Further Analysis of RNA-seq data

Mark Dunning

Last modified: March 23, 2016

Contents

1 Clustering and PCA

1.1 Introduction to the DESeq2 Vignette

The general application of clustering to RNA-seq data is outlined by Section 2.2 of the *DESeq2* vignette. We will follow this section of their vignette closely.

The DESeq2 vignette uses the *pasilla* dataset; a popular example dataset in Bioconductor concerning the RNAi knock-out of the pasilla splicing factor in Drosophilla. Full details of the dataset can be found in the vignette for the *pasilla* package. For convenience, the gene-level counts for these data are available as the `pasillaGenes` object in the *pasilla* package.

The counts and experiment metadata can be accessed using the `counts` and `pData` functions respectively.

1.2 Importing the pasilla dataset

Use Case: Load the pasilla Bioconductor package and save the counts and metadata as new R objects.

```
library("pasilla")
library("Biobase")
data("pasillaGenes")
countData <- counts(pasillaGenes)
head(countData)
pData(pasillaGenes)
colData <- pData(pasillaGenes)[,c("condition","type")]
```

A dataset ready for analysis with DESeq2 can be constructed using the `DESeqDataSetFromMatrix` function in DESeq2. You need to specify the counts matrix and experiment metadata, along with an experimental design. In this case, we want to test for *treated* versus *untreated*.

Use Case: Create a DESeq2 dataset for the pasilla dataset

```
library(DESeq2)
dds <- DESeqDataSetFromMatrix(countData = countData,
  colData = colData,
  design = ~ condition)
dds
```

In some of the analysis we are going to perform, it will be useful to supply extra annotation for the features (genes). The DESeq2 package makes extensive use of the GenomicRanges infrastructure that we introduced previously in the course. Specifically, we can add feature annotation to the `mcols` of the dataset

Use Case: Add feature annotation to the dataset

```
featureData <- data.frame(gene=rownames(pasillaGenes))
mcols(dds) <- DataFrame(mcols(dds), featureData)
dds
```

Use Case: Calculate the scaling factors for the dataset

```
dds <- estimateSizeFactors(dds)
```

1.3 Quality assessment using heatmaps, clustering and PCA

Data quality assessment and quality control (i. e. the removal of insufficiently good data) are essential steps of any data analysis. These steps should typically be performed very early in the analysis of a new data set, preceding or in parallel to the differential expression testing. We define the term quality as fitness for purpose. Our purpose is the detection of differentially expressed genes, and we are looking in particular for samples whose experimental treatment suffered from an anormality that renders the data points obtained from these particular samples detrimental to our purpose.

To produce heatmaps, we will use the *pheatmap* (*"pretty heatmap"*) package. Similar steps can be performed in the *heatmap.plus* package, the `heatmap.2` function in *gplots* or the base `heatmap` function.

Drawing a heatmap can be computationally-expensive. Also, they are more-easily interpretable when the number of rows (genes) is low. Typically we filter our dataset to only include informative genes that are expressed, or variable, in our dataset.

Use Case: Select the 20 most-highly expressed genes in the dataset

```
library("pheatmap")
N <- 20
select <- order(rowMeans(counts(dds,normalized=TRUE)),decreasing=TRUE)[1:N]
```

In order to test for differential expression, we operate on raw counts and use discrete distributions as described in the previous practical. However for other downstream analyses – e.g. for visualization or clustering – it might be useful to work with transformed versions of the count data.

Maybe the most obvious choice of transformation is the logarithm. Since count values for a gene can be zero in some conditions (and non-zero in others), some advocate the use of *pseudocounts*, i. e. transformations of the form

$$y = \log_2(n+1) \quad \text{or more generally,} \quad y = \log_2(n+n_0), \tag{1}$$

where n represents the count values and n_0 is a positive constant. The `normTransform` function implements this transformation.

Use Case: Transform the data onto a scale suitable for visualisation (e.g. $\log_2$). Extract the transformed values for the 20 most highly expressed genes

```
nt <- normTransform(dds) # defaults to log2(x+1)
log2.norm.counts <- assay(nt)[select,]
```

We are now ready to produce the heatmap. The function we are going to use is `pheatmap`. As usual, we can find out more about this function by typing `?pheatmap`. It is useful to annotate the columns (samples) in the heatmap using particular levels from the experiment metadata. `pheatmap` is able to add this information provided we supply it with a data frame with the same number of *rows* as the number of columns in our matrix. We can have as many columns of meta data as we like.

