Annotation and Visualisation of sequencing data in Bioconductor

Mark Dunning

Last modified: July 22, 2015

Contents

1 Annotation using an online resource

The *biomaRt* package provides an interface to the biomart website; allowing the data to be accessed in a consistent manner and without having write complex SQL queries or understand the underlying schema. The databases available through BioMart can be obtained with the `listMarts` function

1.1 General BiomaRt Usage

Use Case: Connect to the default version of Ensembl through BiomaRt. What vesrion of the human genome does it provide?

```
library(biomaRt)
listMarts()
```

```
ensemblLatest <- useMart("ensembl")

datasets <- listDatasets(ensemblLatest)
datasets[which(datasets[,1] == "hsapiens_gene_ensembl"),]
```

Unfortunately, our RNA-seq was aligned and annotated against *hg19*. Different versions of biomaRt and genome version can have different attributes and different underlying annotation. Therefore, if we annotate using a conflicting version of Ensembl our results may be misleading.

Use Case: Connect to the human genome in the February 2014 archive of biomaRt. Verify that hg19 is being accessed.

```
ensembl_75 = useMart(biomart="ENSEMBL_MART_ENSEMBL", host="feb2014.archive.ensembl.org",
                     path="/biomart/martservice", dataset="hsapiens_gene_ensembl")
datasets <- listDatasets(ensembl_75)
datasets[which(datasets[,1] == "hsapiens_gene_ensembl"),]
```

The function `getBM` is used to build-up queries in biomart. We must specify one or more *attributes* that we want to retreive along with the *filters* that we are using the query the database.

1.2 Annotating the RNA-seq results

Having now connected to a suitable database through *biomaRt*, we will now proceed to use the online resources to annotate our RNA-seq results.

Use Case: Read the results of the edgeR analysis. If you have a set of results from the RNA-seq practical, feel free to use these. Otherwise, you can find a set of results in the Day3 course directory. Take just the top 100 results to make the queries run quicker

```
load("Day3//topHits_FWER_0.01.RData")

head(topHits)
topHits <- topHits[1:100,]
```

The `topHits` data frame provides us with a list of genes that are found to be *statistically significant*. However, it is a separate investigation to determine if the genes are *Biologically significant*.

We can use *biomaRt* to annotate the results of our RNA-seq analysis. Firstly, we have to recognise that the first column of the `topHits` data frame are Entrez IDs. We can then use these IDs as filters to access the database. When we want to access the database, it is important to know the exact name that *biomaRt* uses for the filter.

Use Case: What is the name of the filter for Entrez gene ID?

comment: We can search for particular text in the data frame of attributes using `grep`, `match` etc

```
flt <- listFilters(ensembl_75)
head(flt)
```

```
flt[grep("entrez",flt[,1]),]
```

Then it is a matter of deciding what attributes we want to retrieve from the database. Again, we need to know the names that *biomaRt* will accept for the particular attributes that we want.

Use Case: What are the attribure names for HGNC symbol, chromosome name and description?

```
attr <- listAttributes(ensembl_75)
attr[1:50,]
attr[grep("symbol",attr[,1]),]
```

We now have everything we need in order to make a biomaRt query.

Use Case: Annotate the top hits from the DEseq analysis with their gene symbol and description. Make sure that you also include Entrez ID in the list of attributes that are returned. *warning: You will need an internet connection in order for this to work*

```
extraInfo <- getBM(attributes = c("entrezgene","hgnc_symbol", "description"),
               filters = "entrezgene",
               values = topHits[,1],
               mart = ensembl_75)

head(extraInfo)
```

You may notice that the results retrieved from biomaRt may not be in the same order as the RNA-seq results that we are trying to annotate. The `merge` function is useful in joining two data frames together and making sure that the rows match-up. In addition to the two data frames we are trying to merge, we also need to have a common identifier in both data frames (in our case, the Entrez ID). The `by.x` and `by.y` arguments are used to specify the column that this identifier can be found in.

