spliceR: An R package for classification of alternative splicing and prediction of coding potential from RNA-seq data

Kristoffer Knudsen, Johannes Waage

5 Dec 2013

Contents

1 Introduction **3**
1.1 Alternative splicing classes 3

2 Cufflinks workflow **3**
2.1 Cufflinks gene annotation problem 4

3 GRanges workflow **5**

4 preFiltering **6**

5 SpliceR **7**

6 PTC-annotation **7**

7 GTF export **8**

8 Visualizations **9**

1 Introduction

`spliceR` is an R package that allows for full isoform classification of transcripts generated by RNA-seq assemblers such as Cufflinks. spliceR outputs information about classes of alternative splicing, susceptibility of transcripts to the nonsense mediated decay mechanism (NMD), and provides functions for export of data to the GTF format for viewing in genome browsers, and a few plotting functions. `spliceR` lies at the end of a typical RNA-seq workflow, using full devoncoluted isoforms generated by assemblers, and supports output of genomic locations of spliced elements, facilitating downstream analyses on sequence elements.

`spliceR` stores transcript information in the `SpliceRList` object, constisting of two `GRanges`, 'transcript_features' and 'exon_features'. The 'transcript_features' `GRanges` object contains meta-columns with additional information about each isoform, including parent gene, and expression values. The 'exon_features' `GRanges` contains all exons, with a isoform ID column, refering back to 'transcript_features'. Such a `SpliceRList` can be generated from a Cufflinks workflow by using the built-in `prepareCuff` function, directly reading from a cuffDB object created by Cufflinks auxiliary package `cummeRbund`. Alternatively, a `SpliceRList` can be generated manually from user-supplied transcript and exon information generated by any RNA-seq assembler.

Additionally, `spliceR` provides annotation of transcripts with nonsense mediated decay susceptibility via `annotatePTC()`, visualization via `spliceRPlot()`, generation of GTF files for genome browsers via `generateGTF()`, and a few other auxilliary functions.

1.1 Alternative splicing classes

`spliceR` annotates the 'transcript_features' `GRanges` with the following splice classes:

- Single exon skipping/inclusion (ESI)
- Multiple exon skipping/inclusion (MESI)
- Intron retention/inclusion (ISI)
- Alternative 5'/3' splice sites (A5 / A3)
- Alternative transcription start site (ATSS) / transcription termination site (ATTS)
- Mutually exclusive exons (MEE)

2 Cufflinks workflow

Cufflinks is part of the Tuxido suite, a well-documented and well-supported array of tools for mapping, assembly and quantification of RNA-seq data. At the end of a Tuxido suite workflow often lies analysis with the `cummeRbund` package, that converts the output of Tuxido RNA-seq assembler Cufflinks with the analysis and visualisation power of R. `spliceR` provides the `prepareCuff()` function for easy conversion of `cummeRbund`s `cuffSet` class to `spliceR`s `SpliceRList` class. As `spliceR` requires full information about isoforms and the genomic coordinate of their constituting exons, the transcript GTF file output by Cufflinks must be supplied when generating cuffSet object. Here, we use the built-in data set supplied with `cummeRbund` to generate a `SpliceRList`:

```
> library("spliceR")
> dir <- tempdir()
> extdata <- system.file("extdata", package="cummeRbund")
> file.copy(file.path(extdata, dir(extdata)), dir)

 [1] TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE
[16] TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE

> cuffDB <- readCufflinks(dir=dir,
+     gtf=system.file("extdata/chr1_snippet.gtf", package="cummeRbund"),genome="hg19" ,rebui
> cuffDB_spliceR <- prepareCuff(cuffDB)
```

The `SpliceRList` is a list, containing two `GRanges` objects, `transcript_features` and `exon_features`, containing information about the genomic coordinates and transcript and gene identifiers of transcripts and their constituent exons, respectively. Using `prepareCuff()` creates additional metacolumns in the `transcript_features GRanges` object, allowing for further filtering later in the spliceR workflow. In addition, the `SpliceRList` object contains information about the sample names, the source of the data, any filters applied, and other data used by `spliceR`.

