Example workflow for regsplice package

Lukas M. Weber

15 June 2016

Package version: regsplice 0.99.0

Contents

1 Introduction

The regsplice package implements statistical methods for the detection of differential exon usage (differential splicing) in RNA sequencing (RNA-seq) and exon microarray data sets.

The regsplice methods are based on the use of the lasso (L1-regularization) to improve the power of standard generalized linear models, with fast runtimes compared to other leading approaches. The statistical methodology and comparisons of performance with other methods are described in our paper:

> Title of paper and link to bioRxiv preprint when available here.

1.1 Example workflow

This vignette demonstrates an example workflow for the regsplice package using a small simulated RNA-seq data set.

There are two options for running regsplice: you can run a complete workflow in one step using the wrapper function regsplice(); or you can run the individual functions for each step in sequence, which provides additional flexibility and insight into the methodology. Both options are demonstrated below.

1.2 Data set

The data set used for the example workflow consists of exon-level read counts for a subset of 100 genes from a simulated human RNA-seq data set, consisting of 6 biological samples, with 3 samples in each of 2 conditions.

The original data set is from the paper:

> Soneson et al. (2016), *Isoform prefiltering improves performance of count-based methods for analysis of differential transcript usage*, Genome Biology, available here

Original data files from this paper, containing the simulated RNA-seq reads (FASTQ and BAM files), are available from ArrayExpress at accession code E-MTAB-3766.

Exon bin counts were generated with the Python counting scripts provided with the DEXSeq package, using the option to exclude exons from overlapping genes instead of aggregating them into multi-gene complexes (see Soneson et al. 2016, Supplementary Material).

For this example workflow, we have selected a subset of the first 100 genes from this simulated data set. The exon-level read counts and the true differential splicing status labels for these 100 genes are saved in the text files vignette_counts.txt and vignette_truth.txt in the extdata/ directory in the regsplice package source code.

1.3 Exon microarray data

The regsplice methods are designed to work with both RNA-seq read counts and exon microarray intensities. If you are using exon microarray data, simply provide a matrix or data frame of exon microarray intensities instead of RNA-seq read counts for the counts input argument. The name of the argument will still be counts, regardless of the type of input data.

The steps in the workflow are the same as shown for RNA-seq data below. See ?regsplice for the required format for the input matrix or data frame.

Note that exon microarray intensities should be log2-transformed, which can either be done externally (for example with limma-voom) or with the voom_norm = TRUE argument. In addition, the filtering parameters filter_n1 and filter_n2 need to be adjusted carefully when using exon microarray data; filtering may also be disabled with filter = FALSE.

2 Workflow

2.1 Load data and create condition vector

Load the vignette example data file, which contains simulated RNA-seq read counts for 100 genes across 6 biological samples. Extract the table of counts and the gene IDs from the raw data.

Then create the condition vector, which specifies the experimental conditions or treatment groups for each biological sample.

```
# load data
file_counts <- system.file("extdata/vignette_counts.txt", package = "regsplice")
data <- read.table(file_counts, header = TRUE, sep = "\t", stringsAsFactors = FALSE)

head(data)
##                   exon sample1 sample2 sample3 sample4 sample5 sample6
## 1 ENSG00000000003:001     576     506     526     643     482     826
## 2 ENSG00000000003:002     141     122     126     157     121     191
## 3 ENSG00000000003:003     123     102     106     133      99     156
## 4 ENSG00000000003:004      86      76      77      98      72     112
## 5 ENSG00000000003:005      97      83      87     113      76     126
## 6 ENSG00000000003:006     133     107     116     155      97     170

dim(data)
## [1] 3191    7

# extract counts and gene IDs from raw data
counts <- data[, 2:7]
gene <- sapply(strsplit(data$exon, ":"), function(s) s[[1]])

head(gene, 6)
## [1] "ENSG00000000003" "ENSG00000000003" "ENSG00000000003" "ENSG00000000003"
## [5] "ENSG00000000003" "ENSG00000000003"

# create condition vector
condition <- rep(c("untreated", "treated"), each = 3)

condition
## [1] "untreated" "untreated" "untreated" "treated"   "treated"   "treated"
```

2.2 Run workflow with wrapper function

The regsplice() wrapper function runs the complete workflow with one command.

