Example workflow for regsplice package

Lukas M. Weber

15 September 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. A key advantage of `regsplice` is that runtimes are fast compared to other leading approaches. The detailed statistical methodology and performance comparisons with other methods will be described in an upcoming paper.

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, Matthes 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 consisting 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, the main steps in the workflow are the same as shown below for RNA-seq data. However, the following adjustments to the workflow are required:

- Instead of RNA-seq read counts, a matrix or data frame of exon microarray intensities is provided to the `counts` input argument. The name of the argument is still `counts`, regardless of the input data type.
- Exon microarray intensities should be log2-transformed externally, before they are provided to `regsplice`. This is usually done during pre-processing of microarray data, and may be done automatically depending on your software.
- Filtering of low-count exons should be disabled, by setting the argument `filter = FALSE`.
- Calculation of normalization factors should be disabled, by setting `normalize = FALSE`.
- Calculation of `limma-voom` transformation and weights should be disabled, by setting `voom = 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 a significance threshold can be specified 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 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. The `seed` argument is used to generate reproducible results. Note that the progress bar does not display well in the vignette; it should display on a single line on your screen.

```
library(regsplice)

res <- regsplice(counts, gene, condition, seed = 123)
## 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_val   : num [1:81] 0.429 0.285 1 1 1 ...
##  $ p_adj   : num [1:81] 0.859 0.773 1 1 1 ...
##  $ LR_stats: num [1:81] 0.625 1.143 NA NA NA ...
##  $ df_tests: int [1:81] 1 1 NA NA NA 2 31 9 1 3 ...
```

2.2.1 Summary table of results

The function `summary_table()` can be used to generate a summary table of the results.

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 (e.g. FDR < 0.05).

The argument `rank_by` chooses whether to rank by FDR or raw p-values.

To display results for all genes up to the significance threshold, set the argument `n = Inf`. To display results for all genes in the data set, set both `n = Inf` and `threshold = 1`.

For more details, see `?summary_table`.

```
summary_table(res)
##              gene        p_val        p_adj   LR_stats df_tests
## 1 ENSG00000004766 1.963457e-17 5.890372e-16 137.057172       25
## 2 ENSG00000001461 1.044422e-05 1.566633e-04  25.811578        3
## 3 ENSG00000003436 1.316879e-04 1.316879e-03  17.870151        2
## 4 ENSG00000003249 2.207283e-03 1.655462e-02   9.368648        1
```

2.3 Run workflow using functions for individual steps

Alternatively, the `regsplice` workflow can be run using the individual functions for each step, 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 (i.e. number of exons per gene).

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

The arguments `filter_min_per_exon` and `filter_min_per_sample` control the amount of filtering. Default values are provided; however, these should be adjusted depending on the total number of samples and the number of samples per condition. Any remaining single-exon genes are also removed.

If you are using exon microarray data, this step should be skipped.

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 Calculate normalization factors

The function `run_normalization()` calculates normalization factors, which are used to scale library sizes.

By default, `run_normalization()` uses the TMM (trimmed mean of M-values) normalization method (Robinson and Oshlack, 2010), implemented in the `edgeR` package. For more details, see the documentation for `calcNormFactors()` in the `edgeR` package.

This step should be done after filtering. The normalization factors are then passed on to `limma-voom` in the next step.

If you are using exon microarray data, this step should be skipped.

For more details, see `?run_normalization`.

```
norm_factors <- run_normalization(Y)

norm_factors
## [1] 0.8720188 1.0031023 1.0828220 1.0338759 0.9833141 1.0385113
```

2.3.4 'voom' transformation and weights

The next step is to use `limma-voom` to transform the counts and calculate exon-level weights, with the `run_voom()` function.

The `limma-voom` methodology transforms counts to log2-counts per million (logCPM), and calculates exon-level weights based on the observed mean-variance relationship. This is required because raw integer counts do not fulfill the statistial assumptions required for linear modeling. After the `limma-voom` transformation and weights, standard linear modeling methods can be applied to the data.

For more details, see the following paper, which introduced `voom`; or the limma User's Guide (section "Differential splicing") available on Bioconductor.

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

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

Normalization factors from `run_normalization()` can be provided with the `norm_factors` argument. These are used by `voom` to calculate normalized library sizes. If they are not provided, `voom` will use non-normalized library sizes (columnwise total counts) instead.

