Finding Differentially Expressed Genes: A Novice's Experience with DESeq2

John F. Ouyang
john.f.ouyang@gmail.com

September 17, 2017

Contents

Abstract

This guide documents the use of DESeq2 to perform differential gene expression analysis from a novice perspective. This entire procedure can be broken down into several steps, namely (i) preparation of input data, (ii) quality control, (iii) running DESeq2.

Throughout this guide, paragraphs beginning with "**Extra Thoughts**" highlights deeper questions that can arise from some of the data treatment. Furthermore, paragraphs beginning with "**Coding**" highlight things to look out for when coding DESeq2.

1 Input Data

To perform DESeq2, we need to prepare two files: (i) “sample_details.tab” containing the description of the samples and (ii) “raw_counts.tab” containing the CAGE counts. The two files will be linked via the RIKEN ID, an unique identifier of the form CNhsNNNNN. The RIKEN ID, sample description and CAGE counts are all contained in a single file which is publically available on FANTOM’s website (http://fantom.gsc.riken.jp/5/):

```
# these commands are executed on the command line
wget http://fantom.gsc.riken.jp/5/datafiles/latest/extra/CAGE_peaks/
  hg19.cage_peak_phase1and2combined_counts_ann.osc.txt.gz

gzip -d hg19.cage_peak_phase1and2combined_counts_ann.osc.txt.gz
```

1.1 Preparing Sample Details

For this guide, we are going to find the differentially expressed genes between dermal fibroblast and embryonic stem cells. To this end, we wrote a perl script to extract the description for *all* the samples from the downloaded file `hg19....` We then select for the samples of interest, namely sample descriptions that contain `Fibroblast_-_Dermal` and `H9_Embryonic_Stem_cells`.

```
# these commands are executed on the command line
./extract_description.pl hg19.cage_peak_phase1and2combined_counts_ann.osc.
  txt > out_sample.txt

grep -e 'Fibroblast_-_Dermal' -e 'H9_Embryonic_Stem_cells' out_sample.txt
  > sample_details.tab
```

The perl script `extract_description.pl` is presented below. To understand this perl script, we first discuss the format of the downloaded file `hg19....` The first few lines of the file contains sample description as well as experimental details. Here, we are interested in the line beginning with `00Annotation`. The sample descriptions then begin on the seventh column of the line. An example description looks like this: `counts.Fibroblast%20-%20Dermal%2c%20donor1.CNhs12499.11269-116G9`. The perl script thn extracts the second and third part (delimited by .), corresponding to the description of interest and RIKEN id.

```
#!/usr/bin/perl -w
use strict;
use warnings;

my $filename = $ARGV[0];
my @sample_names;
open(FILE,"$filename");
```

```
while(<FILE>){
    chomp;
    my $line = $_;
    if($line =~ /^00Annotation/){
        @sample_names = split(/\t/,$line);
        @sample_names = @sample_names[7..scalar(@sample_names)-1];
    }
}
my @line;
my $idx = 0;
while($idx < scalar(@sample_names)){
    @line = split(/\./,$sample_names[$idx]);
    $line[1] =~ s/%../_/g;
    print $line[2]."\t".$line[1]."\n";
    $idx++;
}
```

1.2 Preparing Count Matrices

Next, we extract the CAGE counts for the dermal fibroblast and embryonic stem cells samples. Similar as before, `process_counts.pl` extracts the counts for *all* the samples and the counts for the samples of interest are then selected using `grep`.

```
./process_counts.pl hg19.cage_peak_phase1and2combined_counts_ann.osc.txt
  > out_counts.txt

grep -e 'Fibroblast%20-%20Dermal' -e 'H9%20Embryonic%20Stem%20cells'
  out_counts.txt | awk -F'[.\t]' '{print $3"\t"$5"\t"$6}' > raw_counts.tab
```

The perl script `process_counts.pl` is presented below. We first skip through comment lines (`^#`) and description lines (`^0X`). We then check if the counts are assigned to a valid entrez gene ID. Finally, we sum up all the counts belonging to a particular gene using the perl hash `%count_data`. As we are interested in differential expression at the gene level, the counts across all the promoters of the same gene are summed, i.e., we do not make a distinction between the promoter sites.

```
#!/usr/bin/perl -w
use strict;
use warnings;

my $filename = $ARGV[0];
my @sample_names;
my @mapped_reads;
my @norm_factors;
my %count_data;
my $error_count = 0;
open(FILE,"$filename");
while(<FILE>){
    chomp;
    my $line = $_;
    unless($line =~ /^#/){
        if($line =~ /^00Annotation/){
            @sample_names = split(/\t/,$line);
        }elsif($line =~ /^01STAT:MAPPED/){
            @mapped_reads = split(/\t/,$line);
        }elsif($line =~ /^02STAT:NORM_FACTOR/){
            @norm_factors = split(/\t/,$line);
        }else{
            my @line = split(/\t/,$line);
            print Dumper $line if $off;
            my $entrez = $line[4];

            unless($entrez eq 'NA'){
                unless($entrez eq ''){
                    my @entrez = split(/:/,$entrez);
                    $entrez = $entrez[1];
                    @entrez = split(/,/,$entrez);
                    $entrez = $entrez[0];
                    if(scalar(@entrez) > 1){
                        $error_count++;
                    }
                    my $length = scalar(@line);
                    my $index = 7;
                    while($index < $length){
                        my $sample = $sample_names[$index];
                        my $count = $line[$index];
                        if(exists($count_data{$sample}{$entrez})){
$count_data{$sample}{$entrez} = $count_data{$sample}{$entrez} + $count;
                        }else{
$count_data{$sample}{$entrez} = $count;
                        }
                        $index++;
                    }
                }
            }
        }
    }
}
```

```
foreach my $sample (sort keys %count_data){
    foreach my $entrez (sort keys %{$count_data{$sample}}){
        print $sample."\t".$entrez."\t".$count_data{$sample}{$entrez}."\n";
    }
}
```

In the general case, one would like to extract the CAGE counts for all the samples and select certain samples based on some criterion. However, for the purposes of this guide, we only extract the relevant count data to reduce the memory footprint in compiling this code.

