`. This list is pasted square brackets after `counts.df`. A comma is placed before this list, meaning the run IDs will be used to select columns: `counts.df[,c("SRR8933535", "SRR8933536", "SRR8933537")]`. As no condition is given to select rows, all the rows are returned. The arrow operator (`<-`) is used to assign this subset to the object `counts_standard.df`. The same process is used to create the rest of the subsets: `counts_anaerobic.df`, `counts_high_temp.df`, `counts_low_pH.df` and `counts_pressure.df`.

```
# Subsetting gene counts according to experimental condition
counts_standard.df  <- counts.df[,c("SRR8933535", "SRR8933536", "SRR8933537")]
counts_anaerobic.df <- counts.df[,c("SRR8933506", "SRR8933511", "SRR8933512")]
counts_high_temp.df <- counts.df[,c("SRR8933532", "SRR8933533", "SRR8933534")]
counts_low_pH.df    <- counts.df[,c("SRR8933530", "SRR8933531", "SRR8933539")]
counts_pressure.df  <- counts.df[,c("SRR8933509", "SRR8933510", "SRR8933538")]
```

The structure of the gene counts object and its subsets can be displayed using the `str()` command. Note that the total gene counts set has 15 variables (i.e. samples), its 5 subsets have 3 variables each, and all these objects have the same number of observations (i.e. genes).

```
# Printing the structure of the gene counts set and subsets
str(counts.df)
```

```
## 'data.frame':    6420 obs. of  15 variables:
##  $ SRR8933532: int  7 0 0 0 23 0 0 0 26 1124 ...
##  $ SRR8933534: int  6 0 0 0 28 0 0 0 25 1045 ...
##  $ SRR8933509: int  8 0 0 1 24 0 0 0 30 556 ...
##  $ SRR8933530: int  7 0 0 1 19 0 0 0 53 1135 ...
##  $ SRR8933511: int  24 0 0 1 32 0 0 0 66 252 ...
##  $ SRR8933533: int  4 0 0 1 24 0 0 0 30 1081 ...
##  $ SRR8933537: int  6 0 0 3 11 0 0 0 33 1288 ...
##  $ SRR8933506: int  11 0 0 3 17 0 0 0 30 235 ...
##  $ SRR8933531: int  12 0 0 2 11 0 0 0 31 388 ...
##  $ SRR8933538: int  3 0 0 3 19 0 0 0 49 569 ...
##  $ SRR8933512: int  28 0 0 0 18 0 0 0 61 209 ...
##  $ SRR8933510: int  5 0 0 0 25 0 0 0 49 567 ...
##  $ SRR8933535: int  5 0 0 2 17 0 0 0 34 350 ...
##  $ SRR8933536: int  6 0 0 1 15 0 0 0 32 474 ...
##  $ SRR8933539: int  5 0 0 1 15 0 0 0 64 1236 ...
```

```
str(counts_standard.df)
```

```
## 'data.frame':    6420 obs. of  3 variables:
##  $ SRR8933535: int  5 0 0 2 17 0 0 0 34 350 ...
##  $ SRR8933536: int  6 0 0 1 15 0 0 0 32 474 ...
##  $ SRR8933537: int  6 0 0 3 11 0 0 0 33 1288 ...
```

```
str(counts_anaerobic.df)
```

```
## 'data.frame':    6420 obs. of  3 variables:
##  $ SRR8933506: int  11 0 0 3 17 0 0 0 30 235 ...
##  $ SRR8933511: int  24 0 0 1 32 0 0 0 66 252 ...
##  $ SRR8933512: int  28 0 0 0 18 0 0 0 61 209 ...
```

```
str(counts_high_temp.df)
```

```
## 'data.frame':    6420 obs. of  3 variables:
##  $ SRR8933532: int  7 0 0 0 23 0 0 0 26 1124 ...
##  $ SRR8933533: int  4 0 0 1 24 0 0 0 30 1081 ...
##  $ SRR8933534: int  6 0 0 0 28 0 0 0 25 1045 ...
```

```
str(counts_low_pH.df)
```

```
## 'data.frame':    6420 obs. of  3 variables:
##  $ SRR8933530: int  7 0 0 1 19 0 0 0 53 1135 ...
##  $ SRR8933531: int  12 0 0 2 11 0 0 0 31 388 ...
##  $ SRR8933539: int  5 0 0 1 15 0 0 0 64 1236 ...
```

```
str(counts_pressure.df)
```

```
## 'data.frame':    6420 obs. of  3 variables:
##  $ SRR8933509: int  8 0 0 1 24 0 0 0 30 556 ...
##  $ SRR8933510: int  5 0 0 0 25 0 0 0 49 567 ...
##  $ SRR8933538: int  3 0 0 3 19 0 0 0 49 569 ...
```