A number of helper functions are provided to extract the transcript and exon GRanges, as well as the sample names:

```
> myTranscripts <- transcripts(cuffDB_spliceR)
> myExons <- exons(cuffDB_spliceR)
> conditions(cuffDB_spliceR)

[1] "iPS"          "hESC"         "Fibroblasts"
```

Once the have the `SpliceRList` object ready, it can be further annotated with alternative splicing classes, using, `spliceR()`, or with information about the coding potential of each transcript with `annotatePTC()`

2.1 Cufflinks gene annotation problem

Cufflinks uses an internal 'gene id'. This gene id is, according to the official manual, 'a unique identifier describing the gene' . But upon investigation we have found that many Cufflinks gene ids actually contain multiple annotated genes. To our knowledge this problem is most visible when running Cufflinks/Cuffdiff with the Ensembl/Gencode as reference transcriptome, using an unstranded RNA-seq experiment. Under these conditions we find that 200-400 (2-4 percent of all) cuffgenes consists of multiple annotated genes. Our hypothesis is, that the problem arises because there are long Ensembl transcripts spanning the intragenic regions between these genes, thereby tricking Cufflinks into thinking that all transcripts, which originates from this region, must belong to the same gene. The global effect of the Cufflinks annotation problem is that the expression levels of all these individual genes are wrong, since the expression level is an average over a couple of genes. This in turns also affects the differential expression analysis of the genes. The Cufflinks annotation problem does however only affect the expression level of genes, meaning that both transcript expression levels and differential expression tests of these are unaffected by this problem.

Our (crude) solution to this problem is to: 1) Identify all affected genes 2) Create a new cuffgene for each real annotated gene within the cuffgenes 3) Recalculate the gene expression according to the new assignment of transcript 4) Set significance test to be negative

This should correct gene expression and remove false positive differentially expressed genes. This setting (The 'fixCufflinksAnnotationProblem' setting in prepareCuff()) is on by default.

3 GRanges workflow

For RNA-seq data from other transcripts assemblers than Cufflinks, a `SpliceRList` can easily be manually created, by supplying the class constructor with transcript and exon `GRanges`. In the following example, a public available transcript and exon model set is downloaded, and expression and p-values between samples are set to faux values to simulate generic RNA-seq assembled data.

First, the UCSC knownGene transcript table is downloaded from the UCSC repository:

```
> session <- browserSession("UCSC")
> genome(session) <- "hg19"
> query <- ucscTableQuery(session, "knownGene")
> tableName(query) <- "knownGene"
> cdsTable <- getTable(query)
```

Next, we download the kgXref table from UCSC, contaning the link between transcript IDs and gene IDs.

```
> tableName(query) <- "kgXref"
> kgXref <- getTable(query)
```

The `GRanges` containing the transcript features is created (setting faux gene expression, fold change and p values), fetching the gene names from the kgXref table:

```
> knownGeneTranscripts <- GRanges(
+         seqnames=cdsTable$"chrom",
+         ranges=IRanges(
+                 start=cdsTable$"txStart",
+                 end=cdsTable$"txEnd"),
+         strand=cdsTable$"strand",
+         spliceR.isoform_id = cdsTable$"name",
+         spliceR.sample_1="placeholder1",
+         spliceR.sample_2="placeholder2",
+         spliceR.gene_id=kgXref[match(cdsTable$"name", kgXref$"kgID"),
+                 "geneSymbol"],
+         spliceR.gene_value_1=1,
+         spliceR.gene_value_2=1,
+         spliceR.gene_log2_fold_change=log2(1/1),
+         spliceR.gene_p_value=1,
+         spliceR.gene_q_value=1,
```

```
+          spliceR.iso_value_1=1,
+          spliceR.iso_value_2=1,
+          spliceR.iso_log2_fold_change=log2(1/1),
+          spliceR.iso_p_value=1,
+          spliceR.iso_q_value=1
+          )
```