2 Quality Control

Before performing DESeq2, it is important to ensure that there are no anomalous samples ("outliers") present in the dataset which can affect downstream analysis. Thus we look at some of the summary statistics for each sample to assess their quality.

2.1 Counts Normalization

The CAGE counts have to be normalized to bring the values to a common scale for comparison. We first load both the sample details and raw CAGE counts into R and add additional information about the samples.

```r
currentDir = system("pwd", intern = TRUE)
currentDir = paste(currentDir, "", sep='/')
sampleDetails = read.table(paste(currentDir, "sample_details.tab", sep=''),
                           header=FALSE)
rawCounts = read.table(paste(currentDir, "raw_counts.tab", sep=''),
                       header=FALSE)
colnames(sampleDetails) = c("RIKENid", "description")
colnames(rawCounts) = c("RIKENid", "gene", "counts")

sampleDetails[grep("Dermal", sampleDetails$description), "category"] = "fib"
sampleDetails[grep("Embryo", sampleDetails$description), "category"] = "esc"
sampleDetails$replicate = c(seq(6), seq(3))
sampleDetails$cat_rep = paste(sampleDetails$category,
                              sampleDetails$replicate, sep = '_')
head(sampleDetails)
```

```
##     RIKENid                  description category replicate cat_rep
## 1 CNhs12499 Fibroblast_-_Dermal__donor1      fib         1   fib_1
## 2 CNhs11379 Fibroblast_-_Dermal__donor2      fib         2   fib_2
## 3 CNhs12028 Fibroblast_-_Dermal__donor3      fib         3   fib_3
## 4 CNhs12052 Fibroblast_-_Dermal__donor4      fib         4   fib_4
## 5 CNhs12055 Fibroblast_-_Dermal__donor5      fib         5   fib_5
## 6 CNhs12059 Fibroblast_-_Dermal__donor6      fib         6   fib_6
```

```r
head(rawCounts)
```

```
##     RIKENid      gene counts
## 1 CNhs12499         1     96
## 2 CNhs12499        10      0
## 3 CNhs12499       100     22
## 4 CNhs12499      1000     10
## 5 CNhs12499     10000     21
## 6 CNhs12499 100009676     20
```

We then proceed to normalize the CAGE counts for library size using one of the commands in the DESeq2 pipeline. The library size for each sample is estimated by the size factors. The counts in each sample are then divided by the corresponding size factors to obtain the normalized counts. In this case, we notice that the first sample `CNhs11379` has a low size factor, indicating that the sample has relatively lower counts across all genes.

Extra Thoughts: One can run the DESeq pipeline using the simple command: `dds = DESeq(dds)` which would also estimate the size factors but execute other commands that are unnecessary at this stage. For now, it should be pointed out that the counts normalization does not depend on the design matrix, hence `design ~1` (no design). However, the number of samples or genes does affect normalization. Normalization attempts to make the counts between samples comparable and is thus sensitive to the number of samples. The number of genes included changes the library size and the normalized counts. The various commands executed in the DESeq pipeline will be covered in greater detail in the "Running DESeq2" section.

Coding: If the `dcast` function gives an error, it is possble that there are duplicate RIKEN IDs present in the count data.

```r
library(reshape2)
library(DESeq2)
```

```r
rawCounts = dcast(rawCounts,
                  gene ~ RIKENid,
                  value.var = "counts")
rownames(rawCounts) = rawCounts$gene
rawCounts = rawCounts[,-1]

grouping = sampleDetails[match(colnames(rawCounts), sampleDetails$RIKENid),
                         "category"]
batching = sampleDetails[match(colnames(rawCounts), sampleDetails$RIKENid),
                         "replicate"]
group = factor(grouping)
batch = factor(batching)

dds = DESeqDataSetFromMatrix(as.matrix(rawCounts),
                             DataFrame(group, batch),
                             design=~1)
dds = estimateSizeFactors(dds)
normCounts = counts(dds, normalized=TRUE)
sizeFactors(dds)
```

```
## CNhs11379 CNhs11917 CNhs12028 CNhs12052 CNhs12055 CNhs12059 CNhs12499
## 0.2942283 1.3992791 1.1115833 1.5650177 1.1065379 1.2781380 0.5385380
## CNhs12824 CNhs12837
## 1.5590878 1.4237669
```