Testing relative standard deviation

Next, a new function is defined: `RSD.test()`. This function tests whether the relative standard deviation (RSD) is less than or equal to one for each row in a data frame. A data frame is a type of object in `R`: in the output from the `str()` commands above, all of the gene count sets and subsets were listed as `'data.frame'` objects. The function `RSD.test()` adds the result to a new variable in the data frame called "RSD.test". For a given row, if `data.frame$RSD.test` is `TRUE`, that row has an RSD less than or equal to one, i.e. RSD <= 1. If `data.frame$RSD.test` is `FALSE`, that row has an RSD outside of this range. These `TRUE`/`FALSE` values are Boolean values, known as *logical* values in `R`. Once this function is defined, it is applied to the gene counts subsets, adding a new column called `RSD.test` to each object.

```
# Defining function "RSD.test()"
RSD.test <- function(dataframe){
  # This function tests whether the relative standard deviation (RSD) is less
  # than or equal to one for each row in a data frame.
  # It adds the result to a new variable in the data frame called "RSD.test".
  # For a given row, if data.frame$RSD.test is TRUE, that row has an RSD less
```

```r
  # than or equal to one, i.e. RSD <= 1.
  # If data.frame$RSD.test is FALSE, that row has an RSD outside of this range.
  RSD_tests = dataframe[,1]
  for (row_index in 1:nrow(dataframe)){
    row = as.numeric(dataframe[row_index,])
    RSD = sd(row) / mean(row)
    RSD_tests[row_index] = RSD <= 1 || is.na(RSD)
  }
  dataframe$RSD.test <- as.factor(RSD_tests)
  levels(dataframe$RSD.test) <- c(FALSE, TRUE)
  return(dataframe)
}

# Applying RSD.test() to gene count subsets
counts_standard.df  <- RSD.test(counts_standard.df)
counts_anaerobic.df <- RSD.test(counts_anaerobic.df)
counts_high_temp.df <- RSD.test(counts_high_temp.df)
counts_low_pH.df    <- RSD.test(counts_low_pH.df)
counts_pressure.df  <- RSD.test(counts_pressure.df)
```

After applying `RSD.test()` to each gene count subset, the structure of each subset is printed again using the `str()` command. Note that each subset now has a fourth variable (column).

```r
# Printing the structure of the gene counts subsets
str(counts_standard.df)
```

```
## 'data.frame':    6420 obs. of  4 variables:
##  $ SRR8933535: int  5 0 0 2 17 0 0 0 34 350 ...
##  $ SRR8933536: int  6 0 0 1 15 0 0 0 32 474 ...
##  $ SRR8933537: int  6 0 0 3 11 0 0 0 33 1288 ...
##  $ RSD.test  : Factor w/ 2 levels "FALSE","TRUE": 2 2 2 2 2 2 2 2 2 2 ...
```

```r
str(counts_anaerobic.df)
```

```
## 'data.frame':    6420 obs. of  4 variables:
##  $ SRR8933506: int  11 0 0 3 17 0 0 0 30 235 ...
##  $ SRR8933511: int  24 0 0 1 32 0 0 0 66 252 ...
##  $ SRR8933512: int  28 0 0 0 18 0 0 0 61 209 ...
##  $ RSD.test  : Factor w/ 2 levels "FALSE","TRUE": 2 2 2 1 2 2 2 2 2 2 ...
```

```r
str(counts_high_temp.df)
```

```
## 'data.frame':    6420 obs. of  4 variables:
##  $ SRR8933532: int  7 0 0 0 23 0 0 0 26 1124 ...
##  $ SRR8933533: int  4 0 0 1 24 0 0 0 30 1081 ...
##  $ SRR8933534: int  6 0 0 0 28 0 0 0 25 1045 ...
##  $ RSD.test  : Factor w/ 2 levels "FALSE","TRUE": 2 2 2 1 2 2 2 2 2 2 ...
```

```r
str(counts_low_pH.df)
```

```
## 'data.frame':    6420 obs. of  4 variables:
##  $ SRR8933530: int  7 0 0 1 19 0 0 0 53 1135 ...
##  $ SRR8933531: int  12 0 0 2 11 0 0 0 31 388 ...
##  $ SRR8933539: int  5 0 0 1 15 0 0 0 64 1236 ...
##  $ RSD.test  : Factor w/ 2 levels "FALSE","TRUE": 2 2 2 2 2 2 2 2 2 2 ...
```

```r
str(counts_pressure.df)
```

```
## 'data.frame':    6420 obs. of  4 variables:
##  $ SRR8933509: int  8 0 0 1 24 0 0 0 30 556 ...
##  $ SRR8933510: int  5 0 0 0 25 0 0 0 49 567 ...
##  $ SRR8933538: int  3 0 0 3 19 0 0 0 49 569 ...
##  $ RSD.test  : Factor w/ 2 levels "FALSE","TRUE": 2 2 2 1 2 2 2 2 2 2 ...
```