Use Case: Create a data frame that has the condition and read-type (single or paired-end) for each sample. Produce a heatmap of the normalized counts of your selected genes that incorporates the metadata for each sample.

```
df <- as.data.frame(colData(dds)[,c("condition","type")])
pheatmap(log2.norm.counts, cluster_rows=FALSE, show_rownames=FALSE,
cluster_cols=FALSE, annotation_col=df)
```

comment: We have chosen not to cluster the rows or columns, so the columns in the plot appear in the same order as the original matrix. Sometimes we might want to decide this order by clustering, as

we will do in subsequent examples.

Another use of the transformed data is sample clustering. This is done in an un-supervised manner to see if the clustering can uncover the known sample groups in our data.

We will switch our analysis to the *regularized log* transformed data, which can be computed using the `rlog` function. This transformation is described in detail in Section 2.1.3 of the DESeq2 vignette and is more similar to the transformations that take place as part of the DESeq2 analysis workflow for Differential expression analysis. The `dist` function can then be use to calculate pairwise distances between all samples. We have to remember to *transpose* the transformed matrix using the `t` function.

Use Case: Compute the regularized log intensities and use these to calculate a distance matrix. What metric is being used to calculate the distances?

```
rld <- rlog(dds)
rlog.intensities <- assay(rld)
head(rlog.intensities)
sampleDists <- dist(t(rlog.intensities))
sampleDists
```

If we wanted, we could use the standard hierachical clustering function in R, `hclust`, to cluster the samples.

Use Case: Produce a dendrogram to visualise the clustering of the samples. Does it confirm the known sample groups?

```
hc <- hclust(sampleDists)
plot(hc)
```

comment: If you have time, you can experiment with different distance metrics and clustering algorithms

A more-appealing visualisation can be produced by `pheatmap`. In the previous example, we told `pheatmap` to retain the original column and row orders. Another way of using `pheatmap` is to supply a pre-computed distance matrix.

Use Case: Produce a heatmap to visualise the sample relationships from the distance matrix that you just computed.

```
sampleDistMatrix <- as.matrix(sampleDists)
rownames(sampleDistMatrix) <- paste(rld$condition, rld$type, sep="-")
colnames(sampleDistMatrix) <- NULL


pheatmap(sampleDistMatrix,
clustering_distance_rows=sampleDists,
clustering_distance_cols=sampleDists)
```

Another way of customising the heatmap is to specify the colour palette used to colour each cell in the matrix. Many visually-appealing palettes are provided in the *RColorBrewer* package.

Use Case: Load the *RColorBrewer* package and see what palettes are available

```
library("RColorBrewer")
display.brewer.all()
```

The `colorRampPalette` function can be used to interpolate a set of given colours

Use Case: Create a palette that ranges from dark-blue (low) to white (high) and use this palette in the heatmap

```
colors <- colorRampPalette(rev(brewer.pal(9, "Blues")) )(255)
pheatmap(sampleDistMatrix,
clustering_distance_rows=sampleDists,
clustering_distance_cols=sampleDists,
col=colors)
```

comment: You should make sure that the colour palette you choose are suitable for those with colour-blindness

1.4 PCA

Principal Components Analysis (PCA) is a useful dimension-reduction technique that can highlight relationships between samples in our dataset. As it can be computationally-expensive, it is best performed on a filtered version of the data. Typically, we pick the "most-variable" genes. The `rowVars` function in *genefilter* is useful for this task. DESeq2 actually provides a function that automates the steps required to do a PCA analysis of a `DESeqDataSet` object and produce an informative plot. However, we will go through the steps manually to illustrate the method.

Use Case: Create a vector of indices for the 500 most variable genes according to the `rld` data

```
library(genefilter)
rv <- rowVars(assay(rld))
select <- order(rv, decreasing=TRUE)[1:500]
```

PCA can be performed using the `prcomp` function. As when computing a distance matrix, we need to remember to *transpose* our matrix of intensities if we want to do PCA on the samples.

Use Case: Perform PCA on the filtered data. How much variance is explained by the first two principal components?

```
pca <- prcomp(t(assay(rld)[select,]))
summary(pca)
plot(pca)
```

Use Case: Produce a plot of the first two principal components and colour each point according to whether the particular sample is "treated" or "untreated. Does the plot show clear separation of sample groups?

```
pca$x
sampcols <- c(rep("blue",3),rep("red",4))
plot(pca$x[,1], pca$x[,2],pch=16,col=sampcols)
```

Remember that the actual values of the Principal components cannot be readily interpreted. We can however plot them in relation to know sample grouping and metadata to ease their interpretation. A boxplot is useful for this task.

