Report 2

Danny Kearns

6/23/2020

This report will show three different ways to analyse a group of biological data and gene expression. I will demonstrate how to do this in `DESeq2`, using a random forest, and with `tidylo`, the weighted log odds ratio

DESeq2

The first thing to do is read in some libraries and data. I'm once agian using the `pasilla` data set

```
library(pasilla)
library(tidyverse)
library(tximeta)
library(tximportData)
library(Rsamtools)
library(DESeq2)
library(magrittr)

pasCts <- system.file("extdata",
                      "pasilla_gene_counts.tsv",
                      package="pasilla", mustWork=TRUE)
pasAnno <- system.file("extdata",
                       "pasilla_sample_annotation.csv",
                       package="pasilla", mustWork=TRUE)
cts <- as.matrix(read.csv(pasCts,sep="\t",row.names="gene_id"))
(coldata <- read.csv(pasAnno, row.names=1))
```

```
##                condition        type number.of.lanes total.number.of.reads
## treated1fb       treated single-read               5              35158667
## treated2fb       treated  paired-end               2         12242535 (x2)
## treated3fb       treated  paired-end               2         12443664 (x2)
## untreated1fb untreated single-read               2              17812866
## untreated2fb untreated single-read               6              34284521
## untreated3fb untreated  paired-end               2         10542625 (x2)
## untreated4fb untreated  paired-end               2         12214974 (x2)
##              exon.counts
## treated1fb      15679615
## treated2fb      15620018
## treated3fb      12733865
## untreated1fb    14924838
## untreated2fb    20764558
## untreated3fb    10283129
## untreated4fb    11653031
```

```
coldata <- coldata[,c("condition","type")]
coldata$condition <- factor(coldata$condition)
coldata$type <- factor(coldata$type)
```

So, I have `cts` which is a matrix that has a list of samples and `coldata` which contains a summary of the experiment. I'm only looking at `type` and `condition`, so then I will then take then names of the treatment groups and use them as my features in my count matrix. The following manipulation is being done to prepare the data for the next step.

```
rownames(coldata) = sub("fb", "", rownames(coldata))
cts = cts[,rownames(coldata)]
```

Using the `DESeqDataSetFromMatrix` function, I'll pass `DESeq2` my count matrix, `coldata` and a formula. Currently, I'm only going to look at how condition affects the overall gene expression, so that is what I'm making my formula in the `design` argument of the function. Going further, I only want to keep any data where the count data is GEQ 10, just to avoid unnecessary outliers

```
dds = DESeqDataSetFromMatrix(countData = cts,
                              colData = coldata,
                              design = ~ condition)
keep = rowSums(counts(dds)) >= 10
dds = dds[keep,]
```

Before going any further, I want to make sure the system understands that there is a difference between `treated` and `untreated` groups, so I'm going to relevel them.

```
dds$condition = relevel(dds$condition, ref = "untreated")
```

Now, I'm going to let `DESeq2` analyse the data. The `results` function will summarise it's findings.

```
dds = DESeq(dds)
(res = results(dds))
```

```
## log2 fold change (MLE): condition treated vs untreated
## Wald test p-value: condition treated vs untreated
## DataFrame with 9921 rows and 6 columns
##               baseMean log2FoldChange     lfcSE       stat    pvalue      padj
##              <numeric>      <numeric> <numeric>  <numeric> <numeric> <numeric>
## FBgn0000008   95.14429     0.00227644  0.223729   0.010175 0.9918817  0.997211
## FBgn0000014    1.05652    -0.49512039  2.143186  -0.231021 0.8172987        NA
## FBgn0000017 4352.55357    -0.23991894  0.126337  -1.899041 0.0575591  0.288002
## FBgn0000018  418.61048    -0.10467391  0.148489  -0.704927 0.4808558  0.826834
## FBgn0000024    6.40620     0.21084779  0.689588   0.305759 0.7597879  0.943501
## ...                ...            ...       ...        ...       ...       ...
## FBgn0261570 3208.38861      0.2955329  0.127350  2.3206264  0.020307  0.144240
## FBgn0261572    6.19719     -0.9588230  0.775315 -1.2366888  0.216203  0.607848
## FBgn0261573 2240.97951      0.0127194  0.113300  0.1122634  0.910615  0.982657
## FBgn0261574 4857.68037      0.0153924  0.192567  0.0799327  0.936291  0.988179
## FBgn0261575   10.68252      0.1635705  0.930911  0.1757102  0.860522  0.967928
```

I'm only going to focus on `baseMean`, `log2FoldChange`, and `pvalue` for the time being. The `baseMean` is the average count of genes per sample, `log2FoldChagne` takes the change in expression between a treated and untreated group, and takes the log_2 of that. `pvalue` obviously is the p-value of the hypothesis test between treated and untreated groups.