We do not want to analyse any genes which failed this RSD test, i.e. any gene for which `RSD.test == FALSE`. These genes can be selected using square brackets after the name of one of the gene count objects, with `which(counts_standard.df$RSD.test == FALSE)` used as the condition to select rows which failed the RSD test. This condition is used to select rows, not columns; so it is placed before the comma in the square brackets after the name of the gene counts object. As we only want the gene names, this can be wrapped in the `rownames()` command, resulting in only the names of the failed rows being returned. The arrow operator (`<-`) is used to assign the names of the genes which failed the RSD test in the standard subset to `RSD_failed_genes`. This process is repeated for the rest of the gene count subsets, using the `append()` command to add the names of the failed genes from each subset to the existing `RSD_failed_genes` object.

At this point, we can assume that there will be duplicate gene names, as it is unlikely that all of the genes that failed the RSD test only failed under a single condition each. To eliminate duplicate gene names, the `unique()` command is applied to the `RSD_failed_genes` object, removing duplicate gene names. The `length()` command is then used on the `RSD_failed_genes` object to give the number of genes which failed the RSD test.

```r
# Creating list of genes which failed RSD test
RSD_failed_genes <- rownames(counts_standard.df[
  which(counts_standard.df$RSD.test == FALSE),])
RSD_failed_genes <- append(RSD_failed_genes, rownames(counts_anaerobic.df[
  which(counts_anaerobic.df$RSD.test == FALSE),]))
RSD_failed_genes <- append(RSD_failed_genes, rownames(counts_high_temp.df[
  which(counts_high_temp.df$RSD.test == FALSE),]))
RSD_failed_genes <- append(RSD_failed_genes, rownames(counts_low_pH.df[
  which(counts_low_pH.df$RSD.test == FALSE),]))
RSD_failed_genes <- append(RSD_failed_genes, rownames(counts_pressure.df[
  which(counts_pressure.df$RSD.test == FALSE),]))
RSD_failed_genes <- unique(RSD_failed_genes)
length(RSD_failed_genes)
```

```
## [1] 373
```

We can see that 373 of the 6420 genes failed the RSD test.

We can select against these genes using the `which()` command. In this command, `!rownames(counts.df) %in% RSD_failed_genes` is given as the selection condition. The exclamation mark (`!`) at the start of the condition is a *NOT* operator. In this case, we want to select the gene names that are *not* in the RSD failed genes list. This can be written in square brackets before the comma as the condition to select rows in `counts.df`: this will result in selecting the genes which did not fail the RSD test under any of the experimental conditions. The arrow operator (`<-`) is used to assign these filtered gene counts to the object `filtered_counts.df`. The structure of this object is then displayed using `str()`.

```r
# Filtering gene counts
filtered_counts.df <- counts.df[
  which(!rownames(counts.df) %in% RSD_failed_genes),]

# Printing the structure of the filtered gene counts
str(filtered_counts.df)
```

```
## 'data.frame':    6047 obs. of  15 variables:
##  $ SRR8933532: int  7 0 0 23 0 0 0 26 1124 1877 ...
##  $ SRR8933534: int  6 0 0 28 0 0 0 25 1045 2280 ...
##  $ SRR8933509: int  8 0 0 24 0 0 0 30 556 618 ...
##  $ SRR8933530: int  7 0 0 19 0 0 0 53 1135 2327 ...
##  $ SRR8933511: int  24 0 0 32 0 0 0 66 252 207 ...
##  $ SRR8933533: int  4 0 0 24 0 0 0 30 1081 2217 ...
##  $ SRR8933537: int  6 0 0 11 0 0 0 33 1288 1583 ...
##  $ SRR8933506: int  11 0 0 17 0 0 0 30 235 175 ...
##  $ SRR8933531: int  12 0 0 11 0 0 0 31 388 1935 ...
##  $ SRR8933538: int  3 0 0 19 0 0 0 49 569 748 ...
##  $ SRR8933512: int  28 0 0 18 0 0 0 61 209 203 ...
##  $ SRR8933510: int  5 0 0 25 0 0 0 49 567 657 ...
##  $ SRR8933535: int  5 0 0 17 0 0 0 34 350 1762 ...
##  $ SRR8933536: int  6 0 0 15 0 0 0 32 474 1647 ...
##  $ SRR8933539: int  5 0 0 15 0 0 0 64 1236 2400 ...
```