2.2 Density Plots and Boxplots

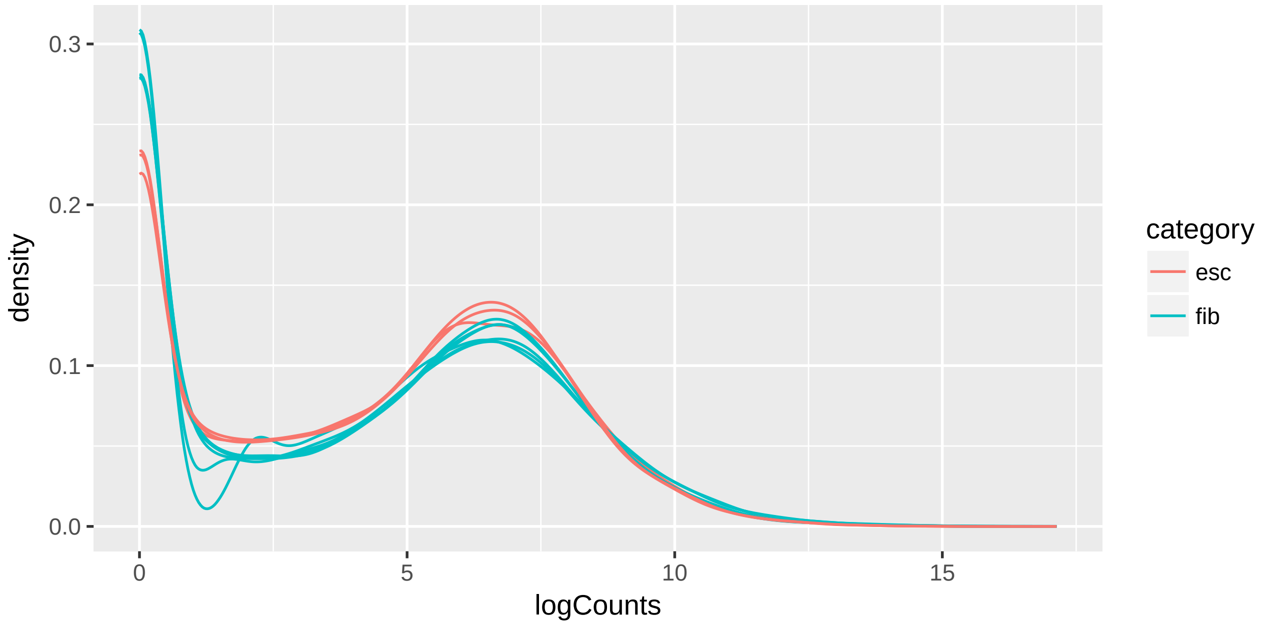
We then log-transform the normalized counts (to make gene expression ratio symmetrical) and perform density plots to visualize the distribution of gene expression in each sample.

Extra Thoughts: Typically, the density plots show two peaks, one centred at a large value and another centred around zero. The width of the former peak predominantly represents the biological variation in gene expression (gene dispersion) present in highly expressed genes. In contrast, the width of the latter peak primarily comprises uncertainties in measuring concentration via counts (shot noise or Poisson noise).

Coding: It is important to use the `match` function when assigning the category of the normalized counts. This is because the ordering of the samples may be scrambled during the `melt` function which converts the normalized counts from wide to long format. The format conversion is necessary for `ggplot` to visualize the data.

```
library(ggplot2)
```

```
normCounts = log2(normCounts+1)
normCounts_long = melt(normCounts,
                       varnames = c("GeneID", "RIKENid"),
                       value.name = "logCounts")
normCounts_long$category = sampleDetails[match(normCounts_long$RIKENid,
                                               sampleDetails$RIKENid),
                                         "category"]
ggden = ggplot(normCounts_long, aes(x=logCounts)) +
          geom_line(stat="density", aes(group=RIKENid, colour=category))
show(ggden)
```

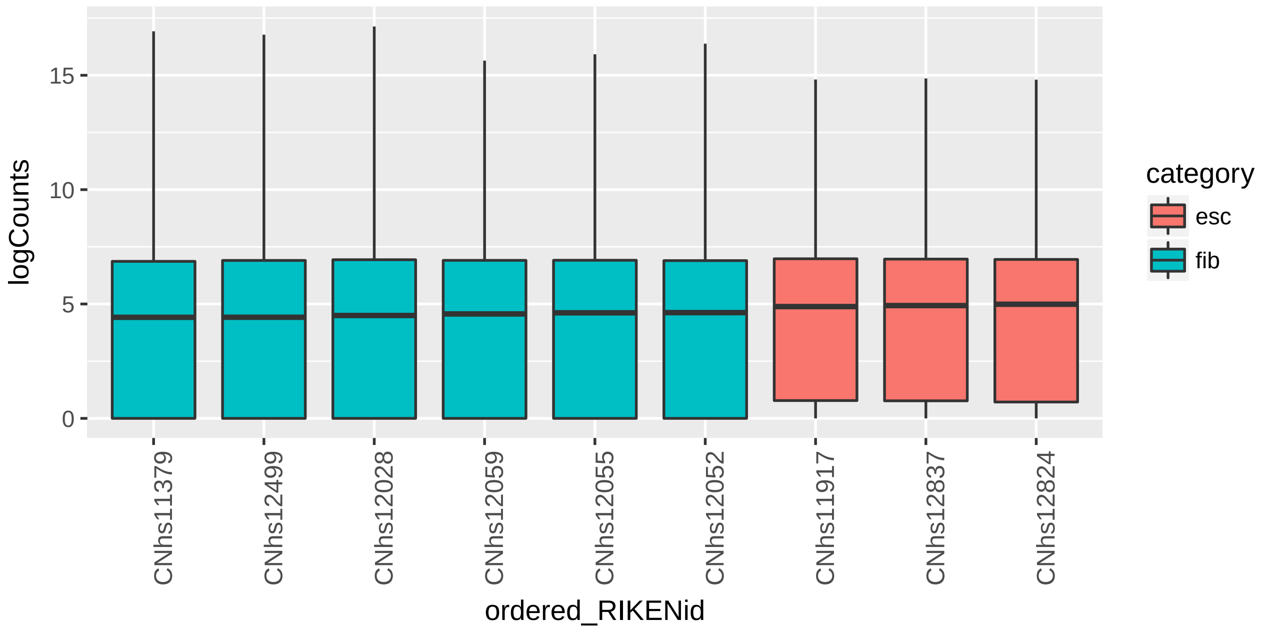


When there are too many samples, the peaks in the density plots may be obscured. In that case, it may be useful to perform box plots for each of the sample, which summarizes the distribution of log-transformed normalized counts. The purpose of the boxplots is to visually identify samples with zero median counts which are indicative of poor sequencing quality. There are none of such samples in this example.