Now, one of things to notice is the `lfcSE`, the logfold change standard error. Ideally, I want to minimize standard error. There are currently a few ways to shrink the logfold change estimates. A current favorite is the `apeglm` algorithm, as developed in Zhu, Ibrahim, and Love 2018.

```
(resLFC = lfcShrink(dds, coef = "condition_treated_vs_untreated",
                    type = "apeglm"))
```

```
## log2 fold change (MAP): condition treated vs untreated
## Wald test p-value: condition treated vs untreated
## DataFrame with 9921 rows and 5 columns
##                baseMean log2FoldChange     lfcSE    pvalue      padj
##               <numeric>      <numeric> <numeric> <numeric> <numeric>
## FBgn0000008   95.14429     0.00119920  0.151897 0.9918817  0.997211
## FBgn0000014    1.05652    -0.00473412  0.205468 0.8172987        NA
## FBgn0000017 4352.55357    -0.18989990  0.120377 0.0575591  0.288002
## FBgn0000018  418.61048    -0.06995753  0.123901 0.4808558  0.826834
## FBgn0000024    6.40620     0.01752715  0.198633 0.7597879  0.943501
## ...                ...            ...       ...       ...       ...
## FBgn0261570 3208.38861     0.24110290 0.1244469  0.020307  0.144240
## FBgn0261572    6.19719    -0.06576173 0.2141351  0.216203  0.607848
## FBgn0261573 2240.97951     0.01000619 0.0993764  0.910615  0.982657
## FBgn0261574 4857.68037     0.00843552 0.1408267  0.936291  0.988179
## FBgn0261575   10.68252     0.00809101 0.2014704  0.860522  0.967928
```

Same counts and p-values, less standard error

```
resOrdered = resLFC[order(res$pvalue),]
summary(resLFC)
```

```
##
## out of 9921 with nonzero total read count
## adjusted p-value < 0.1
## LFC > 0 (up)       : 518, 5.2%
## LFC < 0 (down)     : 536, 5.4%
## outliers [1]       : 1, 0.01%
## low counts [2]     : 1539, 16%
## (mean count < 6)
## [1] see 'cooksCutoff' argument of ?results
## [2] see 'independentFiltering' argument of ?results
```

```
# How many adjusted p-values are less than 0.1?
sum(resLFC$padj < 0.1, na.rm = T)
```

```
## [1] 1054
```

I'll do the same thing with the results function, eliminating anything with a p-value less than 0.95

```
res05 = results(dds, alpha = 0.05)
summary(res05)
```

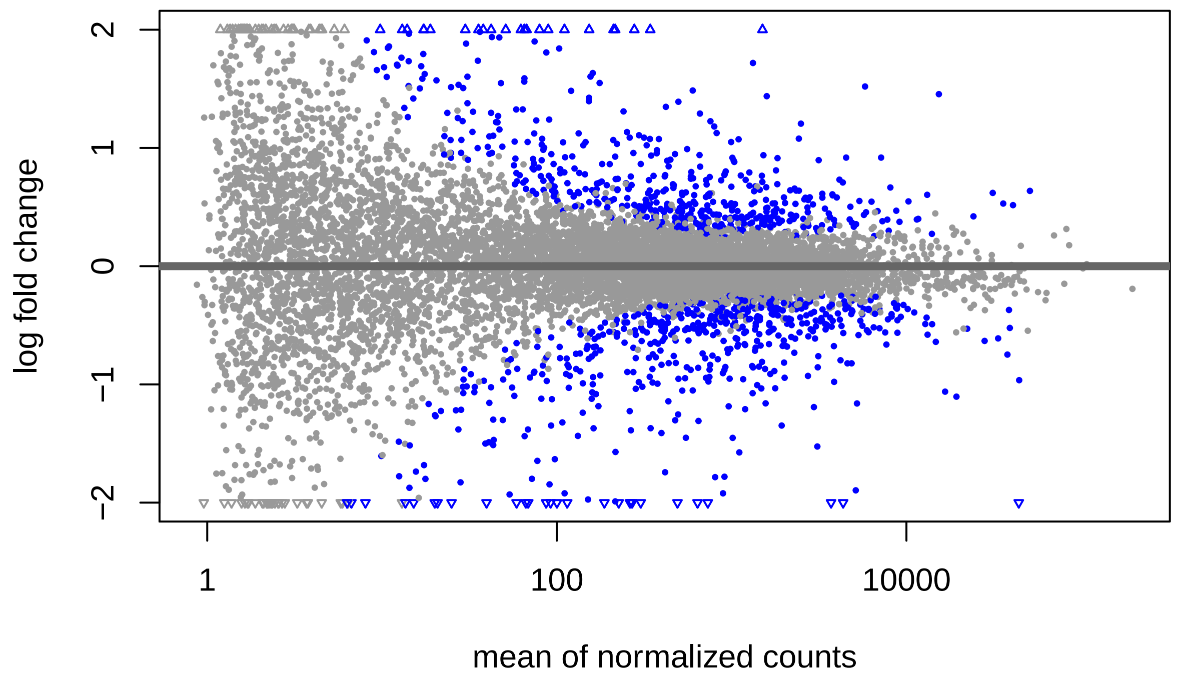
```
##
## out of 9921 with nonzero total read count
## adjusted p-value < 0.05
## LFC > 0 (up)       : 407, 4.1%
## LFC < 0 (down)     : 431, 4.3%
## outliers [1]       : 1, 0.01%
## low counts [2]     : 1347, 14%
## (mean count < 5)
## [1] see 'cooksCutoff' argument of ?results
## [2] see 'independentFiltering' argument of ?results
```

```
sum(res05$padj < 0.05, na.rm = T)
```

```
## [1] 838
```