If the gene counts were correctly filtered, then the number of genes (rows) in the gene count table minus the number of genes that failed the RSD test should be equal to the number of genes (rows) in the filtered gene count table. We write this in `R` using the `nrow()` command to count the number of rows in the gene count tables (before and after filtering), the `length()` command to count the number of failed genes in the `RSD_failed_genes` object, and the minus operator (`-`) to subtract one value from another. The double equals sign operator (`==`) can be used to verify the expression: it will return `TRUE` if the values on either side of the operator are equal, and `FALSE` if the values are not equal.

```r
# Checking that gene counts were correctly filtered
nrow(counts.df) - length(RSD_failed_genes) == nrow(filtered_counts.df)
```

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

As this statement returned `TRUE`, the gene filtering based on the `RSD.test()` results worked as expected. We now have an object of gene counts in which none of the genes have an RSD greater than one across any experimental condition: `filtered_counts.df`.

At this point, many of the objects that have been created are no longer of use. They can be removed from the `R` *envrionment* using the `rm()` command. The `R` environment is the set of objects that have been called/created and are ready to be used within `R`. Keeping this environment clean and free of unnecessary will make it easier to navigate the available data. It is also good practice for avoiding mistakes: if there are many objects in the environment with similar names, the wrong objects can easily be selected by accident.

```r
# Removing redundant objects from R environment
rm(counts_anaerobic.df, counts_high_temp.df, counts_low_pH.df,
   counts_pressure.df, counts_standard.df, counts.df, RSD_failed_genes)
```

Creating a DGEList object

Now that our gene counts are correctly formatted and filtered, we can begin the DE analysing using `{edgeR}`. The first `edgeR` command we need to use is `DGEList()`. This command creates a "DGEList" *class* object. An object's class describes how the data in the object is structured, and determines which functions can be applied to it. Using `DGEList()` with `filtered_counts.df`, an DGEList object of the gene counts is created. This DGEList object is assigned to `counts.DGEList` using the arrow operator (`<-`).

```r
# Creating a DGEList object using the filtered gene counts
counts.DGEList <- DGEList(counts = filtered_counts.df,
                          genes = rownames(filtered_counts.df))
```

Now that the DGEList object is created, we need to add information about it. The experimental condition of each sample can be added as *grouping* information in the DGEList object. Printing the design table once more, we can see that the information we want to add is in the variable `design.df$condition`.

```r
# Printing the design table
print(design.df)
```

```
##           run         name        condition
## 1  SRR8933532 SCEhightemp3        high_temp
## 2  SRR8933534 SCEhightemp1        high_temp
## 3  SRR8933509      SCEkcl3 osmotic_pressure
## 4  SRR8933530    SCElowPH2           low_pH
## 5  SRR8933511    SCEanaer2        anaerobic
## 6  SRR8933533 SCEhightemp2        high_temp
## 7  SRR8933537     SCEstan1         standard
## 8  SRR8933506    SCEanaer3        anaerobic
## 9  SRR8933531    SCElowPH1           low_pH
## 10 SRR8933538      SCEkcl1 osmotic_pressure
## 11 SRR8933512    SCEanaer1        anaerobic
## 12 SRR8933510      SCEkcl2 osmotic_pressure
## 13 SRR8933535     SCEstan3         standard
## 14 SRR8933536     SCEstan2         standard
## 15 SRR8933539    SCElowPH3           low_pH
```

Before assigning this variable to the grouping variable in the DGEList object, we need to confirm that the column names in the `filtered_counts.df` object (used to create the DGEList object) match the order of the `run` variable in the design table. We can do this using the double equals sign operator (`==`), which will return `TRUE` each time the run variable in `design.df` matches a column name in `filtered_counts.df`.

```r
# Confirming samples are in the same order in the gene counts and design table
summary(colnames(filtered_counts.df) == design.df$run)
```

```
##    Mode    TRUE
## logical      15
```