If you are using exon microarray data, this step should be skipped.

For more details, see `?run_voom`.

```
out_voom <- run_voom(Y, condition, norm_factors = norm_factors)

Y <- out_voom$Y
weights <- out_voom$weights
```

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.

If you are using exon microarray data, the `weights` arguments should be omitted. The default settings will weight exons equally in this case.

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

```
# set random seed for reproducibility
seed <- 123

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

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

# fit full GLMs (not required if 'when_null_selected = "ones"' in next step)
fit_GLM <- fit_models_GLM(Y, condition, weights, seed = seed)
## 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 <- LR_tests(fit_reg, fit_null)

str(res)
## List of 5
##  $ gene    : chr [1:81] "ENSG00000000003" "ENSG00000000419" "ENSG00000000457" "ENSG00000000460" ...
##  $ p_val   : num [1:81] 0.429 0.312 1 1 1 ...
##  $ p_adj   : num [1:81] 0.945 0.945 1 1 1 ...
```

```
##  $ LR_stats: num [1:81] 0.625 1.022 NA NA NA ...
##  $ df_tests: int [1:81] 1 1 NA NA NA 5 32 7 4 3 ...
```

2.3.7 Summary table of results

The function `summary_table()` can be used to generate a summary table of the results.

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 (e.g. FDR < 0.05).

The argument `rank_by` chooses whether to rank by FDR or raw p-values.

To display results for all genes up to the significance threshold, set the argument `n = Inf`. To display results for all genes in the data set, set both `n = Inf` and `threshold = 1`.

For more details, see `?summary_table`.

```
summary_table(res)
##              gene        p_val        p_adj  LR_stats df_tests
## 1 ENSG00000004766 5.573266e-22 1.839178e-20 130.22077       12
## 2 ENSG00000001461 1.311623e-05 2.164178e-04  25.33879        3
## 3 ENSG00000003436 2.418061e-03 2.659867e-02  18.46368        5
## 4 ENSG00000003249 5.907319e-03 4.873538e-02  10.26313        2
```

3 Analyze results

For the simulated data set in this vignette, the true differential splicing status of each gene is known. In this section, we show how to analyze the results and calculate a contingency table showing the number of true positives, true negatives, false positives, and false negatives.

3.1 Summary of all significant genes

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

```
summary_table(res, n = Inf)
##              gene        p_val        p_adj  LR_stats df_tests
## 1 ENSG00000004766 5.573266e-22 1.839178e-20 130.22077       12
## 2 ENSG00000001461 1.311623e-05 2.164178e-04  25.33879        3
## 3 ENSG00000003436 2.418061e-03 2.659867e-02  18.46368        5
## 4 ENSG00000003249 5.907319e-03 4.873538e-02  10.26313        2
```

The total number of genes with significant evidence for DEU at a given threshold can also be calculated.

Note that we are using the multiple testing adjusted p-values (Benjamini-Hochberg false discovery rates, FDRs) for this calculation. A standard threshold of FDR < 0.05 implies that 5% of genes in the list are expected to be false discoveries.

```
sum(res$p_adj < 0.05)
## [1] 4

table(res$p_adj < 0.05)
##
## FALSE  TRUE
##    77     4
```

3.2 Contingency table

As mentioned above, the true differential splicing (DS) status is known for each gene, since this is a simulated data set. Therefore, we can calculate contingency tables comparing the true and predicted DS status for each gene at a given significance threshold. Increasing the significance threshold returns more genes, at the expense of a large number of false positives.

```
# 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

# contingency table comparing true and predicted DS status for each gene
# (significance threshold: FDR < 0.05)
table(true = data_truth$ds_status, predicted = res$p_adj < 0.05)
##     predicted
## true FALSE TRUE
##    0    72    3
##    1     5    1

# increasing the threshold detects more genes, at the expense of more false positives
table(true = data_truth$ds_status, predicted = res$p_adj < 0.99)
##     predicted
## true FALSE TRUE
##    0    59   16
##    1     2    4
```

4 Additional information

4.1 Additional user 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.

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)`.

4.2 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 does not need to be used directly. In this section, we demonstrate how it works for a single gene, and show an example design matrix, in order to provide further 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
```