The results of a regsplice analysis consist of a set of multiple testing adjusted p-values (Benjamini-Hochberg false discovery rates, FDR) quantifying the statistical evidence for differential exon usage (DEU) for each gene. The adjusted p-values are used to rank the genes in the data set according to their evidence for DEU, and an appropriate significance threshold can be used to generate a list of genes with statistically significant evidence for DEU.

The wrapper function also returns gene names, raw p-values, likelihood ratio (LR) test statistics, and degrees of freedom of the LR tests.

The required inputs for the regsplice() wrapper function are counts (matrix or data frame of RNA-seq read counts or exon microarray intensities), gene (vector of gene IDs), and condition (vector of experimental conditions for each sample).

See ?regsplice or help(regsplice) for additional details, including other available inputs and options. Note that the progress bar does not display well in the vignette; it will display on a single line on your screen.

```
library(regsplice)

res <- regsplice(counts, gene, condition)
## removed 936 exon(s) with zero counts
## removed 1 remaining single-exon gene(s)
## removed 240 low-count exon(s)
## removed 4 remaining single-exon gene(s)
## Fitting regularized (lasso) models...
##
  |
  |                                                                 |   0%
  |
  |================================                                 |  50%
  |
  |=================================================================| 100%
## Fitting null models...

str(res)
## List of 5
##  $ gene    : chr [1:81] "ENSG00000000003" "ENSG00000000419" "ENSG00000000457" "ENSG00000000460" ...
##  $ p_vals  : num [1:81] 1 0 1 1 1 1 0 0 0 1 ...
##  $ p_adj   : num [1:81] 1 0 1 1 1 1 0 0 0 1 ...
##  $ LR_stats: num [1:81] NA 334795 NA NA NA ...
##  $ df_tests: int [1:81] NA 17 NA NA NA NA 32 5 3 NA ...
```

2.3 Run workflow with functions for individual steps

Alternatively, you can run the individual functions for each step in the regsplice workflow in sequence, which provides additional flexibility and insight into the statistical methodology. The steps are described below.

2.3.1 Prepare data

The first step consists of pre-processing the input data and preparing it into the format required by other functions in the regsplice pipeline. This is done with the prepare_data() function.

Inputs are a table of RNA-seq read counts or exon microarray intensities (counts) and a vector of gene IDs (gene). The vector gene must have length equal to the number of rows in counts; i.e. one entry for each exon, with repeated entries for multiple exons within the same gene. The repeated entries are used to determine gene length.

The prepare_data() function removes exons (rows) with zero counts in all biological samples (columns); splits the count or intensity table into a list of sub-tables (data frames), one for each gene; and removes any remaining single-exon genes. The output is a list of data frames, where each data frame contains the RNA-seq read counts or exon microarray intensities for one gene.

For more details, see ?prepare_data.

```
library(regsplice)

Y <- prepare_data(counts, gene)
```

```
## removed 936 exon(s) with zero counts
## removed 1 remaining single-exon gene(s)

# length equals the number of genes
length(Y)
## [1] 87
```

2.3.2 Filter low-count exons

Next, we filter low-count exons with `filter_exons()`.

The optional arguments `n1` and `n2` control the amount of filtering. Default values are provided; however these may not be optimal for some experimental designs, so you should consider how much filtering is appropriate for your data set. Any remaining single-exon genes are also removed.

For more details, see `?filter_exons`.

```
Y <- filter_exons(Y)
## removed 240 low-count exon(s)
## removed 4 remaining single-exon gene(s)

# length equals the number of genes
length(Y)
## [1] 81
```

2.3.3 'voom' weights

In many cases, power to detect differential exon usage can be improved by using exon-level precision weights.

The `voom_weights()` function uses `limma-voom` to calculate exon-level precision weights; as well as an optional log2-counts per million continuous transformation and scale normalization across samples, which in some cases may further improve power.

For more information, see the following paper, which introduced `voom`; or the limma User's Guide available on Bioconductor.

- Law et al. (2014), *voom: precision weights unlock linear model analysis tools for RNA-seq read counts*, Genome Biology, available here

By default, `voom_weights()` only returns the weights (see `?voom_weights` for further information). If you wish to use the transformed and normalized data, set the argument `norm = TRUE`.

Note that `voom` assumes that exons (rows) with zero or low counts have already been removed, so this step should be done after filtering.

Exon microarray intensities should be log2-transformed prior to model fitting; so if you are using exon microarray data, either transform the data externally (for example with `voom`), or set the argument `norm = TRUE`.