Now that we have confirmed that these values match up, we can add the grouping information directly from `design.df$condition` to the DGEList object `counts.DGEList`. We use the `as.factor()` command to ensure that this variable is being added as a factor to `counts.DGEList`. This `as.factor()` command is used to create *categorical* variables; these variables describe different categories. In this case, the categories are the experimental conditions.

```r
# Add grouping information to DGEList object
counts.DGEList$samples$group <- as.factor(design.df$condition)
```

Now that the grouping information is added, we can examine `counts.DGEList` using the `print()` command.

```r
# Printing counts.DGEList
counts.DGEList
```

```
## An object of class "DGEList"
## $counts
##           SRR8933532 SRR8933534 SRR8933509 SRR8933530 SRR8933511 SRR8933533
## YAL068C            7          6          8          7         24          4
## YAL067W-A          0          0          0          0          0          0
## YAL067C            0          0          0          0          0          0
## YAL064W-B         23         28         24         19         32         24
## YAL064C-A          0          0          0          0          0          0
##           SRR8933537 SRR8933506 SRR8933531 SRR8933538 SRR8933512 SRR8933510
## YAL068C            6         11         12          3         28          5
## YAL067W-A          0           0           0           0           0           0
## YAL067C            0           0           0           0           0           0
## YAL064W-B         11          17          11          19          18          25
## YAL064C-A          0           0           0           0           0           0
##           SRR8933535 SRR8933536 SRR8933539
## YAL068C            5           6           5
## YAL067W-A          0           0           0
## YAL067C            0           0           0
## YAL064W-B         17          15          15
## YAL064C-A          0           0           0
## 6042 more rows ...
##
## $samples
##                        group lib.size norm.factors
## SRR8933532         high_temp  7251197            1
## SRR8933534         high_temp  7591623            1
## SRR8933509 osmotic_pressure  7286624            1
## SRR8933530            low_pH  6690543            1
## SRR8933511         anaerobic  7552294            1
## 10 more rows ...
##
## $genes
##               genes
## YAL068C     YAL068C
## YAL067W-A YAL067W-A
## YAL067C     YAL067C
## YAL064W-B YAL064W-B
## YAL064C-A YAL064C-A
## 6042 more rows ...
```

We can see that this object contains all the information we have described so far: the gene names, gene counts, and sample grouping. Here is a summary of `counts.DGEList` created using the `dim()` command. This command gives the dimensions of an `R` object. In this case the dimensions are the number of genes (`x`) and the number of samples (`y`).

```r
# Summary of the counts.DGEList object: number of genes, number of samples
dim(counts.DGEList)
```

```
## [1] 6047   15
```

The object `counts.DGEList` contains 6047 genes, across 15 samples.

Filtering lowly expressed genes

At this point, we want to filter out lowly expressed genes, as they will not be useful for DE analysis. We can do this using the `{edgeR}` command `filterByExpr()`. This command will create a set of logical (`TRUE`/`FALSE`) values that can be used to filter lowly expressed genes. This output is assigned to `counts.keep` using the arrow (`<-`) operator.

```r
# Creating an object to filter genes with low expression
counts.keep <- filterByExpr(counts.DGEList)
summary(counts.keep)
```

```
##    Mode   FALSE    TRUE
## logical     261    5786
```

We can use `counts.keep` to filter lowly expressed genes using square brackets after `counts.DGEList`. This object is written before the first comma to specify that it is used as a condition to select rows. We will use a new argument with this command: `keep.lib.sizes = FALSE`. This is added so that the library sizes (number of RNA-seq reads) of the samples are recalculated after filtering. This filtered `counts.DGEList` object is assigned to the same name using the arrow (`<-`) operator. After filtering, `dim()` is used again to display the dimensions of `counts.DGEList`.

```r
# Filtering lowly expressed genes
counts.DGEList <- counts.DGEList[counts.keep, , keep.lib.sizes = FALSE]
dim(counts.DGEList)
```

```
## [1] 5786   15
```

We can see that there are now 5786 genes in `counts.DGEList`. This can be confirmed with another logical statement: that the number of `TRUE` values in `counts.keep` is equal to the number of genes/rows in `counts.DGEList`. If this is the case, the `R` console will return `TRUE`.

```r
# Confirming that the number of genes in counts.DGEList is the same as the
# number of TRUE values in counts.keep
length(counts.keep[counts.keep == TRUE]) == dim(counts.DGEList)[1]
```

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

```
[1] TRUE
```

We can now remove `counts.keep`, as it will not be used again.

```r
# Removing counts.keep
rm(counts.keep)
```