Use Case: Use a boxplot to visualise the values of the first two principal components in relation to the sample condition and type variables

```
colData(dds)
boxplot(pca$x[,1] ~ colData(dds)[,"condition"])
boxplot(pca$x[,2] ~ colData(dds)[,"type"])
```

At this point, we introduce the `plotPCA` function which automates the steps we have just performed and produces and attractive plot. The plot is produced by the popular *ggplot2*, and later-on we will see how to customise this plot

Use Case: Use the in-built `plotPCA` function to produce the PCA plot

```
plotPCA(rld, intgroup=c("condition", "type"))
```

1.5 Re-visiting our ESCC dataset

1.5.1 Pre-processing

In the previous DE practical, we have already gone through the processing of this dataset. We can repeat the steps now, if you have closed your RStudio session since the DE practical.

```
library(DESeq2)
load("Day2/Counts.RData")
#Load data
Counts <- tmp$counts
colnames(Counts) <- c("16N", "16T", "18N", "18T", "19N", "19T") #Rename the columns
Coldata <- data.frame(sampleReplicate=c("16", "16", "18", "18", "19", "19"),
sampleType=c("N", "T", "N", "T", "N", "T"))
rownames(Coldata) <- c("16N", "16T", "18N", "18T", "19N", "19T")

deSeqData <- DESeqDataSetFromMatrix(countData=Counts, colData=Coldata,
              design= ~sampleReplicate + sampleType)
deSeqData <- deSeqData[rowSums(counts(deSeqData))>1,]
deSeqData <- estimateSizeFactors(deSeqData)
```

1.5.2 Sample Count Heatmap

Use Case: Produce a heatmap of the normalized counts of the most expressed genes in the ESCC dataset

```
nt <- normTransform(deSeqData) # defaults to log2(x+1)
select <- order(rowMeans(counts(deSeqData,normalized=TRUE)),decreasing=TRUE)[1:20]

log2.norm.counts <- assay(nt)[select,]
df <- as.data.frame(colData(deSeqData)[,c("sampleReplicate","sampleType")])
pheatmap(log2.norm.counts, cluster_rows=FALSE, show_rownames=FALSE,
cluster_cols=FALSE, annotation_col=df)
```

```
pheatmap(log2.norm.counts, cluster_rows=FALSE, show_rownames=FALSE,
cluster_cols=TRUE, annotation_col=df)
```

1.5.3 Sample Distances Heatmap

Use Case: Perform the `rlog` transformation and make a heatmap of the pairwise sample distances.

```
rld <- rlog(deSeqData)
sampleDists <- dist(t(assay(rld)))
sampleDists

sampleDistMatrix <- as.matrix(sampleDists)
rownames(sampleDistMatrix) <- paste(rld$sampleReplicate, rld$sampleType, sep="-")
colnames(sampleDistMatrix) <- NULL
colors <- colorRampPalette( rev(brewer.pal(9, "Blues")) )(255)
pheatmap(sampleDistMatrix,
clustering_distance_rows=sampleDists,
clustering_distance_cols=sampleDists,
col=colors)
```

1.5.4 PCA

Use Case: Perform a PCA analysis and visualise the results

```
plotPCA(rld, intgroup=c("sampleType","sampleReplicate"))
```

The PCA plot is configurable provided we know a tiny bit about *ggplot2*. The DESeq2 authors recognise that some might want more control over the PCA plots, so provide a way of accessing the data that would be displayed on the plot with the `returnData=TRUE` argument.

Use Case: Re-do the PCA analysis, but this time save the PCA results as an object

```
pcData <- plotPCA(rld, intgroup=c("sampleType","sampleReplicate"),returnData=TRUE)
pcData
```

Use Case: Visualise the PCA result, but colour each point according to sample group and plot a different shape for each patient in the study

```
library(ggplot2)
percentVar <- round(100*attr(pcData, "percentVar"))
ggplot(pcData, aes(x=PC1,y=PC2,color=sampleType,shape=sampleReplicate))+
  geom_point(size=5)+
  xlab(paste0("PC1: ", percentVar[1], "% variance")) +
  ylab(paste0("PC2: ", percentVar[2], "% variance"))
```

1.5.5 Heatmap of DE genes

Sometimes we might want to take the list of top genes from a DE analysis and use this to produce a heatmap.