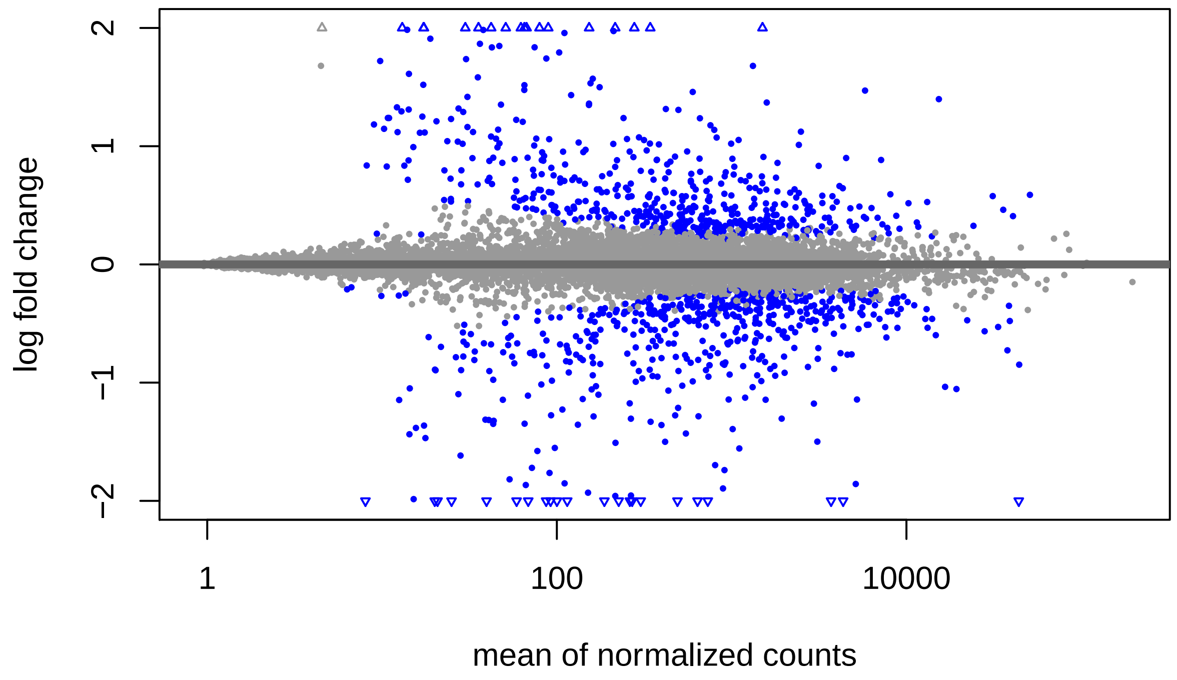
DESeq2 uses the plotMA function to plot log2 fold changes to a given variable over the mean of normalized counts for all samples in the data sets. In this case, the condition is going to be the cause of my log2 fold change. If the point is in blue, the adjusted p-value is less then 0.1. Any points that exceed the window parameters are plotted as open triangles, either pointing up or down

```r
plotMA(res, ylim = c(-2,2))
```



Using the MA plot with the shrink algorithm, the background noise from low counts goes away and it gives a better visual.

```r
plotMA(resLFC, ylim = c(-2,2))
idx = identify(res$baseMean, res$log2FoldChange)
```



```r
rownames(res)[idx]
```