Extra Thoughts: The box plots assumes normality but we actually assume that the normalized counts follow a negative binomial distribution. Thus, the outliers counts depicted in the box plots (interestingly, there are none in this example) may not truly be outliers.

Coding: Here, we reorder the samples by the median so that samples with zero median counts can be easily identified.

```r
ordered_RIKENid = with(normCounts_long, reorder(RIKENid, logCounts, median))
ggbox <- ggplot(normCounts_long, aes(x=ordered_RIKENid, y=logCounts,
                                               fill=category)) +
            geom_boxplot(outlier.colour = "red", outlier.shape = 1) +
            theme(axis.text.x = element_text(angle=90, hjust=0, size=10))
show(ggbox)
```



```r
# find samples (columns) with zero median
# zero_median = which(apply(normCounts, 2, FUN=median) == 0)
```

2.3 Clustering of Correlation

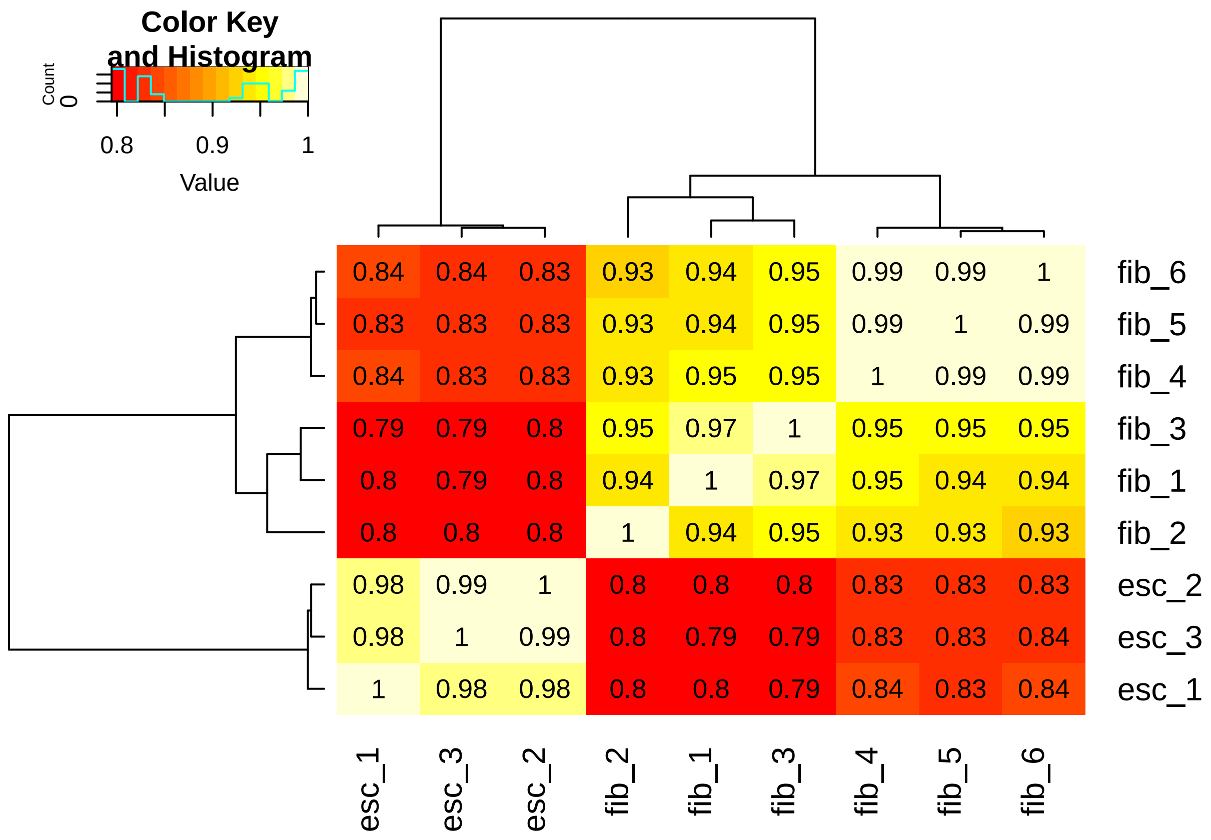
Apart from visualizing the distribution of gene expression *within each sample*, we can also look at the correlation of gene expression *between samples*. To this end, we compute the Pearson correlation of gene expression between samples and cluster the correlation. Ideally, replicates of the same cell type (or experimental condition) should cluster together. This is because we expect the variation between biological replicates to be (much) less than the true biological variation between different cell types (or experimental conditions). Here we see that the biological replicates do cluster together.

Extra Thoughts: Clustering can also be performed on the gene expression directly and genes which have similar expression patterns will be also clustered (on the y-axis). However, since we are only interested in the relationships between samples, clustering the correlation is computationally more efficient. One can also perform the clustering with Spearman correlation.