Use Case: Create a merged table containing both edgeR results and the annotations.

```
annotatedHits <- merge(topHits, extraInfo, by.x = 1, by.y = 1,sort=FALSE)

head(annotatedHits)
```

The resulting table, `annotatedHits`, can now be used to ease the interpretation of the RNA-seq analysis and can be shared with collaborators.

1.3 Retrieving Gene Sequences

In this section we will explore how to retrieve sequences for our most differentially-expressed genes. Such a set of sequences could be used to discover motifs, for example.

Use Case: Retrieve the chromosome names, start and end positions and strand for the top hits. Restrict the data to just chromsomes 1 to 22 and the sex chromosomes.

```
posInfo <- getBM(attributes = c("entrezgene", "chromosome_name","start_position"
                                ,"end_position","strand"),
              filters = "entrezgene",
              values = topHits[,1],
              mart = ensembl_75)

posInfo <- posInfo[posInfo[,2] %in% c(1:22,"X","Y"),]
```

You should be aware that plenty of functionality exists to manipulate and analyse intervals in the form of GRanges objects. You'll see that the chromosome names from *biomaRt* are numerical values, whereas other packages in Bioconductor expect chromosome names with the *chr* prefix. Also, the strand is not given in a standard format and we need to convert these into '+' or '-' characters.

Use Case: Create a GenomicRanges representation of these gene coordinates. You will need to make sure that the sequence names and strand information is acceptable.

comment: The ifelse function is a useful way of populating a vector based on a logical test. i.e. each value in the output vector is determined by the result of the logical test.

```
library(GenomicRanges)
strand = ifelse(posInfo[,5] == 1, "+", "-")
genePos <- GRanges(paste0("chr",posInfo[,2]),
                   IRanges(posInfo[,3],posInfo[,4],names=posInfo[,1]),strand)
```

comment: You may sometimes find the with function used in these situations. It allows the columns from a data frame to be accessed by name, so the code is a bit more elegant. The result should be the same however.

```
genePos <- with(posInfo, GRanges(paste0("chr",chromosome_name),
                                 IRanges(start_position,end_position,names=entrezgene),
                                 strand))
```

Use Case: Load the package that provides genome sequence for hg19 and get the DNA sequences for the top hits. Translate the sequences into amino acids

```
library(BSgenome.Hsapiens.UCSC.hg19)
hg19 <- BSgenome.Hsapiens.UCSC.hg19
myseqs <- getSeq(hg19,genePos)
myseqs
translate(myseqs)
```

The GRanges object that we created can be manipulated using the functionality from the *IRanges* package. For instance we can resize, extend, collapse the intervals.

Use Case: Get the sequences 100 base-pairs upstream of each gene and write these to a *fasta* file

```
promSeqs <- getSeq(hg19, flank(genePos,100))
writeXStringSet(promSeqs , file = "topGenesPromoters.fa")
```

2 Annotation using pre-built packages

Organism-level packages provide an alternative to *biomaRt* and permit annotation queries offline. These packages are re-built every 6 months as part of the Bioconductor development cycle and are version controlled. Therefore it is easy to keep track of which version of the package was used for a particular analysis. Each package is built around a central identifier (e.g. Entrez ID) and meta data surrounding the annotation sources used can be displayed once such an organism package has been installed and loaded.

The terminology of `columns` and `keys` is equivalent to `attributes` and `filters` used by *biomaRt*. The valid columns and keys can be interrogated using `columns` and `keytypes`.

Use Case: Load the annotation package for Humans and display the metadata describing how the package was created. Find out what information can be retrieved from the package and what types of key can be used.

comment: For covenience, we often assign a shorter name to the annotation package.

```
library(org.Hs.eg.db)
hs <- org.Hs.eg.db
columns(hs)
keytypes(hs)
```

In the analysis of this RNA-seq dataset, we allocated reads to genes. However, in this section we will explore the ways of counting reads that align to other genomic regions of interest. In this case we will use *exons* and have to re-visit the orginal aligned reads.