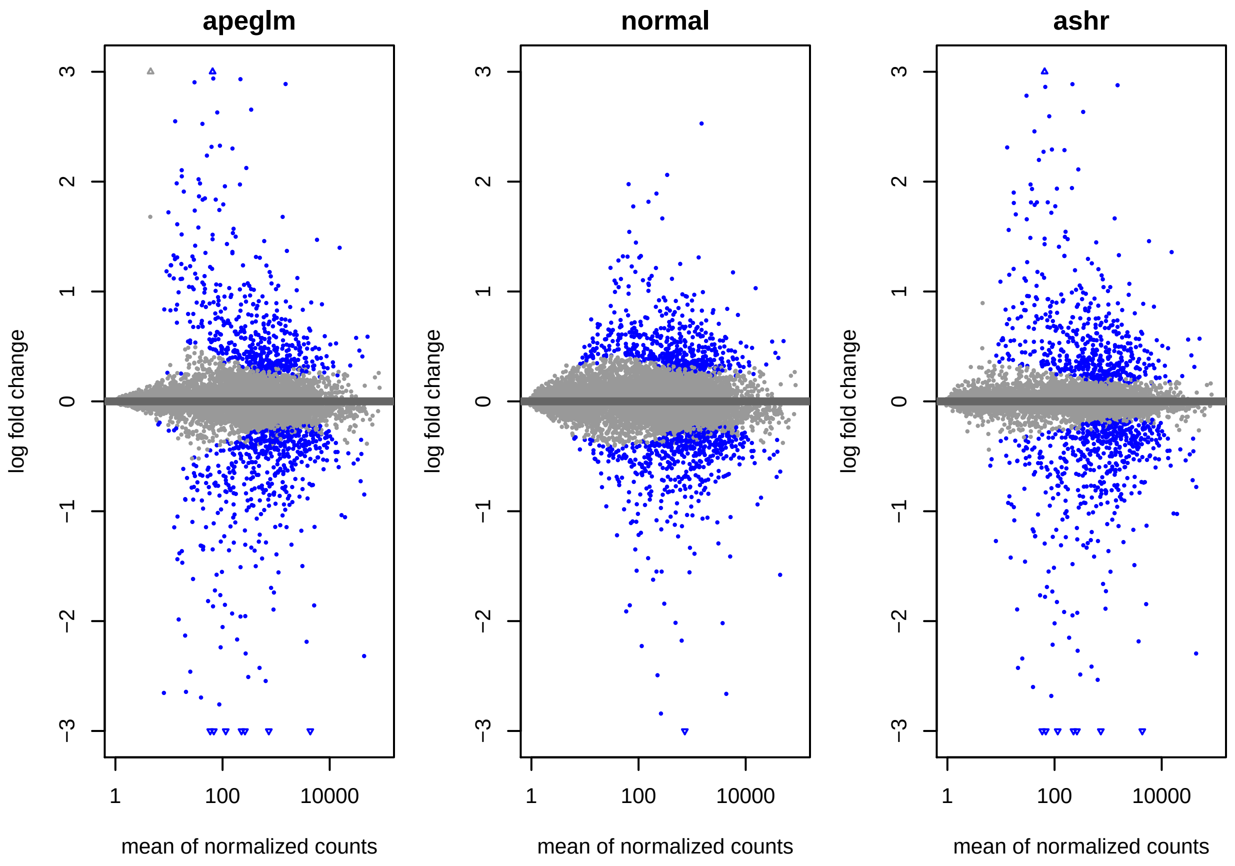
```
## character(0)
```

Of course, there are other algorithms for shrinkage to use. There's the `ashr` algorithm, discussed in Stephens, M (2016), there's also the normal shrinkage algorithm. The `normal` shrinkage algorithm can also be used

```r
resNorm = lfcShrink(dds, coef = 2, type = "normal")
resAsh = lfcShrink(dds, coef = 2, type = "ashr")
```

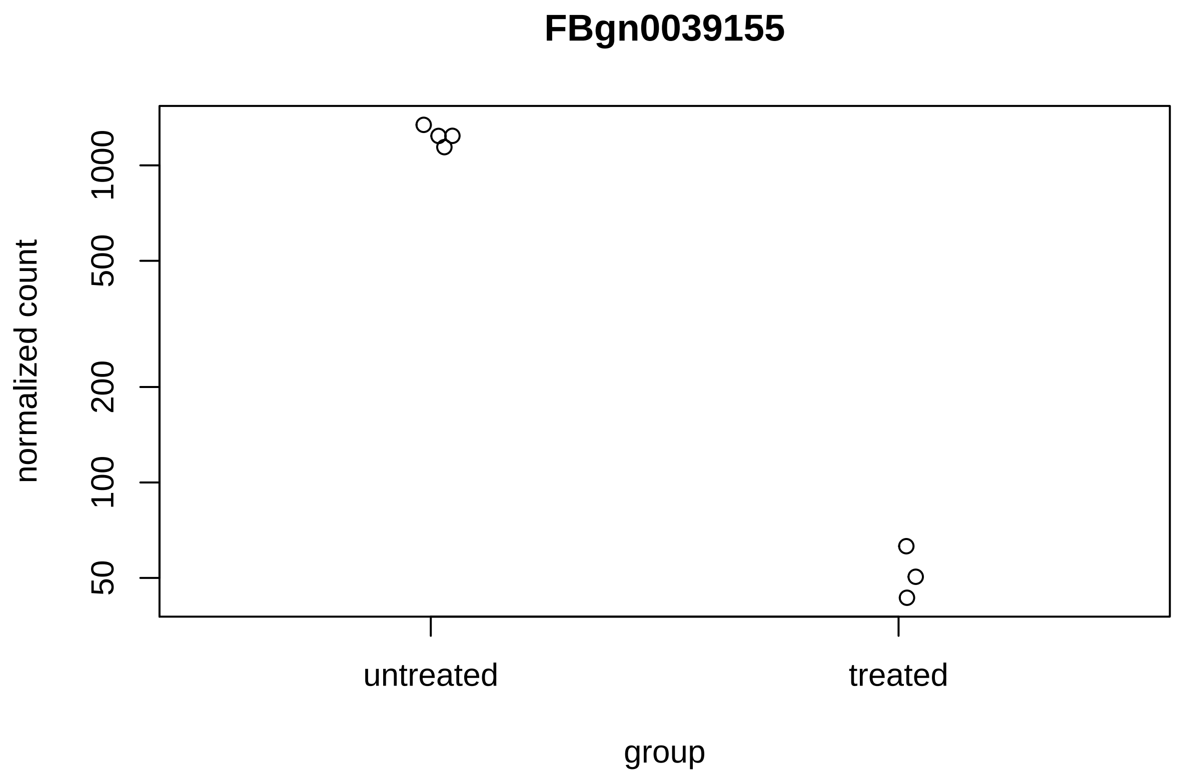
Now, to plot all of them

```r
par(mfrow = c(1,3), mar = c(4,4,2,1))
xlim = c(1,1e5); ylim = c(-3,3)
plotMA(resLFC, xlim = xlim, ylim = ylim, main = "apeglm")
plotMA(resNorm, xlim = xlim, ylim = ylim, main = "normal")
plotMA(resAsh, xlim = xlim, ylim = ylim, main = "ashr")
```