Coding: In the `heatmap.2` function, the hierachiral clustering is performed using complete linkage method which favours spherical clusters. Other linkage methods such as ward can be applied. The choice of linkage methods for hierachiral clustering is an art in itself and will not be discussed here.

```
library(gplots)
```

```
cor_pearson = cor(normCounts)
colnames(cor_pearson) = sampleDetails[match(colnames(cor_pearson),
                             sampleDetails$RIKENid), "cat_rep"]
rownames(cor_pearson) = sampleDetails[match(rownames(cor_pearson),
                             sampleDetails$RIKENid), "cat_rep"]
heatmap.2(cor_pearson, trace='none',
          cellnote = round(cor_pearson, digits=2), notecol = "black")
```



2.4 PCA Plots

Alternatively, we can also perform Principal Component Analysis (PCA) on the gene expression. Again, biological replicates should be close to each other on the PCA plots. In addition, PCA plots allow us to visually identify batch effects which is harder to distinguish from clustering alone. In this example, the embryonic stem cell samples are well clustered together while the dermal fibroblast samples are "somewhat" apart on the second PC. However, on closer inspection, only 9% of the variance is recovered by PC2 which is rather negligible.

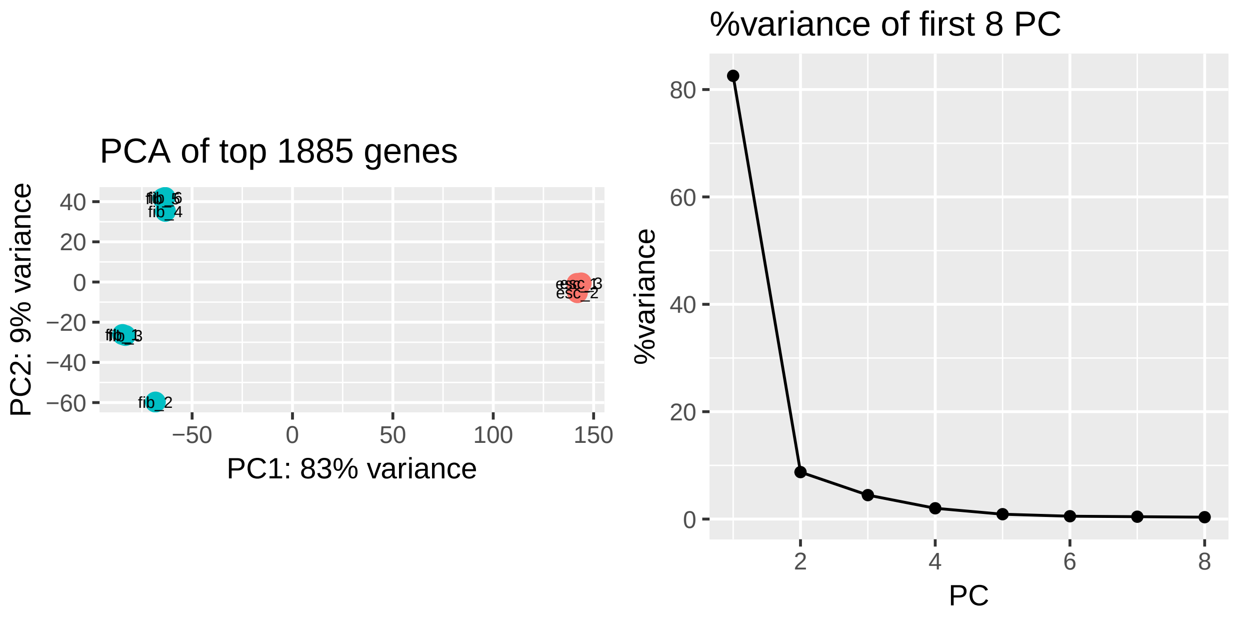
Extra Thoughts: In PCA, we want to visualize the differences in gene expression between samples. Therefore, we would only want to include genes that are variationally expressed, i.e., genes with a high variance across samples. In this case, we chose to include the top 10% of variationally expressed genes.

Coding: DESeq2 has a built-in `plotPCA` function where you can specify the `nTop` variationally expressed genes to be included. However, `plotPCA` can only plot the first two PCs and the variance captured by the PCs are not shown as well.

```r
library(genefilter)
library(gridExtra)
```

```r
# select top n most variationally expressed genes
ntop = round(0.1*dim(normCounts)[1])
rv = rowVars(normCounts)
select = order(rv, decreasing = TRUE)[seq_len(min(ntop,length(rv)))]

pca = prcomp(t(normCounts[select, ]))
percentVar = pca$sdev^2/sum(pca$sdev^2)

# plot PC1 and PC2
ggpca = data.frame(PC1 = pca$x[, 1], PC2 = pca$x[, 2], PC3 = pca$x[, 3],
                   group = group, cat_rep = sampleDetails[match(
                  colnames(normCounts), sampleDetails$RIKENid), "cat_rep"])
plot12 = ggplot() +
  geom_point(data = ggpca, aes_string(x="PC1", y="PC2", color="group"),
             size = 3) +
  geom_text(data = ggpca, aes_string(x="PC1", y="PC2", label="cat_rep"),
            color = "black", cex = 2) +
  ggtitle(paste('PCA of top ', ntop, ' genes', sep='')) +
  xlab(paste0("PC1: ", round(percentVar[1] * 100), "% variance")) +
  ylab(paste0("PC2: ", round(percentVar[2] * 100), "% variance")) +
  coord_fixed() + theme(legend.position="none")

# plot the variance captured by PCs
percentVar8 = percentVar[seq(8)]*100
percentVar8 = as.data.frame(percentVar8)
```

```
percentVar8$PC = seq(8)
plotVar = ggplot(data=percentVar8, aes_string(x="PC", y="percentVar8")) +
  geom_point() + geom_line() + ylab("%variance") +
  ggtitle('%variance of first 8 PC')

grid.arrange(plot12, plotVar, ncol=2, respect=TRUE)
```



2.5 Summary of QC Methods

Here, we summarize the various methods to perform quality control checks, the information captured by these methods and the problems that can be uncovered.