In order to keep the computational requirements for the analysis down file we will restrict the analysis of genes that occur on chromosome 22. We therefore need a list of the IDs of all such genes. We can do this using the *org.Hs.eg.db* organism package. The `select` function is used to make a query to the organism database package. We have to supply a set of valid keys and columns, in a similar manner to how we used *biomaRt*.

Use Case: Use the appropriate organism package to retrieve the Entrez IDs of all genes located on Human chromosome 22.

```
chr22Genes <- select(hs, columns = "ENTREZID", keytype = "CHR", keys = "22")
head(chr22Genes)
chr22ID <- chr22Genes[, 2]
```

warning: If you get the error message:
`unused arguments (columns = "ENTREZID", keytype = "CHR", keys = "22")`
when trying to run the `select`, it is because another R package has defined another function called `select` and over-written the function from AnnotationDBI that we intended to use. If this occurs the statement
`select <- AnnotationDbi::select`
should allow you to run the line of code.

2.1 Transcript packages

Bioconductor provide a number of pre-built packages that have a database of all transcript information for a particular organism. The package we are going to use is *TxDb.Hsapiens.UCSC.hg19.knownGene*. A full list of available transcript packages is available on the Bioconductor website. Look for the packages that start with `TxDb.`.

As its name suggests, *TxDb.Hsapiens.UCSC.hg19.knownGene* was built from the UCSC tables for the hg19 genome. Convenient functions exist that can return the gene structure as a *GRanges* object. One such function is `exonsBy` which returns a list of `GRanges` objects, each item in the list pertaining to a particular gene. Thus, the output of `exonsBy` can be subset in the usual manner (`'[['`).

comment: transcripts and cds regions can be selected in a similar fashion. See the help pages of `intronsBy` and `cdsBy`

Use Case: Get all the locations of exons on chromosome 22, grouped according to gene. How many Genes are there on chromosome 22?

```
library(TxDb.Hsapiens.UCSC.hg19.knownGene)
tx <- TxDb.Hsapiens.UCSC.hg19.knownGene

exo <- exonsBy(tx, "gene")
exo

length(exo)
exo22 <- exo[names(exo) %in% chr22ID]
length(exo22)
```

comment: If you cannot find a TranscriptDb package that provides the annotation you need, you can use the `makeTxDbPackage` function to create one from the UCSC browser, biomaRt or another source. See the help page `?makeTxDbPackage`for more details.

comment: Make sure that you understand how the ranges in this exon list are accessed. HINT: Why does `exo22[[1]]` work, but `exo22[["1"]]` not

Use Case: Import the aligned reads for sample "16N" using *GenomicAlignments*. Restrict the query to just chromsome 22 reads.

```
library(GenomicAlignments)
bam <- readGAlignments(file="Day3//16N_aligned.bam",
                       param=ScanBamParam(which = GRanges("chr22",IRanges(1,51304566))),
                       use.name=TRUE)
```

We are now in a position to begin overlapping reads with exons. You should have already seen the `findOverlaps` function that will report the amount of ovelap between two sets of ranges. We will start simple by just doing an overlap of a single gene to make sure that we understand the output

Use Case: Use the `countOverlaps` function to count the number of reads covering each exon of Entrez ID "4627".

```
counts <- countOverlaps(exo22[["4627"]], bam)
counts
```

summarizeOverlaps extends findOverlaps by providing options to resolve reads that overlap multiple features. Each read is counted a maximum of once. Different modes of counting are available. See the help page for summarizeOverlaps for more details.

Use Case: Repeat the counting for all genes on chromosome 22

```
so <- summarizeOverlaps(exo22, bam)
gCounts <- assays(so)$counts
dim(gCounts)
head(gCounts)
```

The summarizeOverlaps function can also take the location of one or more bam files as an argument. This means that the processing of reads and counting is handled in one step. We have to use the special BamFileList constructor when dealing with multiple files.