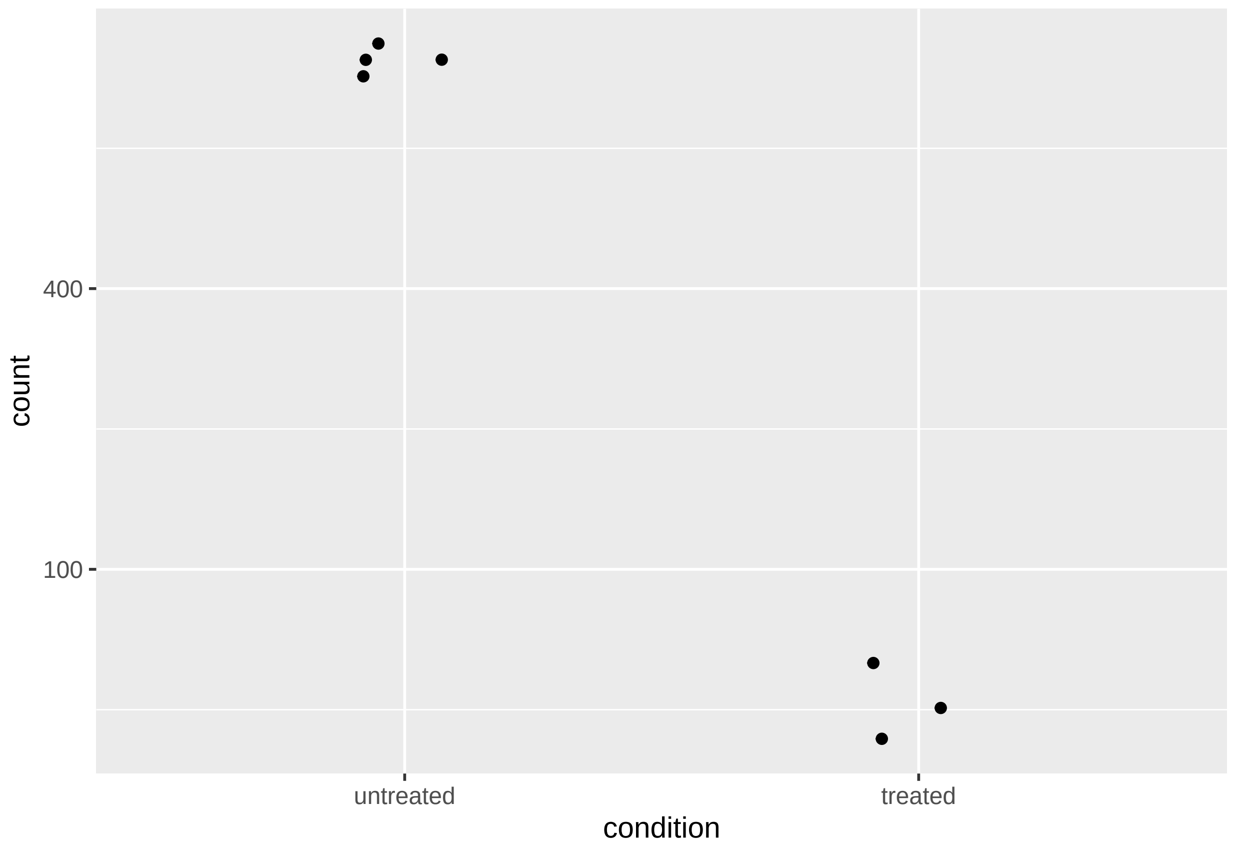
Of course, it could help to plot the counts of reads for a single gene across the groups. `plotCounts` does that. Here, I plotted the gene with the greatest changed between treated and untreated groups based on adjusted p-values.

```r
plotCounts(dds, gene = which.min(res$padj), intgroup = "condition")
```



This can also be done in ggplot2

```
d = plotCounts(dds, gene = which.min(res$padj), intgroup = "condition",
               returnData = T)
ggplot(d, aes(x = condition, y = count)) +
  geom_point(position = position_jitter(w=0.1, h = 0)) +
  scale_y_log10(breaks = c(25,100,400))
```



Multi-factor

I can use a lot of the same steps to compare multiple features. In this case, I can also include type of analysis in my comparison of samples

```
ddsMulti = dds
levels(ddsMulti$type) = sub("-.*", "", levels(ddsMulti$type))
levels(ddsMulti$type)
```

```
## [1] "paired" "single"
```

Just now, I'll change the `design` argument of the `DESeq` object to include both `condition` and `type`

```
design(ddsMulti) <- formula(~ type + condition)
ddsMulti <- DESeq(ddsMulti)
```

```
resMulti = results(ddsMulti)
head(resMulti)
```

```
## log2 fold change (MLE): condition treated vs untreated
## Wald test p-value: condition treated vs untreated
## DataFrame with 6 rows and 6 columns
##               baseMean log2FoldChange     lfcSE       stat    pvalue      padj
##              <numeric>      <numeric> <numeric>  <numeric> <numeric> <numeric>
## FBgn0000008   95.14429     -0.0405571  0.220040 -0.1843169 0.8537648  0.949444
## FBgn0000014    1.05652     -0.0835022  2.075676 -0.0402289 0.9679106        NA
## FBgn0000017 4352.55357     -0.2560570  0.112230 -2.2815471 0.0225161  0.130353
## FBgn0000018  418.61048     -0.0646152  0.131349 -0.4919341 0.6227659  0.859351
## FBgn0000024    6.40620      0.3089562  0.755886  0.4087340 0.6827349  0.887742
## FBgn0000032  989.72022     -0.0483792  0.120853 -0.4003139 0.6889253  0.890201
```

So, it looks like `type` has no effect. As a follow-up, I can also analyse with just `type` considered

```r
resMFType <- results(ddsMulti,
                     contrast=c("type", "single", "paired"))
head(resMFType)
```

```
## log2 fold change (MLE): type single vs paired
## Wald test p-value: type single vs paired
## DataFrame with 6 rows and 6 columns
##               baseMean log2FoldChange     lfcSE      stat    pvalue      padj
##              <numeric>      <numeric> <numeric> <numeric> <numeric> <numeric>
## FBgn0000008   95.14429      -0.262373  0.218505 -1.200767 0.2298414  0.536182
## FBgn0000014    1.05652       3.289885  2.052786  1.602644 0.1090133        NA
## FBgn0000017 4352.55357      -0.100020  0.112091 -0.892310 0.3722268  0.683195
## FBgn0000018  418.61048       0.229049  0.130261  1.758388 0.0786815  0.291789
## FBgn0000024    6.40620       0.306051  0.751286  0.407369 0.6837368  0.880472
## FBgn0000032  989.72022       0.237413  0.120286  1.973744 0.0484108  0.217658
```

When testing differential expression, typically raw counts and discrete distributions are used. However, for other downstream analyses, like visualisation or clustering, transforming the count data may be beneficial. `DESeq2` already uses the log2 fold change as a means of analysis, but there are other ones to consider.

