Some Classes and Methods Relevant to RNA-seq Analysis
Laurel Stell
Biomedical Data Science
Stanford University
May 2, 2018

Contents
1 S4 Class System 1
   1.1 Inspecting Objects ........................................... 2
   1.2 Inspecting Methods ........................................... 2
   1.3 Vector Class ..................................................... 3
   1.4 DataTable Class .............................................. 3
   1.5 DataFrame Class .............................................. 3
2 General Bioconductor Classes 4
   2.1 SummarizedExperiment ........................................ 4
3 biomaRt Package 5
4 GenomicRanges Package and GRanges Class 5
5 DESeq2 6
   5.1 Import from HTSeq ............................................ 7
   5.2 Import from RSEM ............................................. 8
   5.3 VST ............................................................. 8
   5.4 Differential Expression Testing .............................. 8
6 edgeR 10

These are notes that I made while working with Bioconductor packages that use the S4 class system. I am sharing them in the hopes that they might be useful to others, but they come with no guarantees, either explicit or implied, as to correctness or completeness.

1 S4 Class System

Bioconductor extensively uses the S4 class system, which is provided by the methods package. Some documentation for S4:

- Search for “methods” in the R search engine and then find “methods:”.

- From the top R HTML help page, click on “Packages” (not “Search Engine & Keywords”), click on “standard” library, and scroll down to “methods”.


• Instead of clicking on “standard” in the previous attempt, click on my personal
directory, then click on “S” in the alphabet, and click on “S4Vectors”. Clicking on
“User guides” reveals three documents:

  – less than one page of code using the Rle class
  – quick overview of S4, which is actually useful—even though it is a set of slides
  – diagram of some classes using S4Vectors

The documentation for the methods package is to help developers create new S4 classes
rather than to help the end users of those classes decipher their contents.

1.1 Inspecting Objects

Use class() on an object as usual. Then showClass() will list the package, slots,
parent classes, and any subclasses. The function str(), which is an old-fashioned generic
function, may or may not give a useful quantity of output.

1.2 Inspecting Methods

To see code for an S4 generic function, first find the method and its signatures and then
get the method for one of the signatures. For example:

> showMethods(assay)
Function: assay (package SummarizedExperiment)
x="RangedSummarizedExperiment", i="missing"
  (inherited from: x="SummarizedExperiment", i="missing")
x="SummarizedExperiment", i="character"
x="SummarizedExperiment", i="missing"
x="SummarizedExperiment", i="numeric"

> selectMethod(assay, c(x="SummarizedExperiment", i="missing"))

Often there is only one argument in the signature, so a simpler call is possible:

> showMethods(assays)
Function: assays (package SummarizedExperiment)
x="DESeqDataSet"
  (inherited from: x="SummarizedExperiment")
x="RangedSummarizedExperiment"
  (inherited from: x="SummarizedExperiment")
x="SummarizedExperiment"

> selectMethod(assays, "SummarizedExperiment")

The function getMethod is very similar to selectMethod, but it does not use inheritance
or group generics.
1.3 Vector Class

Almost everything in the S4Vectors package extends the virtual class Vector. Unfortunately, searching for “s4vectors” or “vector” in the R HTML search engine fails to find this class. Go to the S4Vectors page as instructed above, click on “V” in the alphabet, and click on either “Vector” or “Vector-class”—they both link to the same page.

This class has two slots:

**metadata** is a list that annotates the object as a whole; accessed through the `metadata` function

**elementMetadata** is a DataTable with a row for each element and a column for each metadata variable; accessed through the `mcols` function

1.4 DataTable Class

The DataTable is a virtual class with no slots, but it specifies various methods that one would want to use on a table such as:

**Accessor functions** `nrow`, `dim`, `rownames`, etc

**Subsetting** `x[i,j]`, `head`, `subset` (which has `drop` argument), `na.omit`, etc

**Combining** `cbind`, `merge`, etc

**Looping** `by(data, INDICES, FUN, ..., simplify = TRUE)`

**Transforming** `transform`

**Utilities** `duplicated`, `unique`, `show` (which shows the head and tail with number of lines shown controlled by `global` options `showHeadLines` and `showTailLines`)

**Coercion** `as.env`, `as.data.frame`

**Tabulation** `xtabs`

1.5 DataFrame Class

The DataFrame is a very widely used implementation of the DataTable class that also extends Vector. It adds the following slots:

**rownames** can be NULL

**nrows**

**listData** is a list

**elementType** has class character
It adds the following coercion methods: \texttt{as(from, "DataFrame")}, \texttt{as.list}, \texttt{as.matrix}.