Use Case: Obtain counts of chromosome 22 gene "4627" for the bam files provided.

warning: This next example uses large amounts of memory, so you may not be able to run at home

```
fls <- paste0("Day3/",dir("Day3/",pattern=".bam"))
fls <- fls[-grep("bai",fls)]
names(fls) <- basename(fls)
allOverlaps <- summarizeOverlaps(exo22[["4627"]], BamFileList(fls))
head(assays(multiOlap)$counts)
sampCounts <- assays(multiOlap)$counts
```

3 Visualisation

3.1 Exporting tracks

It is also possible to save the results of a Bioconductor analysis in a browser to enable interactive analysis and integration with other data types, or sharing with collaborators. For instance, we might want a browser track to indicate where our differentially-expressed genes are located. We shall use the *bed* format to display these locations.

At the moment, we have a GenomicFeatures object that represents every exon. However, we do not need this level of granularity for the bed output.

Use Case: Use the range function to obtain a single range for every gene

```
geneRegions <- unlist(range(exo))
```

We can now select the names of the statistically significant genes from the edgeR output in the usual manner.

Use Case: Select all significant genes from the edgeR analysis (lets use a p-value of 0.01), and then get the genomic intervals that correspond to these genes.

```
load("Day3//topHits_FWER_0.01.RData")
sigResults <- topHits
head(sigResults)
sigGeneRegions <- geneRegions[na.omit(match(sigResults[,1],names(geneRegions)))]
```

comment: For some reason, some of the genes reported in the edgeR output do not have associated genomic features. Hence we have to discard such genes from the output by using `na.omit`

Rather than just representing the genomic locations, the .bed format is also able to colour each range according to some property of the analysis (e.g. direction and magnitude of change) to help highlight particular regions of interest. A *score* can also be displayed when a particular region is clicked-on. A useful propery of `GenomicRanges` is that we can attach metadata to each range using the `mcols` function. The metadata can be supplied in the form of a data frame.

Use Case: Store the adjusted p-values and log fold-change in the ranges object

```
mcols(sigGeneRegions) <- sigResults[match(names(sigGeneRegions), sigResults[,1]),]
```

Use Case: Use the human organism package to add gene symbols to this object

```
anno <- select(org.Hs.eg.db,keys = sigGeneRegions$genes,
               keytype="ENTREZID",columns="SYMBOL")
mcols(sigGeneRegions)$Symbol <- anno[,2]
```

Use Case: Tidy-up the GenomicRanges object so that only chromosomes 1 to 22 and sex chromosomes are included

```
seqlevels(sigGeneRegions)
sigGeneRegions <- keepSeqlevels(sigGeneRegions, paste0("chr", c(1:22,"X","Y")))
```

comment: Here, `keepSeqLevels` *is used as a convenience to retain only the chromosomes that we are interested in*

Use Case: Create a score from the p-values that will displayed under each region, and colour scheme for the regions based on the fold-change. For convenience, restrict the fold changes to be within the region -3 to 3.

```
Score <- -log10(sigGeneRegions$FWER)
rbPal <-colorRampPalette(c("red", "blue"))

logfc <-  pmax(sigGeneRegions$logFC, -3)
logfc <-  pmin(logfc , 3)

Col <- rbPal(10)[as.numeric(cut(logfc, breaks = 10))]
```

The colours and score can be saved in the *GRanges* object and `score` and `itemRgb` columns respectively and will be used to construct the track when exporting. The *rtracklayer* function can be used to import

and export browsers tracks.

Use Case: Export the signifcant results from the DE analysis as a *.bed* track using *rtracklayer*. You can load the resulting file in IGV, if you wish.

```
mcols(sigGeneRegions)$score <- Score

mcols(sigGeneRegions)$itemRgb <- Col
library(rtracklayer)
export(sigGeneRegions , con = "topHits.bed")
```

The export of other formats such as *gff*, *bedGraph*, *wig* and *bigwig* is also possible with *rtracklayer*. We do not need to explicitly tell the function what format to export the data in, it automatically detects this from the file name.