Normalising samples

We now need to normalise the library sizes between the samples in `counts.DGEList`. This is done to minimise bias towards highly expressed genes. The `{edgeR}` function `calcNormFactors()` normalises the library sizes by finding a set of scaling factors for the library sizes that minimizes the log-fold changes between the samples for most genes. This function can be assigned directly to the `counts.DGEList` object. By printing the normalisation factors for the samples (`counts.DGEList$samples$norm.factors`) before and after assigning `calcNormFactors()` to the DGEList object, we can see the effect of this command.

```r
# Printing the normalisation factors for the libraries
counts.DGEList$samples$norm.factors
```

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

```r
# Calculating normalisation factors and applying them to counts.DGEList
counts.DGEList <- calcNormFactors(counts.DGEList)
counts.DGEList$samples$norm.factors
```

```
##  [1] 1.0260074 1.0962181 0.9884704 0.9641469 1.0908860 1.0833048 0.9228469
##  [8] 0.9657677 1.0155710 0.9607328 1.0797312 0.9803494 0.9144696 0.9374429
## [15] 1.0004370
```

Estimating dispersion

Next, we need to apply `estimateDisp()` to `counts.DGEList`. This command is used to estimate common dispersion and tagwise dispersion. In this case, the tags are synonymous with genes. By estimating gene dispersion, we are estimating the relative variability of true expression levels between replicates. We can use `design.df` to quickly specify a *design matrix* for this command. A design matrix specifies the experimental design; in this case, the stress condition of each sample. While this may not be necessary in this case as we have already added grouping information, specifying a design matrix is good practice. This design matrix is created by assigning `design.df$condition` to an object `condtion_` using the arrow operator (`<-`); and then using the function `model.matrix()` on this `condition_` object. This is used int he `estimateDisp()` command as the `design =` argument. The `estimateDisp()` function can be applied directly to `counts.DGEList` using the arrow operator (`<-`).

```
# Estimating common dispersion and tagwise dispersion
condition_ <- design.df$condition
counts.DGEList <- estimateDisp(counts.DGEList,
                               design = model.matrix(~condition_))
```

We can see that this command has added a lot of information to `counts.DGEList`.

```
counts.DGEList
```

```
## An object of class "DGEList"
## $counts
##            SRR8933532 SRR8933534 SRR8933509 SRR8933530 SRR8933511 SRR8933533
## YAL068C             7          6          8          7         24          4
## YAL064W-B          23         28         24         19         32         24
## YAL063C            26         25         30         53         66         30
## YAL062W          1124       1045        556       1135        252       1081
## YAL061W          1877       2280        618       2327        207       2217
##            SRR8933537 SRR8933506 SRR8933531 SRR8933538 SRR8933512 SRR8933510
## YAL068C             6         11         12          3         28          5
## YAL064W-B          11         17         11         19         18         25
## YAL063C            33         30         31         49         61         49
## YAL062W          1288        235        388        569        209        567
## YAL061W          1583        175       1935        748        203        657
##            SRR8933535 SRR8933536 SRR8933539
## YAL068C             5          6          5
## YAL064W-B          17         15         15
## YAL063C            34         32         64
## YAL062W           350        474       1236
## YAL061W          1762       1647       2400
## 5781 more rows ...
##
## $samples
##                        group lib.size norm.factors
## SRR8933532         high_temp  7250773    1.0260074
## SRR8933534         high_temp  7591195    1.0962181
## SRR8933509 osmotic_pressure  7286231    0.9884704
## SRR8933530            low_pH  6690227    0.9641469
## SRR8933511         anaerobic  7551939    1.0908860
## 10 more rows ...
##
## $genes
##                genes
## YAL068C      YAL068C
## YAL064W-B YAL064W-B
## YAL063C      YAL063C
## YAL062W      YAL062W
## YAL061W      YAL061W
## 5781 more rows ...
##
## $design
##   (Intercept) condition_high_temp condition_low_pH condition_osmotic_pressure
## 1           1                   1                0                          0
## 2           1                   1                0                          0
## 3           1                   0                0                          1
## 4           1                   0                1                          0
## 5           1                   0                0                          0
##   condition_standard
## 1                  0
## 2                  0
## 3                  0
## 4                  0
## 5                  0
## 10 more rows ...
##
## $common.dispersion
## [1] 0.01862555
##
## $trended.dispersion
## [1] 0.03268122 0.03139264 0.02839365 0.01389939 0.01830658
## 5781 more elements ...
##
## $tagwise.dispersion
## [1] 0.03645635 0.01763052 0.05423434 0.15491398 0.01442296
## 5781 more elements ...
##
## $AveLogCPM
## [1] 0.5931716 1.5663444 2.5472675 6.5992064 7.5615585
## 5781 more elements ...
##
## $trend.method
## [1] "locfit"
##
## $prior.df
## [1] 3.31197
##
## $prior.n
## [1] 0.331197
##
## $span
## [1] 0.3197199
```