In this discussion, I will consider the variance stabilizing transformation (VST), as noted in Tibshirani (1988); Huber et al. (2003); Anders and Huber (2010) and the regularized logarithm (rlog), as described in Love, Huber, and Anders (2014). These methods take log2 transformed data and normalize the data with respect to specific factors. The key of these transformations is to remove the dependence of the variance of the mean. In particular, there is a higher variance of logarithm of count data when the mean is low. They reduce the variance in comparison to experiment-wide factors.

The code to transform the data with VST and rlog is shown below.

```r
vsd = vst(dds, blind = F)
rld = rlog(dds, blind = F)
```

One thing to note is the `blind` argument of the functions. This tells the system whether or not to consider parameters of the experiment or not when doing analyses. For instance, in the case of the `pasilla` data, there are known factors in place, like `condition` that can explin the change in gene expression. In this case, it would be best to let `blind = FALSE` because I want the system to consider `condition` when estimating counts.

Now, the VST stabilises the variance based on the size of the gene. The transformed data is on the log2 scale for large counts

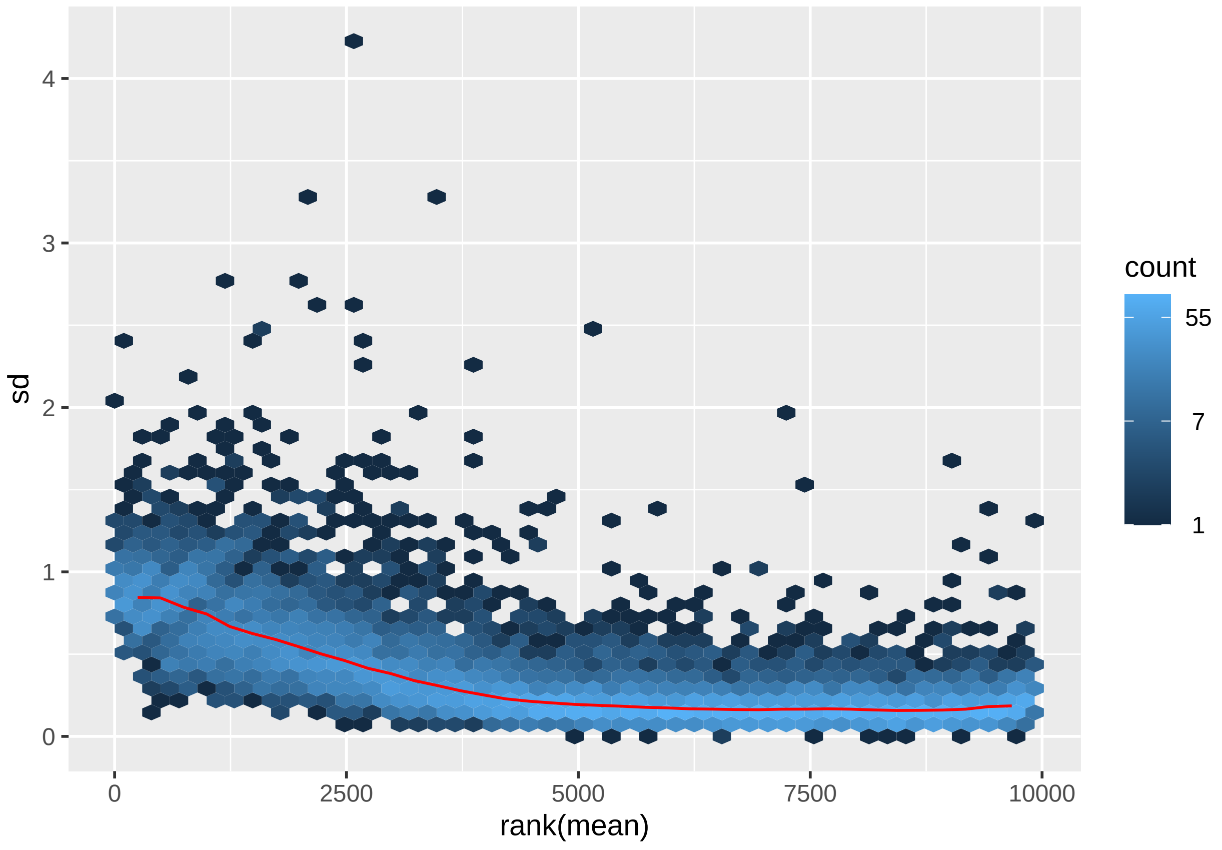
In comparison, the regularized log, takes the original count data to the log2 scale by fitting a model with a term for each sample and a prior distribution based on coefficients estimated from data. The formula is estimated as:

$$log_2(q_{ij}) = \beta_{i0} + \beta_{ij}$$

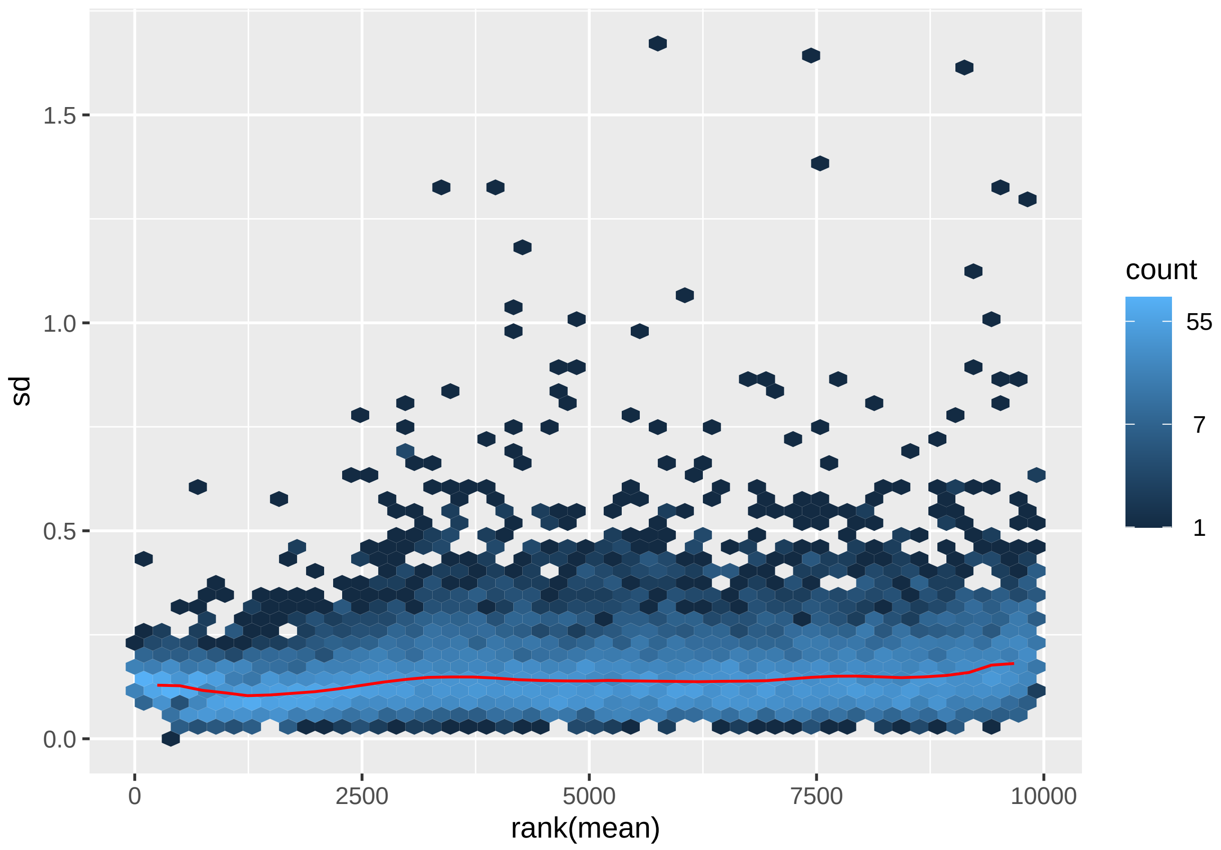
Here, q_{ij} is a parameter proportional to the expected true concentration of frgements for gene i and sample j, β_{i0} is like the background noise (there's always a little bit of error that cannot be explained) and therefore does not undergo shrinkage. β_{ij} is the sample specific coefficent which is shrunk toward zero based on the dispersion mean over the dataset. Because of its nature, `rlog` tends to have a larger effect of shrinkage.

Finally, it's time to plot these transformations. Below is code for transformed data across samples against the mean, using a shifted log transformation, the VST, and the rlog transformation.