3.2 Brief introduction to ggplot2

ggplot2 is a popular package that is capable of generating sophisticated graphics. The key concept is that a plot is generated by adding a series of *layers* onto a dataset that define how the data are transformed and displayed. To construct a plot in *ggplot2*, we must have our data in a *data frame* and be able to define the characteristics (*aesthetics*) of the plot from the variables in the data frame.

A volcano plot is commonly used to identify changes in high-throughput studies. The magnitude of change is shown on the x-axis and significance on the y-axis. In gene-expression studies, the significance measure is usually the *log-odds* score (commonly-denoted as 'B' in limma and related projects).

For these plots, we will load up the edgeR results for *all* genes and use ggplot2 to signify which are the differentially expressed genes.

Use Case: Load the complete table of edgeR results, and create another column to denote some measure of significance; with increasing values denoting increasing evidence for differential expression

```
load("Day3/edgeRAnalysis_ALLGENES.RData")
head(y)
y$significance <- -10*log10(y$FDR)
edgeRresults <- y
```

Use Case: Make a volcano plot from the edgeR results using ggplot2. Select appropriate columns from the data frame to map to the x and y axis.

```
library(ggplot2)
ggplot(edgeRresults, aes(x = logFC,y=significance)) + geom_point()
```

comment: To construct the plot, we have to specify what variables are mapped to the aesthetics on the plot using the `aes` *function. However, this is not enough to produce the plot. We also have to specify how the variables are arranged on the plot by picking an appropriate geom. In this case we choose* `geom_point` *for a scatter plot.*

A real advantage of *ggplot2* is that we are able to use the variables in the data frame to set other properties of the plot, such as the colour, size and plotting characters. *ggplot2* will take care of the colouring and create an appropriate legend.

Use Case: Create a new variable in the edgeR results data frame to indicate whether each gene is statistically significant or not. Use this new variable to colour the points on the volcano plot

```
edgeRresults$DE <- edgeRresults$FDR < 0.01
ggplot(edgeRresults, aes(x = logFC,y=significance,col=DE)) + geom_point()
```

A disadvantage of *ggplot2* is that is not immediately obvious how to change the appearance of the plot once it has been created; especially if one is familiar with *base* graphics. An example of how to change the default colours that ggplot2 assigns is shown below.

Use Case: Make an MA-plot of the edgeR results. Recall that this displays the average expression level on the x axis, and log fold-change on the y-axis. Modify the plot so that significant genes are shown in red.

```
ggplot(edgeRresults, aes(x = logCPM,y = logFC,col = DE)) + geom_point()
ggplot(edgeRresults, aes(x = logCPM,y = logFC,col = DE)) + geom_point(alpha=0.4) +
  scale_color_manual(values=c("black","red"))
```

Use Case: Modifty the aesthetics again so that the size of each point is relative to the level of statistical significance

```
ggplot(edgeRresults, aes(x = logCPM,y = logFC,col = DE,size=significance)) +
  geom_point(alpha=0.4) +
scale_color_manual(values=c("black","red"))
```

More information on ggplot2 is covered in the R cookbook

3.3 ggbio

We will now take a brief look at one of the visualisation packages in Bioconductor that takes advantage of the `GenomicRanges` and `GenomicFeatures` object-types. In this section we will show a worked example of how to combine several types of genomic data on the same plot. The documentation for *ggbio* is very extensive and contains lots of examples.

`http://www.tengfei.name/ggbio/docs/`

The *Gviz* package is another Bioconductor package that specialising in genomic visualisations, but we will not explore this package in the course.

The *Manhattan* plot is a common way of visualising genome-wide results, and this is implemented as the `plotGrandLinear` function. We have to supply a value to display on the y-axis, typically this is some measure of significance. Changing this variable displayed on the y-axis is done by using the `aes` function, which is inherited from *ggplot2*. The positioning of points on the x-axis is handled automatically by *ggbio*, using the ranges information to get the genomic coordinates of the ranges of interest.