For more details, see `?voom_weights`.

```
out_voom <- voom_weights(Y, condition)

weights <- out_voom$weights
```

2.3.4 Optional: Create design matrices

The function `create_design_matrix()` creates the model design matrix for each gene. This function is called automatically by the model fitting functions, so you do not need to run it directly. Here, we demonstrate how it works for a single gene and show an example design matrix, to provide insight into the statistical methodology.

The design matrix includes main effect terms for each exon and each sample, and interaction terms between the exons and conditions.

Note that the design matrix does not include main effect terms for the conditions, since these are absorbed into the main effect terms for the samples. In addition, the design matrix does not include an intercept column, since it is simpler to let the model fitting functions add an intercept term later.

For more details, see `?create_design_matrix`.

```
# gene with 3 exons
# 4 biological samples; 2 samples in each of 2 conditions
design_example <- create_design_matrix(condition = rep(c(0, 1), each = 2), n_exons = 3)

design_example
##    Exon2 Exon3 Samp2 Samp3 Samp4 Exon2:Cond1 Exon3:Cond1
## 1      0     0     0     0     0           0           0
## 2      1     0     0     0     0           0           0
## 3      0     1     0     0     0           0           0
## 4      0     0     1     0     0           0           0
## 5      1     0     1     0     0           0           0
## 6      0     1     1     0     0           0           0
## 7      0     0     0     1     0           0           0
## 8      1     0     0     1     0           1           0
## 9      0     1     0     1     0           0           1
## 10     0     0     0     0     1           0           0
## 11     1     0     0     0     1           1           0
## 12     0     1     0     0     1           0           1
```

2.3.5 Fit models

There are three model fitting functions:

- `fit_models_reg()` fits regularized (lasso) models containing an optimal subset of exon:condition interaction terms for each gene. The model fitting procedure penalizes the interaction terms only, so that the main effect terms for exons and samples are always included. This ensures that the null model is nested, allowing likelihood ratio tests to be calculated.
- `fit_models_null()` fits the null models, which do not contain any interaction terms.
- `fit_models_GLM()` fits full GLMs, which contain all exon:condition interaction terms for each gene.

The fitting functions fit models for all genes in the data set. The functions are parallelized using `BiocParallel` for faster runtime. For `fit_models_reg()`, the default number of processor cores is 8, or the maximum available if less than 8. For `fit_models_null()` and `fit_models_GLM()`, the default is one core, since these functions are already extremely fast.

Note that in this example, we have used a single core for `fit_models_reg()`, in order to simplify testing and compilation of this vignette. We have also used `suppressWarnings()` to hide warning messages related to the small number of observations per gene in this data set.

The `weights` argument is optional. If it is not included (default value is `NULL`), exons are weighted equally.

For more details, see `?fit_models_reg`, `?fit_models_null`, or `?fit_models_GLM`.

```
# fit regularized models
fit_reg <- suppressWarnings(fit_models_reg(Y, condition, weights, n_cores = 1))
## Fitting regularized (lasso) models...

# fit null models
fit_null <- fit_models_null(Y, condition, weights)
## Fitting null models...

# fit full GLMs (not required if 'when_null_selected = "ones"' in next step)
fit_GLM <- fit_models_GLM(Y, condition, weights)
## Fitting full GLMs...
```

2.3.6 Calculate likelihood ratio tests

The function `LR_tests()` calculates likelihood ratio (LR) tests between the fitted models and null models.

If the fitted regularized (lasso) model contains at least one exon:condition interaction term, the LR test compares the lasso model against the nested null model. However, if the lasso model contains zero interaction terms, then the lasso and null models are identical, so the LR test cannot be calculated. The `when_null_selected` argument lets the user choose what to do in these cases: either set p-values equal to 1 (`when_null_selected = "ones"`); or calculate a LR test using the full GLM containing all exon:condition interaction terms (`when_null_selected = "GLM"`), which reduces power due to the larger number of terms, but allows the evidence for differential exon usage among these genes to be distinguished. You can also return NAs for these genes (`when_null_selected = "NA"`).

The default option is `when_null_selected = "ones"`. This simply calls all these genes non-significant, which in most cases is sufficient since we are more interested in genes with strong evidence for differential exon usage. However, if it is important to rank the low-evidence genes in your data set, use the `when_null_selected = GLM` option. If `when_null_selected = "ones"` or `when_null_selected = "NA"`, the full GLM fitted models are not required, so you can set `fit_GLM = NULL` (the default).