Method	Information	Quality Control
Density Plots / Boxplots	Within sample	Samples with zero-median counts
Clustering of Correlation	Between sample	Poor-quality biological replicates
PCA Plots	Between sample	Poor replicates / Batch effects

3 Running DESeq2

After quality control checks, we are now in a position to find differentially expressed genes using DESeq2.

3.1 Quick Start on DESeq2

Running DESeq2 can be split into three main steps: (i) Preparation of data into a DESeq-friendly "DESeqDataSet" format, (ii) Executing the DESeq2 pipeline and (iii) Extracting and analyzing the results from the pipeline.

For the first step, we need to create a "DESeqDataSet" which comprises three pieces of data: count data, genomic ranges data and sample/experimental information. The counts and genomic ranges data are contained in the `rawCounts` matrix. In our case, the genomic ranges correspond to the entrez gene IDs. Information on the samples are contained in `DataFrame(group)`, for example, cell type and replicate ID. Finally, we need to specify the design matrix which tells DESeq2 the conditions for which we wish to test for differential expression. In our case, we want to find differentially expressed genes between dermal fibroblast and embryonic stem cells. Hence, we specify the cell type as the design, denoted by `group`. The abovementioned variables have been generated in the section "Counts Normalization".

Coding: It is important to provide *integer* and un-normalized count matrices for DESeq2's statistical model to hold. Thus, DESeq will throw up an error if real number counts are provided. The statistical model in DESeq2 will be covered in greater detail in the next section "Brief Description of the Theory behind DESeq2".

```
dds = DESeqDataSetFromMatrix(as.matrix(rawCounts),
                             DataFrame(group), design=~group)
```

For the second step, the standard differential analysis steps are wrapped into a single function `DESeq`. By default, the Wald test is performed which compares whether the log2 fold change of gene expression between two sets of samples are different. Alternatively, one can run the liklihood ratio test (LRT) for comparing differential gene expression between a full and reduced model (see "Wald Test vs. Likelihood Ratio Test" for more details). The various analysis steps will be covered in "Brief Description of the Theory behind DESeq2".

```
dds = DESeq(dds)
```

For the third step, we generate the results table from DESeq2 analysis. The results table contains the log2 fold change (LFC) between the dermal fibroblast and embryonic stem cells and the associated standard error. More importantly, the values of interest are the adjusted p-value `padj`. Note that in some cases we get `NA` values which apply to genes with outlier counts or genes with low counts. In this case, both gene `100653515` and `100750325` have low counts, indicated by the mean of the counts across all samples `baseMean`. A summary of the results can also be extracted, indicating the number of differentially expressed genes (split into increase and decreased expression).

```
res = results(dds)
res[,c(1:3,6)]
```

```
## log2 fold change (MAP): group fib vs esc
##
## DataFrame with 18852 rows and 4 columns
##             baseMean log2FoldChange     lfcSE        padj
##            <numeric>      <numeric> <numeric>   <numeric>
## 1         96.6886320      1.6775597 0.4754466 0.001321546
## 2          5.2917883      3.8457086 1.5491993 0.028627730
## 9         15.8140321     -0.8812698 0.4742817 0.111474903
## 10         0.6108301     -1.9001322 1.9152647 0.427795326
## 12         0.1779978      0.1588146 1.9310956          NA
## ...              ...            ...       ...         ...
## 100653515 0.00000000             NA        NA          NA
## 100750247 0.09995752      0.6843919  1.931096          NA
## 100750325 0.00000000             NA        NA          NA
## 100820829 0.07804024     -0.1039746  1.931096          NA
## 100859930 0.36934065     -0.5204751  1.881538   0.8375937
```

```
summary(res)
```

```
##
## out of 16219 with nonzero total read count
## adjusted p-value < 0.1
## LFC > 0 (up)     : 3823, 24%
## LFC < 0 (down)   : 4611, 28%
## outliers [1]     : 95, 0.59%
## low counts [2]   : 933, 5.8%
## (mean count < 0)
## [1] see 'cooksCutoff' argument of ?results
## [2] see 'independentFiltering' argument of ?results
```

Coding: Here, we can also introduce an additional parameter `alpha` which specify the adjusted p-value cutoff for statistical significance. Interestingly, changing `alpha` affects the threshold of an independent filtering procedure which filters low counts. Thus, the number of genes which are differentially expressed will be different.

```
res05 = results(dds, alpha=0.05)
sum(res$padj < 0.05, na.rm=TRUE) == sum(res05$padj < 0.05, na.rm=TRUE)
```

```
## [1] FALSE
```

Furthemore, we can order our results table by the smallest adjusted p-value and subset a list of genes which are deemed to be differentially expressed at our given adjusted p-value cutoff.

```r
resOrdered <- res[order(res$padj),]
head(resOrdered[,c(1:3,6)])
```

```
## log2 fold change (MAP): group fib vs esc
##
## DataFrame with 6 rows and 4 columns
##        baseMean log2FoldChange     lfcSE          padj
##       <numeric>      <numeric> <numeric>     <numeric>
## 857   5361.4302       8.208967 0.2472871 1.861682e-237
## 54596 1408.4919      -9.802369 0.3983783 8.341756e-130
## 79026 1653.5466       6.154834 0.2554555 1.478400e-124
## 11040  871.5982      -6.333103 0.2788533 1.317608e-110
## 5054  7129.2725       8.910413 0.3944704 1.716323e-109
## 6277  6891.5609       9.504375 0.4414586  2.087548e-99
```

```r
resSig <- subset(resOrdered, padj < 0.05)
dim(resSig)
```

```
## [1] 7501    6
```

