**How to use DESeq to analyse RNAseq data**

Posted on [April 15, 2013](http://dwheelerau.com/2013/04/15/how-to-use-deseq-to-analyse-rnaseq-data/)

Several packages have been written to measure gene expression differences using RNAseq data. In this post I’m going to do a run through of the popular R based [DESeq](http://www-huber.embl.de/users/anders/DESeq/) package. Basically, I’m just following along with the packages Vignette which is really well written and informative! For the test data I’m using 2 treatments, with 3 biological reps each containing 750K reads (yes its small).

**A. Counting reads**

DEseq uses a table of raw counts as the basis of DE calls. Any transformed counts, such as FPKM counts will not work with this package. The top row of the table can be a descriptor, these can be imported into R to act as column labels. Gene names should be column 1 (remember R counting starts from 1 rather than 0), each column after that should be a raw count for each biological replicate and for each treatment. Note that if technical reps are performed, these should be merged based on the sum of the two runs (they will be normalised by read count latter).

Here I used BWA to align the reads to my reference and generate SAM files that are then processed by [HTseq](http://www-huber.embl.de/users/anders/HTSeq/doc/overview.html) count. Note that you have to run this on each sample, so in this current example I will generate 6 sam files, 1 for each treatment and condition. Also, see that I used the [intersection-strict](http://www-huber.embl.de/users/anders/HTSeq/doc/count.html) mode for HTseq.

>bwa aln sarcop\_pep\_rnaseqfiltered.fa Sample\_ST72-1/ST72-1.subset.fastq > ST72-1.subset.sai

>bwa samse sarcop\_pep\_rnaseqfiltered.fa ST72-1.subset.sai Sample\_ST72-1/ST72-1.subset.fastq > ST72-1.subset.sam

>htseq-count -m intersection-strict –stranded=no ST72-1.subset.sam sarcop\_pep\_rnaseqfiltered.gtf > ST72-1.subset.counts

In the above ST72-1.subset.counts is a table of counts that can be merged with the other count data before being imported into DEseq as a R data table. Here is a [python script](https://docs.google.com/file/d/0B99ewTsEZTkidElVekliSUxldW8/edit?usp=sharing) for merging, see the header of the script for usage, if that helps. For full detailed explanation of HTseq see (<http://www-huber.embl.de/users/anders/HTSeq/doc/install.html#installation-on-linux>) and this seems to be the default way of doing it (instructions :<http://www-huber.embl.de/users/anders/HTSeq/doc/count.html>).

**B. R DESeq package**

Enter R environment from a terminal window open where your data is.

>R

>library(“DESeq”)

>countsTable <- read.delim(“merged\_counts.txt”,header=TRUE)

>rownames(countsTable) <- countsTable$gene

>countsTable <- countsTable[,-1]

This imports the DESeq library, reads the tab delineated file created by the python script into a data table. The next two lines create row names from the gene column created by the script, if you use some other method to make the counts you need to change the name of the gene column to whatever it is. Finally, we get rid of the column used to make the row names. My table looks like this

> head( countsTable)

                  UN72\_1 UN72\_2 UN72\_3 ST72\_1 ST72\_2 ST72\_3

GWD8ZKM01ESPR2          0      0      0      0      0      0

g10671t00001            2      1      1      0      0      0

g16789t00001            0      0      0      0      0      0

Now we need to add the treatment information so that DESeq knows what data each column represents, order is obviously important.

>conds <- factor( c( “untreated”, “untreated”, “untreated”, “treated”,”treated” ,”treated”  ) )

> cds <- newCountDataSet( countsTable, conds )

This has created a instantiate a CountDataSet, which is the central data structure in the DESeq. The next step will adjust for different library sizes among the treatments, something called “size factors”. DESeq only needs relative library sizes, so if (on average) the counts for non-DE genes in one library are twice that of a second library the size factor should be twice as large.

>cds <- estimateSizeFactors( cds )

To see these size factors, do this:

>sizeFactors( cds )

UN72\_1    UN72\_2    UN72\_3    ST72\_1    ST72\_2    ST72\_3

0.9805609 1.0420015 0.9164864 1.1083583 0.9599370 1.1462168

Apparently, the way this works (<http://www.biostars.org/p/12273/>) is that you take the genometric mean of each condition for a gene and use this as its reference value. Then you divide the gene expression value for each conditions by its reference value to create a list of values (quotient ie result of the division) for each condition. The size factor is the median value of this list, as there is a list per condition it will generate a size factor value for each condition. The counts are normalised by dividing each column by its size factor, here lets look before and after:

> head(counts(cds))