Use Case: Plot the genomic locations of the significant genes on a linear scale. Modify the colour scheme to show whether each gene is up- or down-regulated

```
library(ggbio)
plotGrandLinear(sigGeneRegions , aes(y = score))
mcols(sigGeneRegions)$Up <- logfc > 0
plotGrandLinear(sigGeneRegions, aes(y = logFC, col = Up))
```

A useful function within *ggbio* is `autoplot`, which will construct an appropriate plot based on the object-type of the input. For example, *ggbio* is able to plot the structure of genes according to a particular model represented by a `GenomicFeatures` object.

For this example, we are going to pick a gene on chromosome 22 that has most evidence for differential expression at the gene-level, and is more-expressed in normals than tumours.

Use Case: What is the first gene in the list of differentially-expressed genes on chromosome 22 that is more highly-expressed in normals than tumours

```
myGene <- which(seqnames(sigGeneRegions)=="chr22" & mcols(sigGeneRegions)$logFC <0)[1]
sigGeneRegions[myGene]
```

Use Case: What is the Entrez ID for the gene you have selected? Plot the gene structure of this gene

```
entrez <- sigGeneRegions$genes[myGene]
autoplot(tx, which = exo22[[entrez]])
```

We can even plot the location of sequencing reads if they have been imported using `readGAlignments` function (or similar).

Use Case: Plot the number of reads in this region surrounding the gene.

```
myreg <- flank(reduce(exo22[[entrez]]), 1000, both = T)
bamSubset <- bam[bam %over% myreg]
autoplot(bamSubset,which=myreg)
```

Use Case: Repeat the plot, but with a smoothed coverage

```
autoplot(bamSubset , stat = "coverage")
```

Like *ggplot2*, *ggbio* plots can be saved as objects that can later be modified, or combined together to form more complicated plots. If saved in this way, the plot will only be displayed on a plotting device when we query the object. This strategy is useful when we want to add a common element (such as an ideogram) to a plot composition and don't want to repeat the code to generate the plot every time.

Use Case: Make an ideogram for Human chromosome 22 and save as an object.

```
idPlot <- plotIdeogram(genome = "hg19",subchr = "chr22")
idPlot
```

The plots produced by *ggbio* (and indeed *ggplot2* on which the package is based) are not in standard R graphics format, so techniques like `par(mfrow)` for arranging plots on the same page are not applicable.

However, *ggbio* allows the plots to be saved as *objects*, which can later be arranged by specialist functions such as `tracks`. This function automatically aligns the x-axis of each plot to be on the same scale.

Use Case: Make a combined plot of the aligned reads and transcripts for the selected gene and comment on what you see. Also show the ideogram for chromosome 22

```
geneMod <- autoplot(tx, which = myreg)
reads1 <- autoplot(bamSubset, stat = "coverage")
tracks(idPlot,geneMod ,reads1)
```

In this case, it might be of interest to compare the coverage of several samples on the same plot. One way of doing this is suggested below. Essentially, we loop for a vector of file names, and calculate the coverage of the chosen area in turn and create a plot object. We can then use the `tracks` function on the individual plots that we have saved.

```
mytracks <- alist()

fls <- paste0("Day3/",dir("Day3/",pattern=".bam"))
fls <- fls[-grep("bai",fls)]
names(fls) <- basename(fls)

for(i in 1:length(fls)){
  bam <- readGAlignments(file=fls[i],
                         param=ScanBamParam(which = GRanges("chr22",IRanges(1,51304566))),
                         use.name=TRUE)
  bamSubset <- bam[bam %over% myreg]
  mytracks[[i]] <-autoplot(bamSubset, stat="coverage") + ylim(0,20)

}
tracks(idPlot,geneMod,mytracks[[1]], mytracks[[2]],
         mytracks[[3]],mytracks[[4]],mytracks[[5]],
         mytracks[[6]])
```

fixme: Doesn't seem to display a separate legend for each bam file at the moment