The summary method is utterly useless. The show method provides some useful information, but not if there are more than a few columns—or if there are more than 10 rows and one actually wants to see the whole \texttt{DataFrame}. The only way that I've found to print more than 10 lines is to set the global option with, for example,

\begin{verbatim}
> options(showHeadLines=40)
\end{verbatim}

By default, this option is not set and the default in show for \texttt{DataFrame} is five head lines and 5 tail lines.

\section{General Bioconductor Classes}

\subsection{SummarizedExperiment}

\texttt{SummarizedExperiment} is a Bioconductor data class (from the package of the same name) that can be used to move data between packages. Searching the R HTML engine for the class name actually works. It extends \texttt{Vector}, with the element metadata containing information about the genes. This class adds the slots

\begin{itemize}
  \item \texttt{assays} which is a \texttt{SimpleListAssays} object. If this is a list, all have the same dimensions and names. The first entry is usually (always?) the counts; some other possible entries discussed below. Accessor functions \texttt{assays}, \texttt{assay(x,i=1)}.
  \item \texttt{colData} which is a \texttt{DataFrame} containing information about the samples, with one row for each column in each entry in \texttt{assay}. The accessor function is \texttt{colData}, but \texttt{x$foo} is a shortcut for \texttt{colData(x)$foo}.
\end{itemize}

\texttt{NAMES} has class \texttt{character}

If the components other than \texttt{assays} are not specified, they will be empty. This class also adds the method \texttt{rowData}, which is a synonym for \texttt{mcols}.

\texttt{SummarizedExperiment} is the parent of the \texttt{RangedSummarizedExperiment} class, which adds the slot \texttt{rowRanges} and the accessor of the same name. This is a \texttt{GenomicRanges} object containing genomic annotation as described below. This class modifies \texttt{subset(rse, subset, select)}

so that the \texttt{subset} argument applies to columns of \texttt{rowRanges} and \texttt{select} to columns of \texttt{colData} as in

\begin{verbatim}
subset(des, seqnames != "Y", batch != "Batch01")
\end{verbatim}
biomaRt Package

biomaRt provides an interface to databases implementing the BioMart software suite, including Ensembl, and dbSNP mapped to Ensembl.

To get the gene annotations:

```r
library(biomaRt)
grch37 <- useMart(biomart="ENSEMBL_MART_ENSEMBL",
host="grch37.ensembl.org",
path="/biomart/martservice",
dataset="hsapiens_gene_ensembl")
attr.gene <- getBM(attributes=c("ensembl_gene_id", "external_gene_name",
"chromosome_name", "strand",
"start_position", "end_position"),
mart=grch37,
filters="ensembl_gene_id", values=rownames(data))
attr.gene <- attr.gene[ with(attr.gene,
order(chromosome_name,start_position)), ]
data <- data[rownames(attr.gene), ]
```

where the rownames of `data` are Ensembl gene IDs, as indicated by the `filters` argument. *Warning:* `getBM()` does not return genes in the same order as in the `values` argument, so I immediately put the genes in genomic order in both the annotation frame and the counts matrix. To get a list of all possible values for the `attributes` option:

```r
foo <- listAttributes(grch37)
```

This returns a rather long list, and I think that

```r
foo <- subset(foo, page == "feature_page")
```

was appropriate for this application; I certainly did not need anything from the “snp” page. I thought that including attributes related to GO (Gene Ontology) could be useful, but including `go_id` in my query caused some genes to be removed and others to be repeated. The functions `duplicated` and `match` in the BiocGenerics package might be useful if I need to use an annotation that results in duplicated entries.

GenomicRanges Package and GRanges Class

The `GRanges` class is an implementation of the virtual class `GenomicRanges`, which extends `Vector`. It adds the slots:

- `seqnames` an `Rle` object containing chromosome names
- `ranges` an `IRanges` or `IPos` object
- `strand` an `Rle` object
seqinfo an Seqinfo object

Some of the methods for this class include:

**Accessor** functions:

- accessor functions have the same names as the components
- `start(x)` is a shortcut for `start(ranges(x))`
- `x$foo` is a shortcut for `mcols(x)$foo`

**Subsetting:** See the class help page, but highlights include `[`, `head`, `tail`, `subset`, and

`window(x, start, end, width, frequency, delta, ...)`

For the last of these, two out of three of arguments 3–5 are required, and `frequency` and `delta` are optional for specifying subsampling within the window.

## 5 DESeq2

DESeq2 has the class `DESeqDataSet`, which is a subclass of `RangedSummarizedExperiment`. It adds the slots:

**design** a `formula` object that specifies the experimental design in terms of the columns of `colData`. For example, the design formula `~ x1 + x2` means that we want to test for the effect of `x2` controlling for the effect of `x1`. If the research aim is to determine for which genes the effect of treatment is different across groups, then interaction terms can be included and tested using a design such as `~ group + treatment + group:treatment`.