The result is a list containing gene names, raw p-values, multiple testing adjusted p-values (Benjamini-Hochberg false discovery rates, FDR), LR test statistics, and degrees of freedom of the LR tests for each gene.

For more details, see `?LR_tests`.

```
res_no_wrapper <- LR_tests(fit_reg, fit_null)

str(res_no_wrapper)
## List of 5
##  $ gene    : chr [1:81] "ENSG00000000003" "ENSG00000000419" "ENSG00000000457" "ENSG00000000460" ...
##  $ p_vals  : num [1:81] 1 0 1 1 1 1 0 0 0 1 ...
##  $ p_adj   : num [1:81] 1 0 1 1 1 1 0 0 0 1 ...
##  $ LR_stats: num [1:81] NA 334795 NA NA NA ...
##  $ df_tests: int [1:81] NA 17 NA NA NA NA 32 8 3 NA ...
```

2.4 Summary table of results

After the results have been calculated (either with the wrapper function or the individual functions), the function `summary_table()` can be used to display the results in a more readable format.

The results are displayed as a data frame of the top `n` most highly significant genes, ranked according to either the false discovery rate (FDR) or raw p-values, up to a specified significance threshold.

Set `n = Inf` to display results for all genes up to the specified significance threshold. Set `n = Inf` and `threshold = 1` to display results for all genes in the data set.

Note that we have used `n = 10` for display purposes in this vignette. Many of the p-values and adjusted p-values (FDRs) are exactly equal to zero in this example, due to the strong differential splicing signal in the simulated data set.

For more details, see `?summary_table`.

```
summary_table(res, n = 10)
##                gene p_vals p_adj     LR_stats df_tests
## 1  ENSG00000000419      0     0 3.347946e+05       17
## 2  ENSG00000001084      0     0 1.772690e+05       32
## 3  ENSG00000001167      0     0 2.309612e+06        5
## 4  ENSG00000001460      0     0 2.091053e+03        3
## 5  ENSG00000001497      0     0 1.326379e+05       17
## 6  ENSG00000001631      0     0 6.260249e+04       28
## 7  ENSG00000002834      0     0 1.050164e+08       12
## 8  ENSG00000003756      0     0 1.132831e+06       58
## 9  ENSG00000004534      0     0 1.359821e+05        5
## 10 ENSG00000004766      0     0 4.178654e+04       46
```

3 Analyze results

In this section, we provide examples of standard summary tables and plots that can be used to investigate the results of a `regsplice` analysis. The plots are generated using functions from external packages, which need to be installed separately from CRAN or Bioconductor.

3.1 Summary tables of genes with significant evidence for DEU

As shown in the workflow above, we can use the `summary_table()` function to display a complete list of genes with significant evidence for differential exon usage (DEU).

```
summary_table(res, n = Inf)
##                gene       p_vals        p_adj     LR_stats df_tests
## 1  ENSG00000000419 0.000000e+00 0.000000e+00 3.347946e+05       17
## 2  ENSG00000001084 0.000000e+00 0.000000e+00 1.772690e+05       32
## 3  ENSG00000001167 0.000000e+00 0.000000e+00 2.309612e+06        5
## 4  ENSG00000001460 0.000000e+00 0.000000e+00 2.091053e+03        3
## 5  ENSG00000001497 0.000000e+00 0.000000e+00 1.326379e+05       17
## 6  ENSG00000001631 0.000000e+00 0.000000e+00 6.260249e+04       28
## 7  ENSG00000002834 0.000000e+00 0.000000e+00 1.050164e+08       12
## 8  ENSG00000003756 0.000000e+00 0.000000e+00 1.132831e+06       58
## 9  ENSG00000004534 0.000000e+00 0.000000e+00 1.359821e+05        5
## 10 ENSG00000004766 0.000000e+00 0.000000e+00 4.178654e+04       46
## 11 ENSG00000004866 0.000000e+00 0.000000e+00 1.425471e+04       11
## 12 ENSG00000005007 0.000000e+00 0.000000e+00 1.729584e+06       36
## 13 ENSG00000005100 0.000000e+00 0.000000e+00 1.199422e+05       18
## 14 ENSG00000005156 0.000000e+00 0.000000e+00 2.722427e+06       50
## 15 ENSG00000005238 0.000000e+00 0.000000e+00 7.061533e+04       18
## 16 ENSG00000005243 0.000000e+00 0.000000e+00 1.073846e+05       20
## 17 ENSG00000005339 0.000000e+00 0.000000e+00 2.374613e+06       46
## 18 ENSG00000005189 1.036752e-40 1.382336e-40 2.049292e+02        7
## 19 ENSG00000002746 1.062923e-39 1.342640e-39 2.483240e+02       23
## 20 ENSG00000004139 9.629984e-31 1.155598e-30 1.328747e+02        1
## 21 ENSG00000004777 1.794784e-25 2.051182e-25 1.357720e+02        8
## 22 ENSG00000005513 1.290827e-08 1.408175e-08 3.633079e+01        2
```