3.2 Brief Description of the Theory behind DESeq2

Thus far, we have been using DESeq2 in a black box manner. To better understand how changing the various inputs and parameters can affect the results, we need some basic understanding of the theory behind DESeq2. DESeq2 assumes that the counts K_{ij} for gene i and sample j follows a negative binomial distribution:

$$K_{ij} \sim NB(\mu_{ij}, \alpha_i)$$
$$\mu_{ij} = s_j q_{ij}$$
$$\log_2(q_{ij}) = x_j \beta_i$$

where the mean of the distribution μ_{ij} depends on the size factor s_j for each sample and a parameter q_{ij} fitted using a generalized linear model of the log2 fold change β_i. The estimated gene-specific dispersion parameter α_i defines the relationship between the variance and mean of the observed counts. The variance of the negative binomial distribution is given as:

$$\text{Var}(K_{ij}) = \mu_{ij} + \alpha_i \mu_{ij}^2$$

Extra Thoughts: A burning question at this point would be: "Why the negative binomial distribution?" As we are using count data, we are restricted to discrete distributions, for example, the binomial, negative binomial and poisson distribution. The poisson distribution is overly restrictive as a model given that the variance is equal to the mean. The same is true for the binomial counterpart where the variance $\leq$ the mean. In reality, we expect that it is possible for the variance of gene expression to exceed its mean value. This leaves us with the negative binomial distribution. Furthermore, the negative binomial distribution can be seen as an overdispersed poisson model. This is because the variance has two components, namely the poisson component in the first term and the overdispersion component in the second term. The poisson component can be loosely treated as the measurement uncertainties while the overdispersion component describes the biological variation in gene expression. This also brings us to reiterate that it is important to provide integer count data as the input to DESeq for the validity of using a negative binomial distribution model.

We can now relate the theory to the steps performed by the `DESeq` function. Briefly, `DESeq` does the following:

- `estimateSizeFactors` which estimates the size factors s_j
- `estimateDispersions` which estimates the dispersion α_i
- `nbinomWaldTest` which performs the negative binomial GLM to obtain q_{ij} and the Wald statistics to compare the log2 fold change between two groups of samples

Thus, the following codes (which are not evaluated here) are equivalent:

```r
dds = DESeq(dds)

# is equivalent to:

dds = estimateSizeFactors(dds)
dds = estimateDispersions(dds)
dds = nbinomWaldTest(dds)
```

Extra Thoughts: Previously, we mentioned that the size factors does not depend on the design matrix. However, the dispersion estimates and Wald test do depend on the design matrix. The dispersions are estimated on a per group basis as dictated by the design matrix. For the case of the Wald test, the design matrix determines the groups to be compared in computing the log2 fold change.

We will discuss more on how the Wald statistical test works in the next section.

3.3 Wald Test vs. Likelihood Ratio Test

Apart from the default Wald test, differential gene expression can also be tested using the likelihood ratio test (LRT). In the Wald test, the shrunken estimate of the log2 fold change β_i is divided by its standard error to give a z-statistic which is then compared to a normal distribution to give a p-value. Here the shrinkage in the log2 fold change serves to correct for lowly-expressed genes suffering from a small signal-to-noise ratio. It should be noted that Wald test only allows for pairwise comparisons, evident from the use of the log2 fold change.

In contrast, the LRT compares two models for the counts: a full model with some terms in the design matrix and a reduced model for which some of the terms are removed. The test determines if there is a significant change in deviance between the full and reduced model, conceptually similar to an analysis of variance (ANOVA) calculation in linear regression. Consequently, the LRT allows for testing multiple terms and groups at once. The LRT is useful in analyzing timeseries experiment where the time factor can be omitted in the reduced model (see example code below, which is not evaluated). Note that in LRT, there is no shrinkage in the log2 fold change estimates and thus Wald test is preferred.

Coding: The LRT p-values obtained from `results(dds)` represents a test of all the variables and level of factors amongst these variables. Thus, the p-values are outputted regardless of the pairs of groups selected by `contrast`. The only difference between pairs of groups is the log2 fold change.

```
dds = DESeqDataSetFromMatrix(as.matrix(rawCounts),
                             DataFrame(group), design=~group)
dds = DESeq(dds, test="LRT", reduced=~1)

# is equivalent to:

dds = estimateSizeFactors(dds)
dds = estimateDispersions(dds)
dds = nbinomLRT(dds, full=~group, reduced=~1)
```

Extra Thoughts: One might wonder whether the Wald test and LRT will yield very different results when there are two groups/conditions to be compared. Here, we test it out with our dataset. The results are likely to be similar with differences attributed to the lack of shrinkage in the log2 fold change estimates in LRT.

```
ddsWald = DESeq(dds, quiet = TRUE)
ddsLRT  = DESeq(dds, quiet = TRUE, test="LRT", reduced=~1)

resWald = results(ddsWald)
resLRT  = results(ddsLRT)

sum(resWald$padj < 0.05, na.rm=TRUE)
```

```
## [1] 7501
```

```
sum(resLRT$padj < 0.05, na.rm=TRUE)
```

```
## [1] 7466
```