The GRanges containing the exon features is created:

```
> startSplit        <- strsplit(as.character(cdsTable$"exonStarts"), split=",")
> endSplit          <- strsplit(as.character(cdsTable$"exonEnds"), split=",")
> startSplit  <- lapply(startSplit, FUN=as.numeric)
> endSplit        <- lapply(endSplit, FUN=as.numeric)
> knownGeneExons <- GRanges(
+          seqnames=rep(cdsTable$"chrom", lapply(startSplit, length)),
+          ranges=IRanges(
+                  start=unlist(startSplit)+1,
+                  end=unlist(endSplit)),
+          strand=rep(cdsTable$"strand", lapply(startSplit, length)),
+          spliceR.isoform_id=rep(knownGeneTranscripts$"spliceR.isoform_id",
+          lapply(startSplit, length)),
+          spliceR.gene_id=rep(knownGeneTranscripts$"spliceR.gene_id",
+          lapply(startSplit, length))
+          )
```

Finally, a SpliceRList object is created from both GRanges, setting assembly, source, and coditions :

```
> knownGeneSpliceRList <- SpliceRList(
+          transcript_features=knownGeneTranscripts,
+          exon_features=knownGeneExons,
+          assembly_id="hg19",
+          source="granges",
+          conditions=c("placeholder1", "placeholder2")
+          )
```

4 preFiltering

Datasets produced by e.g. Cufflinks in multi-sample setups can quickly become very large, as all pairwise combinations are considered. Many genes and transcripts reported by Cufflinks are based on reference GTF files provided at run time, and may not be expressed in any or all samples.

Similarly, Cufflinks may not have confidence in all reported transcripts, marking these models as 'FAIL' or 'LOW DATA'. Filtering these genes and transcripts out of the SpliceRList before running spliceR() and downstream analyses may decrease runtimes considerably.

Given here is an example of a prefiltering step, using standard filters. For more on these filters, refer to the help page for spliceR(). NB: Note that preSpliceRFilter() reports the

number of total pairwise comparison, while `spliceR()` reports the total number of unique isoforms.

```
> #cuffDB_spliceR_filtered <- preSpliceRFilter(
> #        cuffDB_spliceR,
> #        filters=c("expressedIso", "isoOK", "expressedGenes", "geneOK")
> #        )
```

5 SpliceR

`spliceR()` takes a `SpliceRList` object and annotates the isoforms contained within with classes of alternative splicing, as well as a few other metacolumns - data could be generated either by `prepareCuff()` or manually, via a `GRanges` object. The `compareTo` parameter, set to either 'preTranscript' or a given sample, tells `spliceR()` how to sort out isoform comparisons when classifying events. Using the 'preTranscript' parameter, `spliceR()` creates a hypothetical pre-RNA based on all isoforms for each unique gene id, aggregating all exon information. Each isoform is then compared to this construct when classifying. Giving a sample name urges `spliceR()` to consider the major isoform (in terms of isoform expression relative to the total gene expression) of that sample as the primary RNA, comparing each isoform to this reference when classifying.

`spliceR()` can be given a number of filters to reduce the overall data size and running time. Most of these filters are only relevant when running on data from cufflinks, as this software generates additional information for each transcript and gene, indicating the overall robustness and confidence in the models. A typical call to `spliceR()` is shown below, setting common filters and setting the reference to the pre-RNA. For details on all the filters, refer to the help page for `spliceR()`.

```
> #Commented out due to problems with vignettes and progress bars.
> mySpliceRList <- spliceR(
+         cuffDB_spliceR,
+         compareTo="preTranscript",
+         filters=c("expressedGenes","geneOK", "isoOK", "expressedIso", "isoClass"),
+         useProgressBar=F
+ )
```

Typical runtime for < 100.000 transcripts for 4 samples on a contemporary workstation should be in the vicinity of 1.5-2 hours. After `spliceR()` is done, the `transcriptFeatures GRanges` is populated with several additional columns. These include columns for isoform fraction values (IF) and delta IF. Each splicing class gets a column, indicating how many events of a given class was detected for the respective transcripts. Finally, the "analyzed" columned indicates whether this transcript was carried through filtering.