                  UN72\_1 UN72\_2 UN72\_3 ST72\_1 ST72\_2 ST72\_3

GWD8ZKM01ESPR2          0      0      0      0      0      0

g10671t00001            2      1      1      0      0      0

> head(counts(cds,normalized=TRUE))

                    UN72\_1    UN72\_2   UN72\_3 ST72\_1 ST72\_2 ST72\_3

GWD8ZKM01ESPR2     0.000000 0.0000000 0.000000      0      0      0

g10671t00001       2.039649 0.9596916 1.091124      0      0      0

g16789t00001       0.000000 0.0000000 0.000000      0      0      0

The next part is a key to how DESeq calculates significance in DE. The package uses the relationship between the data’s variance (or dispersion) and its mean. The inference in DESeq relies on an estimation of the typical relationship between the data’s variance and their mean, or, equivalently, between the data’s dispersion and their mean. The dispersion can be understood as the square of the coefficient of biological variation. The variance between counts is the sum of two factors, firstly, the level of variation between replicates of each treatment, and secondly an uncertainty measure based on the concentration of the counts. This latter factor takes into consideration the level of noise predicted for low expressed genes based on a poisson distribution (see the [DEseq](http://www-huber.embl.de/users/anders/DESeq/) manual for more detail).

>cds <- estimateDispersions( cds )

Now we write a little function that will plot this.

>plotDispEsts <- function( cds ){

plot(rowMeans( counts( cds, normalized=TRUE ) ), fitInfo(cds)

$perGeneDispEsts, pch = ‘.’, log=”xy”, ylab=”dispersion”,

xlab=”mean of normalized counts”)

xg = 10^seq( -.5, 5, length.out=300 )

lines( xg, fitInfo(cds)$dispFun( xg ), col=”red” )

}

>plotDispEsts(cds)

> jpeg(“DispEsts\_DESeq.jpg”)

> plotDispEsts(cds)

> dev.off()

The Above commands make a plotting function, the pass cds to that function and save the result in a jpg file.

The empirical (real) dispersion values (black dots) and fitted values (red line)

Remember that the level of dispersion is related to the biological variation seen in each treatment. Note this is a double log graph, ahhh my brain hurts, but look carefully at what is happening on the Y axis, as read count increases dispersion decreases, as we would expect. However, a lot of those black dot values below the red fitted line are probably underestimates of dispersion based on the small samples sizes used (only 3 replicates values measured in this example). So, to be conservative DESeq moves all these values below the red line up to the fitted value, BUT keeps all the empirical values above the red line, even if some of these are over estimates of dispersion, as I said it is being conservative. Why does dispersion matter? The more dispersion, or biological variation, the bigger the difference between counts between treatments is required before it differences become significant!

Finally we are ready to call DE values!

>res = nbinomTest( cds, “untreated”, “treated” )

>head(res)

                 id  baseMean baseMeanA baseMeanB foldChange log2FoldChange

1     GWD8ZKM01ESPR2 0.0000000 0.0000000         0        NaN            NaN

2       g10671t00001 0.6817440 1.3634880         0          0           -Inf

3       g16789t00001 0.0000000 0.0000000         0        NaN            NaN

      pval padj

1        NA   NA

2 0.2083517    1

3        NA   NA

Now lets plot the log2 fold changes (untreated versus treated) against the mean normalised counts, colouring in red those genes that are significant at 10% FDR.

jpeg(“plotMA.jpg”)

> plotMA(res)

> dev.off()

Note how lower lower expressed genes need a bigger log2 fold change value to be made red (not so obvious in this data due to low number of reads used in mapping).

Lets explore this a little more by comparing the normalized counts between all three replicates of just the untreated sample.

>ncu = counts( cds, normalized=TRUE )[ , conditions(cds)==”untreated” ]

>jpeg(“MA\_untreated\_only.jpg”)

>plotMA(data.frame(baseMean = rowMeans(ncu),log2FoldChange = log2( ncu[,2] / ncu[,1] )),

col = “black”)

>dev.off()

Wow, look at all the DE in a single treatment,  thus we should call a gene DE only if the change between treated and untreated samples is stronger than what we see between replicates, and hence, the dividing line between red and black in the first figure mimics that for the second.

And now a histogram of p values to look at their distribution.

hist(res$pval, breaks=100, col=”skyblue”, border=”slateblue”, main=””)

> jpeg(“hist\_pval.jpg”)

> hist(res$pval, breaks=100, col=”skyblue”, border=”slateblue”, main=””)

> dev.off()

The pileup of very low p-values are the DE genes, the pileups at the right hand side of the distribution are from very low count genes that are given discrete values.

