Example workflow for regsplice package

Lukas M. Weber

12 June 2016

Package version: regsplice 0.2

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 to other methods are described in our paper:

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

1.1 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 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 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

If you are using exon microarray data, the workflow steps are the same as for RNA-seq data. Note that the argument name `counts` is still used, since the methods were initially developed for RNA-seq data. Simply provide the (log2-transformed) exon microarray intensity matrix instead of the RNA-seq read counts matrix. The intensity matrix should have the same format; i.e. a matrix or data frame with exons in rows, and biological samples in columns.

2 Example 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 in one command.

The results consist of p-values quantifying the statistical evidence for differential exon usage for each gene. The p-values can be used to rank the genes in the data set according to their evidence.

Multiple testing adjusted p-values (Benjamini-Hochberg false discovery rates, FDR) are also provided, as well as the likelihood ratio test statistics and degrees of freedom for each test.

The required inputs for the `regsplice()` wrapper function are `counts` (matrix or data frame of RNA-seq read counts), `gene` (vector of gene IDs), and `condition` (vector of experimental conditions for each sample). Use `?regsplice` or `help(regsplice)` to see 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 4
##  $ p_vals  : num [1:81] 1 0 1 1 0 1 0 0 0 1 ...
##  $ p_adj   : num [1:81] 1 0 1 1 0 1 0 0 0 1 ...
##  $ LR_stats: num [1:81] NA 335730 NA NA 12739 ...
##  $ df_tests: int [1:81] NA 17 NA NA 24 NA 30 8 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 an RNA-seq read count table (`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 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 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 is equal to 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 is equal to the number of genes
length(Y)
## [1] 81
```

2.3.3 Optional: 'voom' weights

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

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

For more information, see the following paper, which introduced voom; or the limma User's Guide, which is 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.

For more details, see ?voom_weights.

```
# calculate 'voom' exon-level precision 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()` to simplify testing and compilation of this vignette. In general, it will be beneficial to use the default or a higher number of cores. We have also used the `suppressWarnings()` command 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, all exons are weighted equally (see above).

For more details, see `?fit_models_reg`.

```
# 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 GLMs (not required if 'when_null_selected = "ones"' in the 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 `"NA"`, the full GLM fitted models are not required, so you can set `fitted_models_GLM = NULL` (the default).

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

For more details, see `?LR_tests`.

```
# calculate likelihood ratio tests
res2 <- LR_tests(fit_reg = fit_reg,
                 fit_null = fit_null,
                 when_null_selected = "ones")
```

2.4 Analyze results

Now that models have been fitted and likelihood ratio tests calculated (either with the wrapper function or the individual functions), we can investigate the results. We will work with the results from the wrapper function (which used `voom` weights but not transformation/normalization; see above).

The `plot_pvals()` function creates a plot of the distribution of p-values or multiple testing adjusted p-values. For typical data sets, this should show a large number of non-significant genes, and a smaller number of genes with significant evidence for differential exon usage (DEU).

The `table_top()` function creates a table of the most highly significant genes.

The `table_significant()` function creates a table of the genes with significant evidence for DEU, according to the multiple testing adjusted p-values with a standard significance threshold for the FDR (default of 5%, which implies that 5% of the genes in the list are expected to be false discoveries).

```
# p-values
# plot(res$p_vals[order(res$p_vals)], type = "b")

# multiple testing adjusted p-values
# plot(res$p_adj[order(res$p_adj)], type = "b")
```

3 Options

full list of other options that can be used with wrapper function, e.g. `alpha`, `lambda_choice`, etc

see help file for more details