6 PTC-annotation

`annotatePTC()` takes a `SpliceRList` object, a `BSGenome` sequence object and a `CDSList` object. The `BSGenome` sequence object can be downloaded and installed from Bioconductor,

and the assembly and chromosome names herein should match the assembly used for mapping with Cufflinks or any other RNA-seq assembler (e.g. BSgenome.Hsapiens.UCSC.hg19 or BSgenome.Mmusculus.UCSC.mm9). The `CDSList` object is generated using `getCDS()`, which takes a assembly and gene track names, and automatically retrieves CDS information from the UCSC Genome Browser table (internet connection required). Alternatively, a `CDSList` object can be created manually with a CDS table of choice (refer to `?CDSList` for more information). Using the `CDSList` and the `BSGenome` object, `annotatePTC()` retrieves genomic sequence for all the exons contained in the `SpliceRList exon_features GRanges`, and iterates over each transcript in the `transcript_features GRanges`, concatenating exons, and translating the transcript with a compatible CDS. If more than one compatible CDS is found, the most upstream is used. For each transcript, the `transcript_features GRanges` is annotated with stop codon position (in mRNA coordinates), stop codon exon (where 0 is the last exon, -1 is the second last exon and so forth), distance to the final exon-exon junction (in mRNA coordinates), and the genomic location of the used CDS. If no CDS is found, the columns are set to NA. Finally, transcripts are annotated with TRUE/FALSE with respect to nonense mediated decay (NMD) sensitivity; this parameter is set to a default of 50 (meaning, that transcripts with STOPs falling more than 50 nt from the final exon-exon junction, and not in the final exon), but can be changed at runtime. Here follows a typical workflow for `annotatePTC()`:

```
> ucscCDS <- getCDS(selectedGenome="hg19", repoName="UCSC")
> require("BSgenome.Hsapiens.UCSC.hg19",character.only = TRUE)
> #Commented out due to problems with vignettes and progress bars.
> #PTCSpliceRList <- annotatePTC(cuffDB_spliceR, cds=ucscCDS, Hsapiens,
> #        PTCDistance=50)
```

7 GTF export

After running assembly with cufflinks or another assembler, and perhaps `spliceR()` and/or `annotatePTC()`, visualization of trancripts in genome browsers is often helpful. To facilitate this, `spliceR` provides the generateGTF function. `generateGTF()` generates a GTF for each sample, and allows for filtering of transcripts, identical to the filters used in `spliceR()` and `annotatePTC()`. Furthermore, `generateGTF()` color codes transcripts according to expression. The exact scoring method is set by the parameter `scoreMethod` - ”local” scores transcripts relative to the transcript most highly expressed within each unique gene identifier, and ”global” scores transcripts in relation to the whole sample. A typical call to `generateGTF()` could look like this:

```
> generateGTF(mySpliceRList, filters=
+ c("geneOK", "isoOK", "expressedGenes", "expressedIso"),
+         scoreMethod="local",
+         useProgressBar=F)
```

GTF-files are output to the current working directory, and are names according to sample names, with the `filePrefix` argument added to each file.