Now lets create some subset tables based around FDR cut offs (in this case 10%)

>resSig = res[ res$padj < 0.1, ]

>head(resSig)

Before we make any mistakes, lets save the data

>write.csv( res, file=”72hour\_unstgVstg\_all.csv” )

>write.csv( resSig, file=”72hour\_unstgVstg\_only\_0.01FDR.csv” )

We are done with the basic analysis, we can do some more sophisticated filtering of the data. The main goal is to remove tests that give little chance of producing a meaningful result, with the aim of keeping the FDR in check. This results in an increased power to detect significant results whilst keeping the same level of type I error the same (false positive rate).

>rs = rowSums ( counts ( cds))

>theta = 0.4

>use = (rs > quantile(rs, probs=theta))

>table(use)

table(use)

FALSE  TRUE

30145 36029

Basically here we are summing the counts across all conditions for each gene, then remove the genes with the lowest 40 % quantile (basically filtering out those that are false above from the original normalised cds table)

>cdsFilt = cds[ use, ]

Now just repeat the tests on the filtered data

>resFilt = nbinomTest( cdsFilt, “untreated”, “treated” )

Now lets look at the difference this makes to the FDR.

tabAll <-table(res$padj<0.1)

tabFilt <-table(resFilt$padj<0.1)

> tabFull

FALSE  TRUE

35266   763

> tabFilt

FALSE  TRUE

35266   763

In this case it didn’t seem to make much difference because most of the dropped values were NA! Just to visualise what is going on here lets plot the un-ajusted pvals of the ~40% quantile that we filtered.

>jpeg(“pval\_bot40.jpg”)

> plot(rank(rs)/length(rs), -log10(res$pval), pch=16, cex=0.45)

> dev.off()

Note that very few p value are less than 0.003 (this corresponds to about 2.5 on the − log10 scale), and this is demonstrated when we plot the pvals of each dataset.

> h1 = hist(res$pval,breaks=50,plot=FALSE)

> h2 = hist(resFilt$pval,breaks=50,plot=FALSE)

>colori <- c(‘do not pass’=”khaki”, ‘pass’=”powderblue”)

> barplot(height = rbind(h1$counts, h2$counts), beside = FALSE, col = colori,

+ space = 0, main = “”, ylab=”frequency”)

> text(x = c(0, length(h1$counts)), y = 0, label = paste(c(0,1)), adj = c(0.5,1.7), xpd=NA)

> legend(“topright”, fill=rev(colori), legend=rev(names(colori)))

> dev.copy(png,’hist\_filthist.png’)

> dev.off()

Now we are going to process the data further, since some counts and be zero in some applications in makes sense to add psuedocounts (ie y = log2 (n + n0 ) where n0 is a small ‘chosen’ constant. The question is what is the value of the small number you add to the zero counts to make them non-zero (but keep them very small). DEseq uses a alternative approach.

>cdsBlind = estimateDispersions( cds, method=”blind” )

>vsd = varianceStabilizingTransformation( cdsBlind )

Now we will plot Per-gene standard deviation (taken across samples), against the rank of the mean, the standard deviation of the transformed data, across samples, against the mean, first using the shifted logarithm transformation (notAllZero is basically one of the numbers in the row is NOT ZERO), with 1 added to all values to remove zeros), then using DESeq’s variance stabilising transformation. While we are meant to see a constant standard deviation for the DESeq method along the whole dynamic range, with the shifted logarithm giving a highly elevated standard deviation in the lower count range for the other method, in fact they both look pretty similar for this small dataset.

>biocLite(“vsn”)

>library(“vsn”)

>par(mfrow=c(1,2))

>notAllZero = (rowSums(counts(cds))>0)

>meanSdPlot(log2(counts(cds)[notAllZero, ] + 1), ylim = c(0,2.5))

>meanSdPlot(vsd[notAllZero, ], ylim = c(0,2.5))

Now to deal with moderate counts that generate highly variable log fold change values and in some cases they become infinite. This will mess up any kind of clustering we may want to perform to identify expression common patterns. We can compare their ordinary log-ratios (lfc) against the moderated log-ratios (mod\_lfc) using a scatterplot, with the genes bined by their log10 mean to colour them according to expression strength.

>mod\_lfc = (rowMeans( exprs(vsd)[, conditions(cds)==”treated”, drop=FALSE] ) – rowMeans( exprs(vsd)[, conditions(cds)==”untreated”, drop=FALSE] ))

> lfc = res$log2FoldChange

> table(lfc[!is.finite(lfc)], useNA=”always”)

-Inf   Inf   NaN  <NA>

5085  4900 30145     0

Many of the latter are infinite (Inf,division of a finite value by 0) or not a number (NaN, resulting from division of 0 by 0).