Use Case: Re-do the DE analysis using the workflow introduced in the DE analysis practical

```
deSeqData <- estimateDispersions(deSeqData)
mcols(deSeqData)
deSeqData <- nbinomWaldTest(deSeqData)
res <- results(deSeqData)
```

Use Case: Select a subset of the DE results with a p-value less than 0.05.

```
res.sig <- res[which(res$padj < 0.05),]
N <- 100
res.sig.ord <- res.sig[order(res.sig$padj,decreasing = FALSE),]
topNGenes <- rownames(res.sig.ord)[1:N]
```

Use Case: Make a heatmap of your top 100 genes. Can you see any distinct clusters of genes in the heatmap?

```
pheatmap(assay(rld)[match(topNGenes, rownames(assay(nt))),],annotation_col=df)
```

If we identify genes with distinct expression patterns, a natural question would be to ask what genes belong in each cluster. We could export the plot as a pdf and try and read the gene names from the row labels. We can also answer this by recalling that the `pheatmap` (unless specified otherwise) is infact using the built-in clustering options in R to cluster the rows. There exist a number of ways that we can "cut" such a dendrogram in R.

Use Case: Produce a dendrogram to show the clustering of the top DE genes.

```
geneDists <- dist(assay(rld)[match(topNGenes, rownames(assay(nt))),])

geneDistMatrix <- as.matrix(geneDists)
```

```
hc <- hclust(geneDists)

plot(hc)
```

The `rect.hclust` and `cutree` functions can be used to split the dendrogram at different heights and number of groups.

Use Case: Use `rect.hclust` and `cutree` to see what genes belong to the same cluster. Choose an appropriate value for k; the number of clusters

```
rect.hclust(hc,k=3)
sort(cutree(hc,k=3))
```

1.5.6 Heatmap of particular gene set

Let's say we are interested in genes belonging to a particular GO term; GO:0030216 (keratinocyte differentiation). We can use the organism-level packages that we just learnt about to retrieve the IDs of genes that belong to this GO term.

Use Case: Obtain the Entrez IDs for the GO term of interest

```
library(org.Hs.eg.db)
keytypes(org.Hs.eg.db)
columns(org.Hs.eg.db)
genes.of.interest <- select(org.Hs.eg.db,keys="GO:0030216",
                            keytype="GO",columns="ENTREZID")
genes.of.interest

genes.of.interest <- genes.of.interest[,4]
```

We might not have all the genes representing in our dataset, particularly if we have done some filtering. So we probably want to remove any IDs that cannot be matched to our dataset. Also we want to remove any duplicated IDs.

Use Case: Make a heatmap from your selected genes

```
selRows <- unique(na.omit(match(genes.of.interest,rownames(assay(nt)))))

pheatmap(assay(rld)[selRows,],annotation_col=df)
```

2 GO analysis

(based on the *goseq* vignette)

There are plenty of software packages in Bioconductor and online tools that deal with gene ontology testing. For this practical, we will use the *goseq* package which has been specifically-developed for the gene onotology analysis of RNA-seq data. An attractive feature of this package is that it accounts for biases in gene length. The methods are equally applicable to other category based testing, such as KEGG pathways.

The input to the package is quite simple. We need to supply a named vector which contains two pieces of information.

- Measured genes - all genes for which RNA-seq data was gathered for your experiment. Each element of your vector should be named by a unique gene identifier
- Differentially expressed genes - each element of your vector should be either a 1 or a 0, where 1 indicates that the gene is differentially expressed and 0 that it is not

Provided that a *supported* genone and naming scheme is being used (we will see how to obtain details on these shortly), *goseq* should be able to proceed with the analysis.

Use Case: Load the object that contains the DE results for all genes.

```
load("Day3/edgeRAnalysis_ALLGENES.RData")
head(y)
```

The next step is to determine which genes are DE. We don't have to be too strict in our selection and can use a typical cut-off of 0.05.

Use Case: Create a vector to signify whether each gene in the analysis was DE or not.

```
genes <- as.integer(y$FDR < 0.05)
names(genes) <- rownames(y)
```

goseq is happy to take care of the mapping between gene names and GO categories for us. However, in order to do this, we need to use one of the genomes and identifier schemes that it knows about. The `supportedGenomes` and `supportedGeneIDs` functions are provided for this purpose. The result of each these functions is a table describing the various options that are available. The first columns are pre-defined names that `goseq` can recognise. The other columns should give you enough information to guide your choice.

Use Case: Check the names of the genomes that are supported by goseq. Make a note of the IDs that you should use.

```
library(goseq)
head(supportedGenomes())[,1:4]
head(supportedGeneIDs(),n=12)[,1:4]
```

comment: For our example dataset, the genome version was `hg19` *and each gene was represented by it's* Entrez *ID; so we would want to specify* `hg19` *and* `knownGene`.