We can also calculate the total number of genes with significant evidence for DEU. Note that we are using the multiple testing adjusted p-values (Benjamini-Hochberg false discovery rates, FDR) with a threshold of 0.05, which implies that 5% of genes in the list are expected to be false discoveries.

```
sum(res$p_adj < 0.05)
## [1] 22

table(res$p_adj < 0.05)
## 
## FALSE  TRUE 
##    59    22
```

Since this is a simulated data set, we know the true differential splicing (DS) status of each gene. We can use this to calculate a confusion matrix comparing the true and predicted DS status for each gene, for a given threshold.

The results show that for this data set, we have excess false positives as well as some false negatives.

```
# load true DS status labels
file_truth <- system.file("extdata/vignette_truth.txt", package = "regsplice")
data_truth <- read.table(file_truth, header = TRUE, sep = "\t", stringsAsFactors = FALSE)

str(data_truth)
## 'data.frame':    100 obs. of  2 variables:
##  $ gene     : chr  "ENSG00000000003" "ENSG00000000005" "ENSG00000000419" "ENSG00000000457" ...
##  $ ds_status: int  0 0 0 0 0 0 0 0 1 0 ...

# remove genes that were filtered during regsplice analysis
data_truth <- data_truth[data_truth$gene %in% res$gene, ]

dim(data_truth)
## [1] 81  2

# number of true DS genes in simulated data set
sum(data_truth$ds_status == 1)
## [1] 6

table(data_truth$ds_status)
## 
##  0  1 
## 75  6

# confusion matrix comparing true and predicted DS status for each gene
# (prediction threshold: FDR < 0.05)
table(true = data_truth$ds_status, predicted = res$p_adj < 0.05)
##     predicted
## true FALSE TRUE
##    0    57   18
##    1     2    4
```

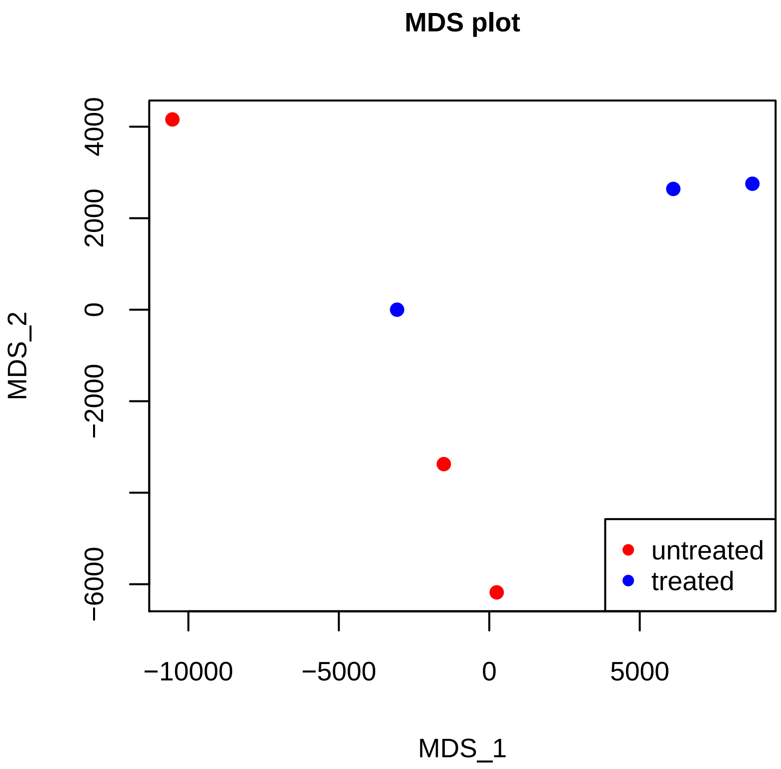
3.2 Exploratory visualizations: MDS plots

Multi-dimensional scaling (MDS) plots can be used to visually judge the similarity between biological samples in a two-dimensional plot. The samples should group into biologically meaningful clusters, for example one for each treatment group or condition. MDS plots are particularly useful for checking for batch effects, which may appear as unexpected clusters.