8 Visualizations

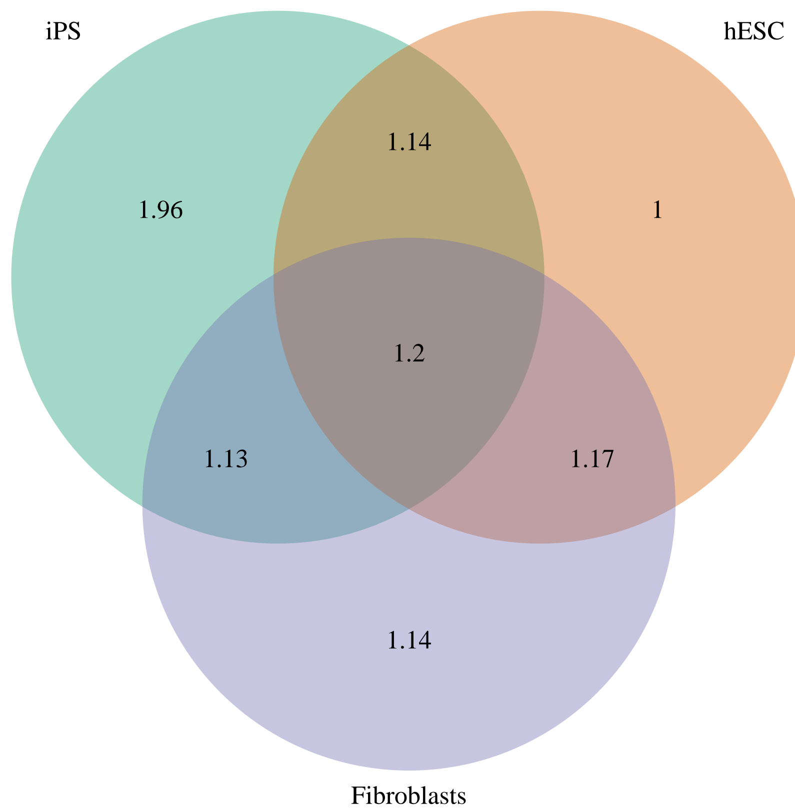
After running `spliceR()`, annotating the spliceRlist object with information about alternative splicing events. the `spliceRPlot()` function can be used for initial data exploration. `spliceRPlot()` generates a range of Venn diagrams, with one circle per condition, analyzing different aspects of the data.

The `spliceRPlot()` function have two layers of control. Firstly the user can control what data is used for the plots by the two parameters expressionCutoff and filters. Since the fundamental purpose of the Venn diagrams is to distinguish between transcripts expressed in the different conditions, the expressionCutoff parameter is used to control at what cutoff is regarded as being expressed. Next `spliceRPlot()` have, like the other core functions of `spliceR`, a filtering argument, accessed by the filters parameter, which allows for selective subsetting of the data.

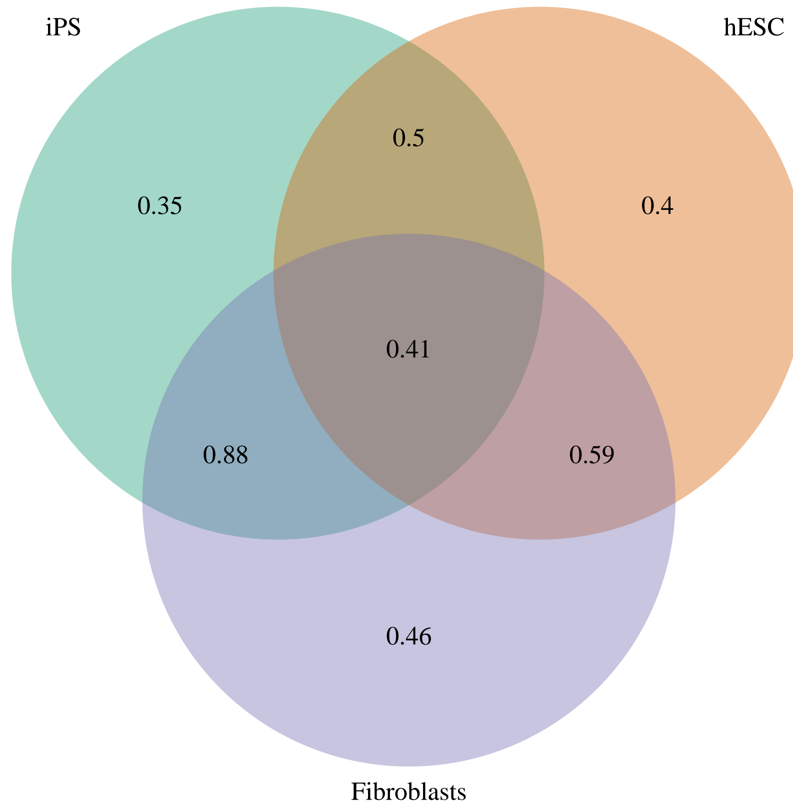
Since filtering data preparation and conversion might take a couple of minutes, `spliceRPlot()` returns a spliceRlist object, where intermediate data structures have been stored, which enables the user to skip this step if another plot is made. If the user which to change the expressionCutoff or the filters parameter, the reset argument should be set to TRUE.