Pairwise testing

In this DE analysis, we want to compare samples for each stress condition to the standard condition samples. This is essentially a series of pairwise tests: standard verus high temperature, standard versus low pH, etc. To carry out these pairwise tests, we need to use the `{edgeR}` command `exactTest()`. This command is used to compute genewise exact tests for differences in the means between two groups. To view the groups that we want to compare, we can display the `condition_` object.

```
condition_
```

```
##  [1] "high_temp"         "high_temp"         "osmotic_pressure" "low_pH"
##  [5] "anaerobic"         "high_temp"         "standard"         "anaerobic"
##  [9] "low_pH"            "osmotic_pressure" "anaerobic"         "osmotic_pressure"
## [13] "standard"         "standard"         "low_pH"
```

Each stress condition can be copied and used as a pair with `standard`. These will be combined into a list and used with the `pair` argument in `exactTest()`. For example, `pair = c("standard", "low_pH")`. The function `exactTest()` also takes the DGEList object as an argument. The results of each `exactTest()` will be assigned to their own `.DGEExact` object using the arrow operator (`<-`).

```r
# Exact tests for differences between experimental conditions
std_anaerobic.DGEExact <- exactTest(counts.DGEList, pair = c("standard",
                                                             "anaerobic"))
std_salt.DGEExact <- exactTest(counts.DGEList, pair = c("standard",
                                                        "osmotic_pressure"))
std_temp.DGEExact <- exactTest(counts.DGEList, pair = c("standard",
                                                        "high_temp"))
std_pH.DGEExact <- exactTest(counts.DGEList, pair = c("standard",
                                                      "low_pH"))
```

Now that the appropriate pairwise tests have been carried out, we can extract the most differentially expressed genes for each stress condition. This is done using the `{edgeR}` command `topTags()`. This command will be used on each `.DGEExact` object, and the output will be assigned to `.topTags` objects using the arrow operator (`<-`).

```r
# Extracting most differentially expressed genes from exact tests
std_anaerobic.topTags <- topTags(std_anaerobic.DGEExact)
std_salt.topTags <- topTags(std_salt.DGEExact)
std_temp.topTags <- topTags(std_temp.DGEExact)
std_pH.topTags <- topTags(std_pH.DGEExact)
```