"To begin the analysis, goseq first needs to quantify the length bias present in the dataset under consideration. This is done by calculating a Probability Weighting Function or PWF which can be thought of as a function which gives the probability that a gene will be differentially expressed (DE),

based on its length alone. The PWF is calculated by fitting a monotonic spline to the binary data series of differential expression (1=DE, 0=Not DE) as a function of gene length. The PWF is used to weight the chance of selecting each gene when forming a null distribution for GO category membership. The fact that the PWF is calculated directly from the dataset under consideration makes this approach robust, only correcting for the length bias present in the data. For example, if goseq is run on a microarray dataset, for which no length bias exists, the calculated PWF will be nearly flat and all genes will be weighted equally, resulting in no length bias correction. In order to account for the length bias inherent to RNA-seq data when performing a GO analysis (or other category based tests), one cannot simply use the hypergeometric distribution as the null distribution for category membership, which is appropriate for data without DE length bias, such as microarray data. GO analysis of RNA-seq data requires the use of random sampling in order to generate a suitable null distribution for GO category membership and calculate each categories significance for over representation amongst DE genes.

However, this random sampling is computationally expensive. In most cases, the Wallenius distribution can be used to approximate the true null distribution, without any significant loss in accuracy. The goseq package implements this approximation as its default option. The option to generate the null distribution using random sampling is also included as an option, but users should be aware that the default number of samples generated will not be enough to accurately call enrichment when there are a large number of go terms. Having established a null distribution, each GO category is then tested for over and under representation amongst the set of differentially expressed genes and the null is used to calculate a p-value for under and over representation."

Use Case: Fit the PWF function to your dataset. Make sure you choose the correct names for the organism and gene identifier.

```
pwf <- nullp(genes ,"hg19","knownGene")
head(pwf)
```

`nullp` plots the resulting fit, allowing verification of the goodness of fit before continuing the analysis. Further plotting of the pwf can be performed using the `plotPWF` function.

The output of `nullp` contains all the data used to create the PWF, as well as the PWF itself. It is a data frame with 3 columns, named `DEgenes`, `bias.data` and `pwf` with the rownames set to the gene names. Each row corresponds to a gene with the DEgenes column specifying if the gene is DE (1 for DE, 0 for not DE), the bias.data column giving the numeric value of the DE bias being accounted for (usually the gene length or number of counts) and the pwf column giving the genes value on the probability weighting function.

We will use the default method, to calculate the over and under expressed GO categories among DE genes. Again, we allow goseq to fetch data automatically, except this time the data being fetched is the relationship between Entrez gene IDs and GO categories.

Use Case: Perform under- and over-representation analysis. How many GO categories and over-represented with a p-value of less than 0.05?

```
go <- goseq(pwf, "hg19","knownGene")
head(go)
sum(go$over_represented_pvalue< 0.05)
```

By default, goseq tests all three major Gene Ontology branches; Cellular Components, Biological Processes and Molecular Functions. However, it is possible to limit testing to any combination of the major branches by using the test.cats argument to the goseq function. This is done by specifying a vector consisting of some combination of the strings GO:CC, GO:BP and GO:MF.

Use Case: Restrict your analysis to just Molecular Functions

```
go.mf <- goseq(pwf, "hg19","knownGene",test.cats=c("GO:MF"))
head(go.mf)
dim(go.mf)
sum(go.mf$over_represented_pvalue< 0.05)
```

Having performed the GO analysis, you may now wish to interpret the results. If you wish to identify categories significantly enriched/unenriched below some p-value cutoff, it is necessary to first apply some kind of multiple hypothesis testing correction. For example, GO categories over enriched using a .05 FDR cutoff [Benjamini and Hochberg, 1995] can be obtained by running the `p.adjust` function on the un-adjusted p-values in the table

Use Case: Adjust the p-values in the table for multiple-testing. How many categories with p-value $<$ 0.05 do you find now?

```
go$over_represented_pvalue_adjusted <- p.adjust(go$over_represented_pvalue,
                                                method="BH")
sum(go$over_represented_pvalue_adjusted < 0.05)

go.enriched <- go[go$over_represented_pvalue_adjusted < 0.05,]
```

The GO identifiers are probably not very useful to you. We can obtain extra information about each GO term using one of the handy pre-built database packages that Bioconductor provides; *GO.db*.

Use Case: Load the *GO.db* package. What mappings are possible with this package?

```
library(GO.db)
columns(GO.db)
keytypes(GO.db)
```

Use Case: Create a data frame with the definition and term for each GO ID in your table of results

```
go.anno <- select(GO.db,keys=go.enriched$category, keytype="GOID",
                  columns = c("TERM","DEFINITION"))
```

Finally, we can merge this data with our existing table and export

```
go.final <- data.frame(go.enriched,go.anno)
write.csv(go.final ,file="goseq-analysis.csv",row.names = FALSE)
```