The second layer of control is to determine what is actually plotted. There are a number of options available through the evaluate parameter, ranging from the number of genes expressed in each of the condition, to the average number of alternative splicing events found in the transcripts expressed in the different conditions. Furthermore the "asType" parameter allows the user to specify a subtype of alternative splicing to evaluate.

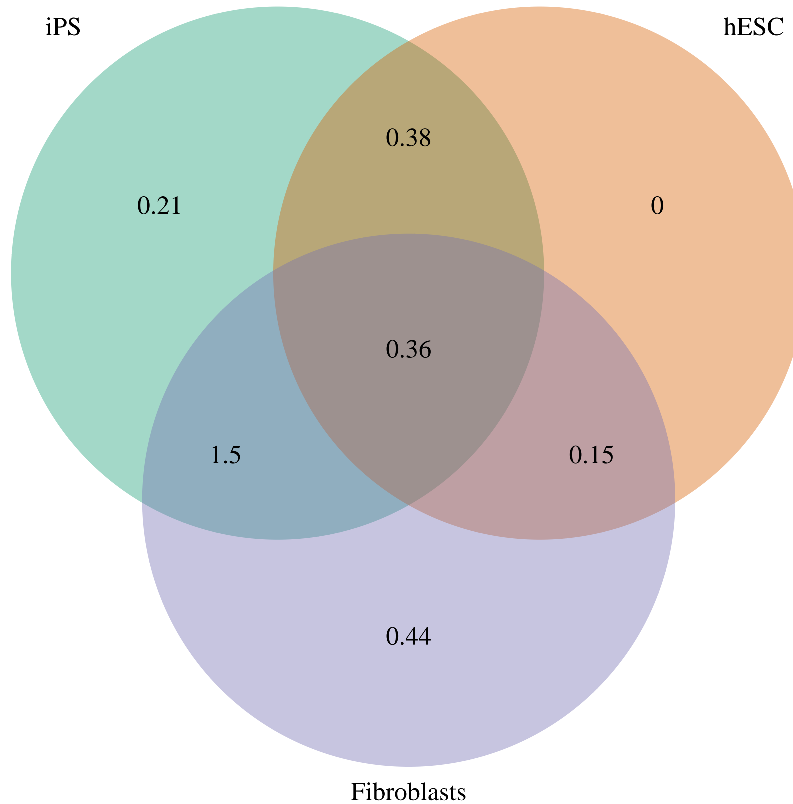
```
> #Plot the average number of transcripts pr gene
> mySpliceRList <- spliceRPlot(mySpliceRList,
+         evaluate="nr_transcript_pr_gene")
```




```
> #Plot the average number of Exon skipping/inclucion evens pr gene
> mySpliceRList <- spliceRPlot(mySpliceRList, evaluate="mean_AS_transcript",
+         asType="ESI")
```




```
> #Plot the average number of Exon skipping/inclucion evens pr gene,
> #but only using transcripts that are significantly differntially expressed
> mySpliceRList <- spliceRPlot(mySpliceRList, evaluate="mean_AS_transcript",
+         asType="ESI", filters="sigIso",reset=TRUE)
```




```
> sessionInfo()

R version 3.4.2 (2017-09-28)
Platform: x86_64-pc-linux-gnu (64-bit)
Running under: Ubuntu 16.04.3 LTS

Matrix products: default
BLAS: /home/biocbuild/bbs-3.6-bioc/R/lib/libRblas.so
LAPACK: /home/biocbuild/bbs-3.6-bioc/R/lib/libRlapack.so

locale:
 [1] LC_CTYPE=en_US.UTF-8       LC_NUMERIC=C
 [3] LC_TIME=en_US.UTF-8        LC_COLLATE=C
 [5] LC_MONETARY=en_US.UTF-8    LC_MESSAGES=en_US.UTF-8
 [7] LC_PAPER=en_US.UTF-8       LC_NAME=C
 [9] LC_ADDRESS=C               LC_TELEPHONE=C
[11] LC_MEASUREMENT=en_US.UTF-8 LC_IDENTIFICATION=C

attached base packages:
 [1] grid      stats4    parallel  stats     graphics  grDevices utils
```

```
 [8] datasets  methods   base

other attached packages:
 [1] BSgenome.Hsapiens.UCSC.hg19_1.4.0 BSgenome_1.46.0
 [3] Biostrings_2.46.0                 XVector_0.18.0
 [5] spliceR_1.20.0                    plyr_1.8.4
 [7] RColorBrewer_1.1-2                VennDiagram_1.6.17
 [9] futile.logger_1.4.3               cummeRbund_2.20.0
[11] Gviz_1.22.0                       rtracklayer_1.38.0
[13] GenomicRanges_1.30.0              GenomeInfoDb_1.14.0
[15] IRanges_2.12.0                    S4Vectors_0.16.0