> logdecade = 1 + round( log10( 1+rowMeans(counts(cdsBlind, normalized=TRUE)) ) )

> lfccol = colorRampPalette( c( “gray”, “blue” ) )(6)[logdecade]

> ymax = 4.5

> plot( pmax(-ymax, pmin(ymax, lfc)), mod\_lfc,

xlab = “ordinary log-ratio”, ylab = “moderated log-ratio”,

cex=0.45, asp=1, col = lfccol,

pch = ifelse(lfc<(-ymax), 60, ifelse(lfc>ymax, 62, 16)))

> abline( a=0, b=1, col=”red3″)

Here grey to blue represents weakly to strongly expressed genes, for highly expressed genes there is general agreement (along diag) but for lowly expressed genes the moderated log-ratio is shrunk towards 0, compared to the ordinary log-ratio (the grey smear flattens out along the horizontal (0 value on the y-axis).

Now to look at heat maps of counts to see any patterns using the transformed  log data.

>library(“gplots”)

>library(“RColorBrewer”)

> cdsBlind = estimateDispersions(cds,method=”blind”)

> vsd = varianceStabilizingTransformation(cdsBlind)

>select = order(rowMeans(counts(cds)), decreasing=TRUE)[1:30]

>hmcol = colorRampPalette(brewer.pal(9, “GnBu”))(100)

>heatmap.2(exprs(vsd)[select,], col = hmcol, trace=”none”, margin=c(10, 6))

and for a laugh the untransformed data.

Now for a

>dists = dist( t( exprs(vsd) ) )

>mat = as.matrix( dists )

> rownames(mat) = colnames(mat) = with(pData(cdsBlind), paste(condition))

> heatmap.2(mat, trace=”none”, col = rev(hmcol), margin=c(13, 13))

No shocks here again. Of note is that we used the method “blind”, thus the variance stabilizing transformation is not informed about the design to prevent bias in the outcome. In comparison, the pooled method (the default for estimating the dispersion) “means that one dispersion is computed for each gene, which is now an average over all cells (weighted by the number of samples for each cells), where the term cell denotes any of the four combinations of factor levels of the design.”

Now a good olde PCA, pretty cool really.

>print(plotPCA(vsd, intgroup=c(“condition”)))

This type of plot is useful for visualizing the overall effect of experimental covariates and batch effect, so you might expect known conditions to group, ie treatment and any time of library prep effects (paired end reads versus single end data). Good to see that even with this subset of data biggest component is treatment (pc1).

>sessionInfo()

R version 2.15.3 (2013-03-01)

Platform: x86\_64-pc-linux-gnu (64-bit)

locale:

[1] LC\_CTYPE=en\_US.UTF-8       LC\_NUMERIC=C

[3] LC\_TIME=en\_US.UTF-8        LC\_COLLATE=en\_US.UTF-8

[5] LC\_MONETARY=en\_US.UTF-8    LC\_MESSAGES=en\_US.UTF-8

[7] LC\_PAPER=C                 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      stats     graphics  grDevices utils     datasets  methods

[8] base

other attached packages:

[1] arrayQualityMetrics\_3.14.0 gplots\_2.10.1

[3] KernSmooth\_2.23-10         caTools\_1.14

[5] gdata\_2.12.0.2             gtools\_2.7.1

[7] RColorBrewer\_1.0-5         vsn\_3.26.0

[9] BiocInstaller\_1.8.3        DESeq\_1.10.1

[11] lattice\_0.20-15            locfit\_1.5-9

[13] Biobase\_2.18.0             BiocGenerics\_0.4.0

loaded via a namespace (and not attached):

[1] affy\_1.36.1           affyio\_1.26.0         affyPLM\_1.34.0

[4] annotate\_1.36.0       AnnotationDbi\_1.20.7  beadarray\_2.8.1

[7] BeadDataPackR\_1.10.0  Biostrings\_2.26.3     bitops\_1.0-5

[10] Cairo\_1.5-2           cluster\_1.14.4        colorspace\_1.2-1

[13] DBI\_0.2-5             gcrma\_2.30.0          genefilter\_1.40.0

[16] geneplotter\_1.36.0    Hmisc\_3.10-1          hwriter\_1.3

[19] IRanges\_1.16.6        latticeExtra\_0.6-24   limma\_3.14.4

[22] parallel\_2.15.3       plyr\_1.8              preprocessCore\_1.20.0

[25] reshape2\_1.2.2        RSQLite\_0.11.2        setRNG\_2011.11-2

[28] splines\_2.15.3        stats4\_2.15.3         stringr\_0.6.2

[31] survival\_2.37-4       SVGAnnotation\_0.93-1  tcltk\_2.15.3

[34] tools\_2.15.3          XML\_3.6-2             xtable\_1.7-1

[37] zlibbioc\_1.4.0

>