3.4 Multi-factor Design and Batch Effects

Samples with more than one factor influencing the count data can be analyzed by including the different factors in the design matrix. For example, the gene expression may be affected by batch effects and biological differences between different cell types (in this case dermal fibroblast and embryonic stem cells). The two factors are easily specified by `design=~batch+group`. The factor of interest, in this case `group`, should be placed at the end of the formula. This allows for the factor of interest to be compared in the default Wald test. Interaction terms between factors can also be specified, for example `design=~batch:group`, but they will not be covered in great detail here.

```
ddsGroup = DESeqDataSetFromMatrix(as.matrix(rawCounts),
                                  DataFrame(batch, group), design=~batch+group)
ddsGroup = DESeq(ddsGroup, quiet=TRUE)
resGroup = results(ddsGroup)
resGroup[,c(1:3,6)]
```

```
## log2 fold change (MAP): group fib vs esc
##
## DataFrame with 18852 rows and 4 columns
##             baseMean log2FoldChange     lfcSE         padj
##            <numeric>      <numeric> <numeric>    <numeric>
## 1         96.6886320      2.1451049 0.3661882 4.770343e-08
## 2          5.2917883      3.5256122 1.5984623 5.957388e-02
## 9         15.8140321     -0.6991929 0.7519776 4.654006e-01
## 10         0.6108301     -1.7519184 1.8546447           NA
## 12         0.1779978      0.1731009 1.8606698           NA
## ...              ...            ...       ...          ...
## 100653515 0.00000000             NA        NA           NA
## 100750247 0.09995752     0.67940698  1.860670           NA
## 100750325 0.00000000             NA        NA           NA
## 100820829 0.07804024    -0.08892905  1.860670           NA
## 100859930 0.36934065    -0.46564772  1.857156           NA
```

Extra Thoughts: The curious ones will ask: "What if we reverse the ordering of design matrix to `design=~group+batch`?" Here we tried it out. It appears that the default settings in `results` compares a pair of variables belonging to the last factor in the design matrix. Thus, when we set `design=~group+batch`, the results compare a pair of variables in `batch`, in this case, comparing batch 1 and 6 (for which the results are rather nonsensical).

```
ddsBatch = DESeqDataSetFromMatrix(as.matrix(rawCounts),
                                  DataFrame(group, batch), design=~group+batch)
ddsBatch = DESeq(ddsBatch, quiet=TRUE)
resBatch = results(ddsBatch)
resBatch[,c(1:3,6)]
```

```
## log2 fold change (MAP): batch 6 vs 1
##
## DataFrame with 18852 rows and 4 columns
##             baseMean log2FoldChange     lfcSE       padj
##            <numeric>      <numeric> <numeric>  <numeric>
## 1         96.6886320     -1.3962221 0.4976686 0.09623081
## 2           5.2917883     -0.5686795 0.4538855         NA
## 9          15.8140321     -0.1539061 0.6628667         NA
## 10          0.6108301     -0.1663843 0.2872503         NA
## 12          0.1779978      0.0000000 0.2836011         NA
## ...               ...            ...       ...        ...
## 100653515 0.00000000             NA        NA         NA
## 100750247 0.09995752              0 0.2836011         NA
## 100750325 0.00000000             NA        NA         NA
## 100820829 0.07804024              0 0.2836011         NA
## 100859930 0.36934065              0 0.2855678         NA
```

Extra Thoughts: Fortunately, we can still retrieve the comparison between variables in `group`, i.e, comparing gene expression between dermal fibroblast `fib` and embryonic stem cells `esc`. This is achieved by specifying the variables to be compared in the `contrast` function. Here, we managed to reproduce the results where the `group` is compared. The list of factors and variables can be obtained by running the `resultsNames` function, for example, `resultsNames(ddsBatch)`.

```
resNew = results(ddsBatch, contrast=c("group","fib","esc"))
resNew[,c(1:3,6)]
```

```
## log2 fold change (MAP): group fib vs esc
##
## DataFrame with 18852 rows and 4 columns
##             baseMean log2FoldChange     lfcSE         padj
##            <numeric>      <numeric> <numeric>    <numeric>
## 1         96.6886320      2.1451049 0.3661882 4.770343e-08
## 2          5.2917883      3.5256122 1.5984623 5.957388e-02
## 9         15.8140321     -0.6991929 0.7519776 4.654006e-01
## 10         0.6108301     -1.7519184 1.8546447           NA
## 12         0.1779978      0.1731009 1.8606698           NA
## ...              ...            ...       ...          ...
## 100653515 0.00000000             NA        NA           NA
## 100750247 0.09995752     0.67940698  1.860670           NA
## 100750325 0.00000000             NA        NA           NA
## 100820829 0.07804024    -0.08892905  1.860670           NA
## 100859930 0.36934065    -0.46564772  1.857156           NA
```

In the design matrix of a `DESeqDataSet`, the batch information (or replicate ID) can be included to describe any batch effects that may be present in the samples. It is important to realize that DESeq2 only *models* the batch effect in performing the statistical tests. The normalized or transformed counts obtained from DESeq2 still contain the batch effects. In order to remove the batch effects from the counts, one should use other packages, for example, `limma` package's `removeBatchEffect` function.

3.5 Computational Finetuning

There are several ways in which we can accelerate the DESeq pipeline computation.

The `DESeq` and `results` functions both contain an argument `parallel` which can be set to `TRUE` to enable parallel computation across multiple cores.

```
dds = DESeq(dds, parallel = TRUE)
```

Furthermore, genes with low counts across all samples can be removed before running DESeq2. While this is not necessary, the removal of these genes lowers the memory consumption and increases the speed of transformation and testing functions in the DESeq2 pipeline. An example is provided here where genes with 0 or 1 count are removed.

```
dds <- dds[ rowSums(counts(dds)) > 1, ]
```