[17] fastcluster_1.1.24                reshape2_1.4.2
[19] ggplot2_2.2.1                     RSQLite_2.0
[21] BiocGenerics_0.24.0

loaded via a namespace (and not attached):
 [1] ProtGenerics_1.10.0           bitops_1.0-6
 [3] matrixStats_0.52.2            bit64_0.9-7
 [5] progress_1.1.2                httr_1.3.1
 [7] tools_3.4.2                   backports_1.1.1
 [9] R6_2.2.2                      rpart_4.1-11
[11] Hmisc_4.0-3                   DBI_0.7
[13] lazyeval_0.2.1                colorspace_1.3-2
[15] nnet_7.3-12                   gridExtra_2.3
[17] prettyunits_1.0.2             RMySQL_0.10.13
[19] bit_1.1-12                    curl_3.0
[21] compiler_3.4.2                Biobase_2.38.0
[23] htmlTable_1.9                 DelayedArray_0.4.0
[25] scales_0.5.0                  checkmate_1.8.5
[27] stringr_1.2.0                 digest_0.6.12
[29] Rsamtools_1.30.0              foreign_0.8-69
[31] pkgconfig_2.0.1               base64enc_0.1-3
[33] dichromat_2.0-0               htmltools_0.3.6
[35] ensembldb_2.2.0               htmlwidgets_0.9
[37] rlang_0.1.2                   BiocInstaller_1.28.0
[39] shiny_1.0.5                   BiocParallel_1.12.0
[41] acepack_1.4.1                 VariantAnnotation_1.24.0
[43] RCurl_1.95-4.8                magrittr_1.5
[45] GenomeInfoDbData_0.99.1       Formula_1.2-2
[47] Matrix_1.2-11                 Rcpp_0.12.13
[49] munsell_0.4.3                 stringi_1.1.5
[51] yaml_2.1.14                   SummarizedExperiment_1.8.0
[53] zlibbioc_1.24.0               AnnotationHub_2.10.0
[55] blob_1.1.0                    lattice_0.20-35
[57] splines_3.4.2                 GenomicFeatures_1.30.0
[59] knitr_1.17                    biomaRt_2.34.0
```

```
[61] futile.options_1.0.0           XML_3.98-1.9
[63] biovizBase_1.26.0              latticeExtra_0.6-28
[65] lambda.r_1.2                   data.table_1.10.4-3
[67] httpuv_1.3.5                   gtable_0.2.0
[69] assertthat_0.2.0               mime_0.5
[71] xtable_1.8-2                   AnnotationFilter_1.2.0
[73] survival_2.41-3                tibble_1.3.4
[75] GenomicAlignments_1.14.0       AnnotationDbi_1.40.0
[77] memoise_1.1.0                  cluster_2.0.6
[79] interactiveDisplayBase_1.16.0
```