We use the function `cmdscale()` from the base R package `stats` to generate an MDS plot. The input is a distance matrix calculated from the exon-level read counts table `counts`, which was also used as an input to `regsplice()`.

Each point in the MDS plot represents a biological sample, and treatment groups are indicated with colors.

```
# calculate distance matrix (use transpose t() so samples are in rows)
d <- dist(t(counts))

# calculate MDS fit
fit_mds <- cmdscale(d)

# MDS plot
cols <- rep(c("red", "blue"), each = 3)
plot(fit_mds, col = cols, pch = 16,
     cex.main = 0.8, cex.lab = 0.8, cex.axis = 0.8,
     main = "MDS plot", xlab = "MDS_1", ylab = "MDS_2")
legend("bottomright", legend = c("untreated", "treated"), col = c("red", "blue"),
       pch = 16, cex = 0.8)
```

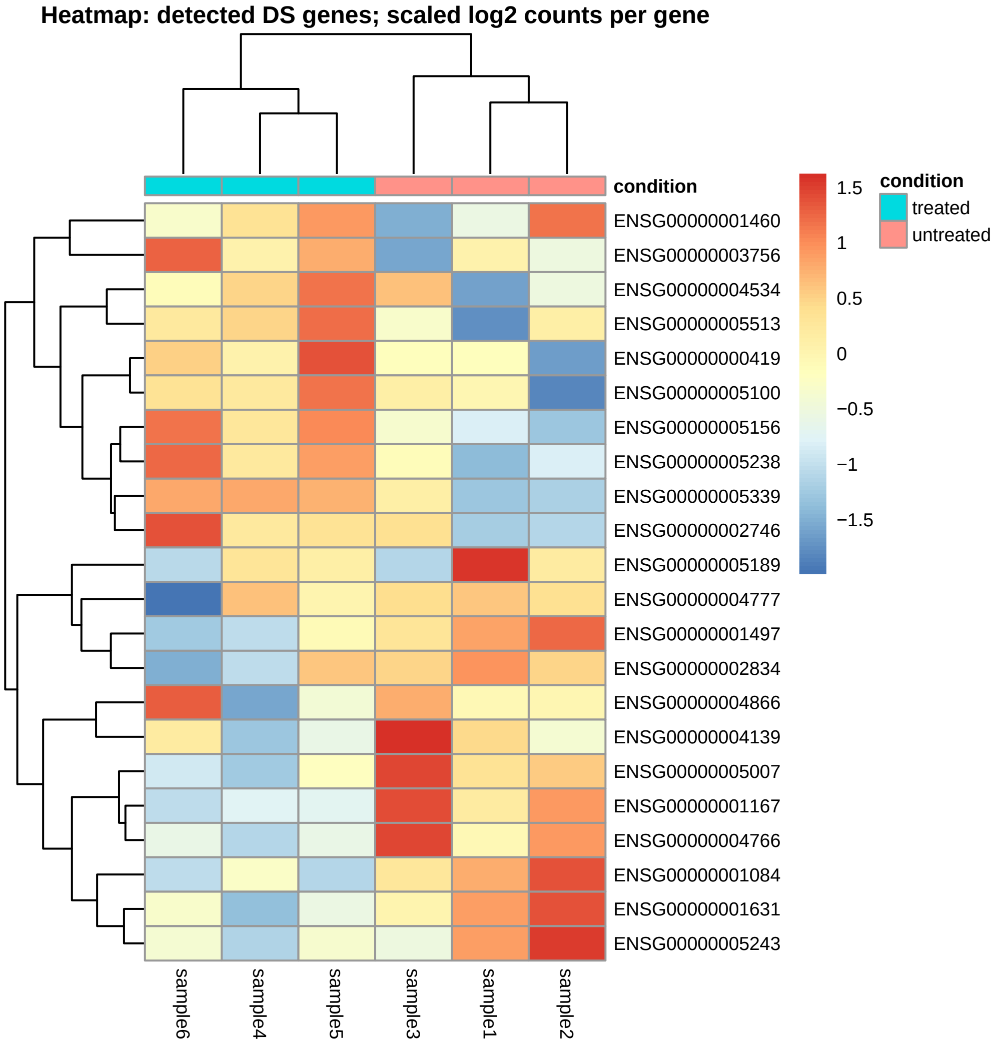


3.3 Summary plots: heatmaps

Heatmaps with hierarchical clustering of rows and columns provide a way to visually compare the data across samples (or treatment groups), for genes that were detected as differentially spliced by the `regsplice` analysis.