**dispersionFunction** a `function` object

It also adds accessor methods with the same names as these slots. This class requires all entries of the count matrix to be non-negative integers and does not allow `colData` to be `NULL`.

Given a counts matrix `data` and a data frame `desc.data` describing samples (with column names of `data` matching row names of `desc.data`, create one of these objects as follows:

```r
des <- DESeqDataSetFromMatrix(data, 
   transform(desc.data, group=grp.cancer),
   ~ batch + group, 
   rowRanges=gr)
```
where the optional GRanges object gr is created above and grp.cancer is a factor identifying healthy individuals and cancer patients. (As expected, there is a constructor that takes a RangedSummarizedExperiment object and a design formula.) Keeping extra columns in the GRanges object means that the gene ID and name are put in both rowRanges and the element metadata, so mcols(des) is identical to mcols(rowRanges(des)). A detailed example at http://www.bioconductor.org/help/workflows/rnaseqGene/#eda gives one possible workflow to get from FASTQ files to DESeqDataSet object, but preferred methods are described below.

Notes on some other methods added by this class:

Accessor functions:

- counts(des) by default is equivalent to assays(des)["counts"], but it also has an option to return normalized counts.
- ranges(des) is a shortcut for ranges(rowRanges(des)) and so on, including start, rank, etc.
- dispersions(des) is a shortcut for mcols(des)$dispersion

Summarizing: summary is way too brief and str is way too long. show (which is used to print the object) is the best I've found, but not ideal.

5.1 Import from HTSeq

https://bioconductor.org/packages/release/bioc/vignettes/DESeq2/inst/doc/DESeq2.html gives a detailed example for importing counts.txt output by htseq-count. It will automagically handle the diagnostic lines at the end of the file. Unfortunately, it does not work if the file contains gene symbol as well as ID. In that case, a very minimalist approach is:

tmp <- list()
for (nnn in seq_along(files)) {
    tmp[[nnn]] <- read.delim(sprintf("%s/%s", DATA_DIR, files[nnn]), header=F)
}
# identical(tmp[[1]][,1:2], tmp[[2]][,1:2])
# identical(tmp[[1]][,1:2], tmp[[3]][,1:2])
data <- cbind(tmp[[1]], tmp[[2]][3], tmp[[3]][3])
names(data) <- c("id", "symbol", ids)
cond <- substr(ids, 1, 2)
df <- data.frame(sampleName=ids, condition=cond)
tmp <- data[,ids]
rownames(tmp) <- data$id
dds <- DESeqDataSetFromMatrix(tmp, df, ~ condition)
tmp <- data[,"symbol",drop=F]
rownames(tmp) <- data$id
rowData(dds) <- tmp
5.2 Import from RSEM

https://bioconductor.org/packages/release/bioc/vignettes/tximport/inst/doc/tximport.html gives examples using the tximport package to read counts output by RSEM and then create a DESeqDataSet object. Suppose that the files are in directory FOO and that data frame samples has a column id.samples that I want to use as well as a column id.rsem that the GBSC has used.

```r
files <- file.path(FOO, paste0(samples$id.rsem, ".genes.results"))
# This will make the names in dds be what I want to use.
names(files) <- rownames(samples$id.samples)
rna.txi <- tximport(files, type="rsem")
# Fix RSEM outputting 0 lengths. (Recommended by Michael Love.)
rna.txi$length[rna.txi$length == 0] <- 1
# Can also add GRanges object as shown above.
row.appear <- which(rowSums(counts(dds)) > 0)
dds <- dds[ row.appear, ]
```

Using DESeqDataSetFromTximport sets assays(dds)["avgTxLength"] to the effective lengths output by RSEM.

5.3 VST

The functions varianceStabilizingTransformation and vst return a DESeqTransform object if given a DESeqDataSet. (They can also be applied to a matrix, in which case they return a matrix.) The DESeqTransform class is also an extension of the RangedSummarizedExperiment class. It does not have the counts accessor function—presumably because it does not actually contain counts.

5.4 Differential Expression Testing

Differential expression testing starts by calling DESeq(des), which returns des with lots of stuff added to it. Assuming that des does not already have normalization or size factors, this will call estimateSizeFactors(des). If des has an assay called avgTxLength (which is the case when tximport was used to import RSEM output), then this function will set assays(object)["normalizationFactors"] to estimateNormFactors return value. Otherwise, estimateSizeFactorsForMatrix() is called and its return value is set to colData(des)$sizeFactor.