The most differentially expressed genes for each stress condition can now be displayed by printing the `.topTags` objects.

```r
# Printing the most differentially expressed genes
std_anaerobic.topTags
```

```
## Comparison of groups:  anaerobic-standard
##           genes     logFC    logCPM        PValue           FDR
## YNL117W YNL117W -6.028560  7.148428  0.000000e+00  0.000000e+00
## YLR413W YLR413W  4.620027  7.177812 1.997785e-304 5.779592e-301
## YLR174W YLR174W -5.976283  9.329009 9.416531e-244 1.816135e-240
## YOR348C YOR348C -6.783899  8.496686 1.781586e-236 2.577064e-233
## YDR046C YDR046C  6.194348  5.553198 5.928128e-232 6.860030e-229
## YPR151C YPR151C -4.815656  7.283962 2.880903e-203 2.778150e-200
## YNL237W YNL237W -4.194697  6.043569 1.041952e-192 8.612474e-190
## YOR011W YOR011W  3.384736  7.022691 6.044665e-188 4.371804e-185
## YGR067C YGR067C -4.281513  8.334530 7.202445e-185 4.630372e-182
## YKL217W YKL217W -4.731764 11.591840 4.649898e-177 2.690431e-174
```

```r
std_salt.topTags
```

```
## Comparison of groups:  osmotic_pressure-standard
##           genes     logFC    logCPM        PValue           FDR
## YJL107C YJL107C  4.413018  6.594644 4.554075e-261 2.634988e-257
## YJL108C YJL108C  4.382795  6.171726 8.880071e-231 2.569005e-227
## IRT1       IRT1 -5.736304  6.956820 3.965577e-177 7.648277e-174
## YPR192W YPR192W -6.975775  5.509654 4.347055e-152 6.288014e-149
## YKL187C YKL187C -5.381030  8.913363 3.776022e-140 4.369613e-137
## YDL022W YDL022W  2.992260 10.029918 4.602418e-107 4.438265e-104
## YNL040W YNL040W -2.345908  6.150795  6.425644e-91  5.311254e-88
## YER062C YER062C  2.396279  7.181993  1.457771e-90  1.054333e-87
## YJL045W YJL045W -3.296462  7.107009  3.010210e-86  1.935231e-83
## YBL075C YBL075C -3.354665 10.419402  4.360822e-82  2.523171e-79
```

```
std_temp.topTags
```

```
## Comparison of groups:  high_temp-standard
##           genes      logFC    logCPM       PValue          FDR
## YBR001C YBR001C  1.2374088  7.195621 1.493799e-36 8.643121e-33
## YLL055W YLL055W -1.3095651  6.698696 4.976411e-25 1.439676e-21
## YDR248C YDR248C -1.0331499  6.307828 1.124151e-23 2.168113e-20
## YJR094C YJR094C  2.1131855  5.676445 1.800442e-21 2.604339e-18
## YLR213C YLR213C  1.1728166  4.704520 1.470135e-20 1.701241e-17
## YOR100C YOR100C  1.1351013  6.336745 5.348729e-19 5.157958e-16
## YNL134C YNL134C -0.9586595  8.268649 1.048559e-18 8.667092e-16
## YCL025C YCL025C  1.2841592  6.188404 2.528173e-17 1.828501e-14
## YML008C YML008C -1.0519014 10.337190 4.096060e-16 2.633311e-13
## YLR136C YLR136C  1.3967615  5.900904 5.044282e-16 2.918622e-13
```

```
std_pH.topTags
```

```
## Comparison of groups:  low_pH-standard
##               genes      logFC   logCPM       PValue          FDR
## YBR004C     YBR004C -0.8886308 5.501671 1.900082e-08 7.226154e-05
## RDN37-2     RDN37-2 -1.1673219 4.492904 2.497806e-08 7.226154e-05
## YMR321C     YMR321C -2.6547077 1.505010 2.068363e-07 3.731196e-04
## YHL010C     YHL010C  0.9556848 5.656570 3.760650e-07 3.731196e-04
## YHL036W     YHL036W -0.7956401 6.006900 5.232517e-07 3.731196e-04
## YDL169C     YDL169C  1.1783452 6.789186 5.379658e-07 3.731196e-04
## YFL066C     YFL066C  0.7116988 6.279952 6.463222e-07 3.731196e-04
## YMR221C     YMR221C -0.8561199 6.315953 7.224923e-07 3.731196e-04
## YIL045W     YIL045W  0.7441531 7.393643 7.557152e-07 3.731196e-04
## YGR174W-A YGR174W-A  0.8523619 4.982261 8.223465e-07 3.731196e-04
```

Session information

The final command that this script runs is `sessionInfo()`. This command prints version information about `R`, the operating system and attached or loaded packages. All of the output on this page was created in one session. Providing information about the exact versions of the software used will make it easier to replicate results.

```
sessionInfo()
```

```
## R version 4.0.2 (2020-06-22)
## Platform: x86_64-w64-mingw32/x64 (64-bit)
## Running under: Windows 10 x64 (build 17763)
##
## Matrix products: default
##
## locale:
## [1] LC_COLLATE=English_Ireland.1252  LC_CTYPE=English_Ireland.1252
## [3] LC_MONETARY=English_Ireland.1252 LC_NUMERIC=C
## [5] LC_TIME=English_Ireland.1252
##
## attached base packages:
## [1] stats     graphics  grDevices utils     datasets  methods   base
```

```
##
## other attached packages:
## [1] edgeR_3.30.3 limma_3.44.3
##
## loaded via a namespace (and not attached):
##  [1] Rcpp_1.0.5      locfit_1.5-9.4  lattice_0.20-41 digest_0.6.25
##  [5] grid_4.0.2      magrittr_1.5    evaluate_0.14   rlang_0.4.7
##  [9] stringi_1.4.6   rmarkdown_2.3   splines_4.0.2   tools_4.0.2
## [13] stringr_1.4.0   xfun_0.15       yaml_2.2.1      compiler_4.0.2
## [17] htmltools_0.5.0 knitr_1.29
```

Versions of this document

- PDF version on GitHub
- Online version on GitHub Pages
- Markdown version on GitHub

See also

- DE_analysis_edgeR_script.R on GitHub
- featCounts_S_cere_20200331.csv
- design_table.csv
- fastq-dump_to_featureCounts.sh
- combining_featCount_tables.py
- Relative file pathways
- The CHASSY project
- R project
- edgeR
- limma

References

- R operators
- edgeR User's Guide
- NCBI BioProject PRJNA531619