Here, we generate a heatmap of log2 total read counts per gene for genes that were detected as differentially spliced (rows), across all biological samples (columns). Note that counts are aggregated to the gene level, while in the previous MDS plot we used exon-level counts. Each row (gene) is scaled to have mean zero and standard deviation one, to make it easier to compare the samples. Treatment groups are indicated with column annotation.

```
suppressPackageStartupMessages(library(pheatmap))  # install from CRAN

# detected DS genes
detected_genes <- summary_table(res, n = Inf)$gene

# select data for detected DS genes (using "counts" and "gene" from beginning of workflow)
Y <- prepare_data(counts, gene)
## removed 936 exon(s) with zero counts
## removed 1 remaining single-exon gene(s)

Y <- filter_exons(Y)
## removed 240 low-count exon(s)
## removed 4 remaining single-exon gene(s)

Y_detected <- Y[detected_genes]

# aggregate counts to gene level
Y_gene <- lapply(Y_detected, colSums)

Y_gene <- as.data.frame(Y_gene)

Y_gene <- t(Y_gene)

Y_gene
##                 sample1 sample2 sample3 sample4 sample5 sample6
## ENSG00000000419    2844    1622    2846    3081    5083    3663
## ENSG00000001084    2319    3396    1700    1208     741     761
## ENSG00000001167    1481    2678    4076     680     695     539
## ENSG00000001460     610     895     496     748     846     649
## ENSG00000001497    1941    2818    1215     379     848     313
## ENSG00000001631    3405    3949    2630    1717    2185    2364
## ENSG00000002834   73646   60281   60454   32241   62627   26437
## ENSG00000003756    7478    5519    3057    7529   11072   14693
## ENSG00000004534    2921    4044    5701    5450    6675    4518
## ENSG00000004766     866    2026    3255     355     541     545
## ENSG00000004866    2879    2902    3532    1961    2622    4036
## ENSG00000005007    8315    8982   12548    4550    6738    5180
## ENSG00000005100    2194    1282    2269    2338    3106    2431
## ENSG00000005156    2882    1923    4418    7436   14206   16350
## ENSG00000005238    2888    3086    3327    3477    3732    3899
## ENSG00000005243    5737    6400    4512    4065    4642    4611
## ENSG00000005339     938    1031    3559    6895    6317    6864
## ENSG00000005189     686     332     166     349     317     167
## ENSG00000002746      46      48      99      92      98     163
## ENSG00000004139     446     378     563     318     364     426
## ENSG00000004777     227     203     204     232     164      52
## ENSG00000005513       1       8       5      12      26       9

# take log2
Y_gene <- log2(Y_gene)

# standardize (mean 0, standard deviation 1)
# (use transpose t() to keep matrix in same shape)
Y_gene <- t(scale(t(Y_gene)))

# heatmap with column annotation (using "condition" from beginning of workflow)
annot_col <- data.frame(condition, row.names = colnames(Y_gene))
pheatmap(Y_gene, annotation_col = annot_col, fontsize = 7,
         main = "Heatmap: detected DS genes; scaled log2 counts per gene")
```



4 Additional options

Additional user options not discussed in the workflow above include:

- `alpha`: Elastic net parameter for `glmnet` model fitting functions. The value of `alpha` must be between 0 (ridge regression) and 1 (lasso). The default value is 1, which fits a lasso model. See `glmnet` package documentation for more details.
- `lambda_choice`: Parameter to select which optimal `lambda` value to choose from the `cv.glmnet` cross validation fit. Available choices are `"lambda.min"` (model with minimum cross-validated error) and `"lambda.1se"` (most regularized model with cross-validated error within one standard error of minimum). The default value is `"lambda.min"`. See `glmnet` package documentation for more details.
- `seed`: Random seed. Provide an integer value to set the random seed in order to generate reproducible results. Default value is `NULL`.

For further details, including a complete list and description of all available user options, refer to the documentation for the `regsplice()` wrapper function, which can be accessed with `?regsplice` or `help(regsplice)`.