Next, DESeq calls estimateDispersions, which calls the following:

- estimateDispersionsGeneEst adds column dispGeneEst containing gene-wise estimates of dispersion to element metadata; sets assays(des)["mu"]
- estimateDispersionsFit sets the dispersionFunction slot; for example
> dispersionFunction(foo)
function (q)
  coefs[1] + coefs[2]/q
<bytecode: 0x12cf0350>
<environment: 0x1a4def30>
attr("coefficients")
  asymptDisp  extraPois
  0.0189232  3.0438190
attr("fitType")
  [1] "parametric"
attr("varLogDispEsts")
  [1] 1.057399
attr("dispPriorVar")
  [1] 0.8760762

estimateDispersionsMAP adds the following columns to element metadata:

  dispersion final estimate of dispersion
  dispIter number of iterations
  dispOutlier dispersion flagged as outlier
  dispMAP maximum a posteriori estimate

The function plotDispEsts(des) uses the following columns of the element metadata:
dispGeneEst, dispOutlier, dispersion, dispFit. I can tell from the element metadata for the element
metadata of the DESeqDataSet object that dispFit was added between adding dispGeneEst and dispersion, but I cannot track it down.

After estimating dispersion, DESeq calls either nbinomWaldTest or nbinomLRT. When I called DESeq in a particular case (first set of Fathman data, 79 individuals, 9 surrogate variables) without specifying any optional arguments, it gave a warning message about beta not converging for one gene and suggested increasing maxit. To identify the bad gene:

nonconv <- which(!mcols(obj.de)$betaConv)

where obj.de is the object returned by DESeq. Since maxit is not an argument to DESeq, I saved its definition to DESeqMaxit.R, changed the function name to DESeqMaxit and added maxit as an input argument and as an argument to the call to nbinomWaldTest. I then increased maxit to 500 from its default value of 100. This not only failed to result in convergence for the troublesome gene but also resulted in another gene failing to converge. The Webosphere suggests that the failure to converge in 100 iterations is usually due to some insidious problem that will prevent convergence in an infinite number of iterations. I went back to using default maxit but kept the code for the experiment.

The function resultsNames returns the names of the coefficients fit. They are buried in mcols(mcols(des)). Before calling DESeq, this function returns an empty character
vector. If the object returned by DESeq is de, one must call results(de) to actually see results. This returns a DESeqResults object, which is a simple subclass of DataFrame. This object contains the columns: baseMean, log2FoldChange, lfcSE, stat, pvalue and padj. Its metadata columns describing these columns. The lfcSE gives the standard error of the log2FoldChange. For the Wald test, stat is the Wald statistic: the log2FoldChange divided by lfcSE, which is compared to a standard Normal distribution to generate a two-tailed pvalue.

By default, results() returns the comparison of the last level of the last variable in the design formula over the first level of this variable. If the last variable has more than two levels, one can obtain other comparisons with a call such as

\[ \text{results(object, contrast=c("group","lvl3","lvl2"))} \]

which returns the comparison of these two levels of the variable group. If log2FoldChange is positive, then lvl3 is more highly expressed than lvl2. If the variable is logical, then use

\[ \text{results(object, contrast=list("treatedTRUE"))} \]

which will have positive log2FoldChange if treated samples are upregulated compared to untreated. Specifying contrast as a list can be generalized to test interaction terms, and there is yet another option that enables using averages of levels, but I have not yet needed those. It is only possible to compare two levels; one cannot ask for a p-value for lvl1 < lvl2 < lvl3, say.

For a one-sided rather than two-sided test, specify altHypothesis="greater" or altHypothesis="less" in the call to results(). In the former case (with lfcThreshold=0), the statistic will be zero if log2FoldChange is negative.

6 edgeR

The edgeR class DGEList inherits directly from list. The initial components to this class are the matrix counts and data frames samples and (optional) genes. To create from objects described above:

\[ \text{dge <- DGEList(counts=data, samples=desc.data, group=grp.cancer,} \]
\[ \quad \text{genes=attr.gene)} \]

All of these arguments are actually optional. Another argument is remove.zeros to remove genes with total count zero, which I am not using because I am also going to remove genes with a total count of one. The constructor adds group, lib.size and norm.factors to dge$samples.

Notes on manipulating a DGEList object:

Accessors: The only accessor for the initial components seems to be getCounts.

Summarizing: summary is way too brief and show is way too long. str is the best I’ve found, but it doesn’t show component names. At least dim(dge) is defined.
Subsetting: The calls

```r
dge[3,17]
dge[3,]
dge[rowSums(getCounts(dge)) > 0, ]
```

return DGEList objects with all components appropriately subsetted. It does not, however, change lib.size unless explicitly instructed with the keep.sizes option (eg dge[3,,keep.sizes=F]). This class does not have a subset function.

Combining: Apparently has to be done manually.