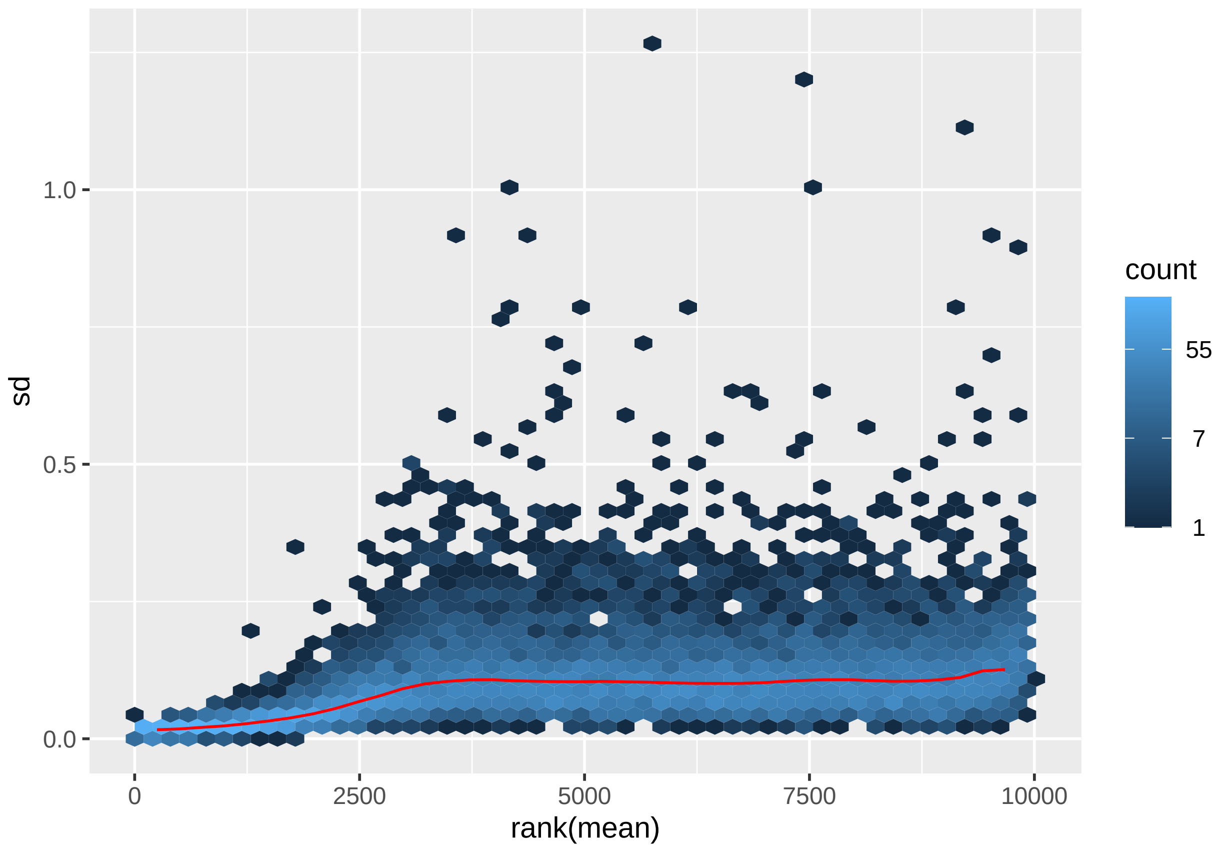
```
ntd = normTransform(dds)
library("vsn")
meanSdPlot(assay(ntd))
```



```
meanSdPlot(assay(vsd))
```



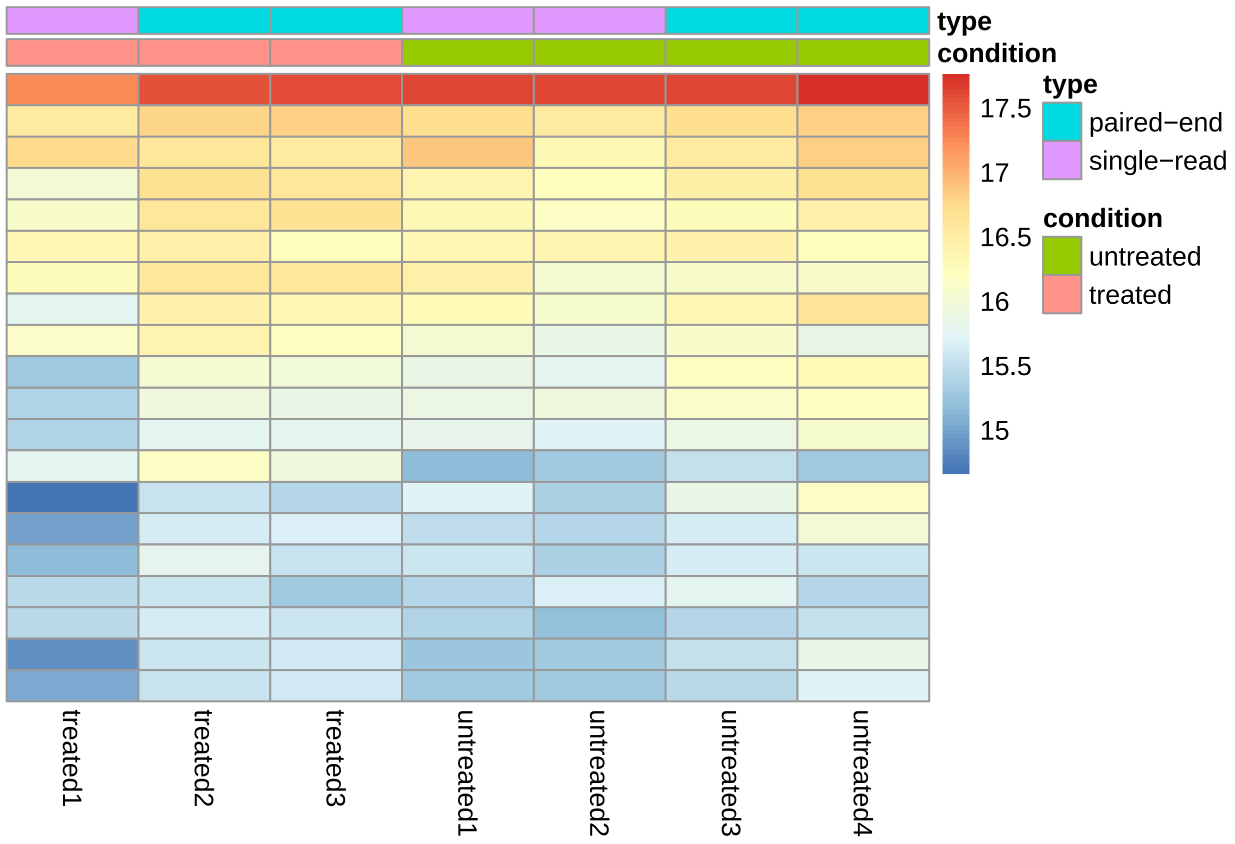
```r
meanSdPlot(assay(rld))
```



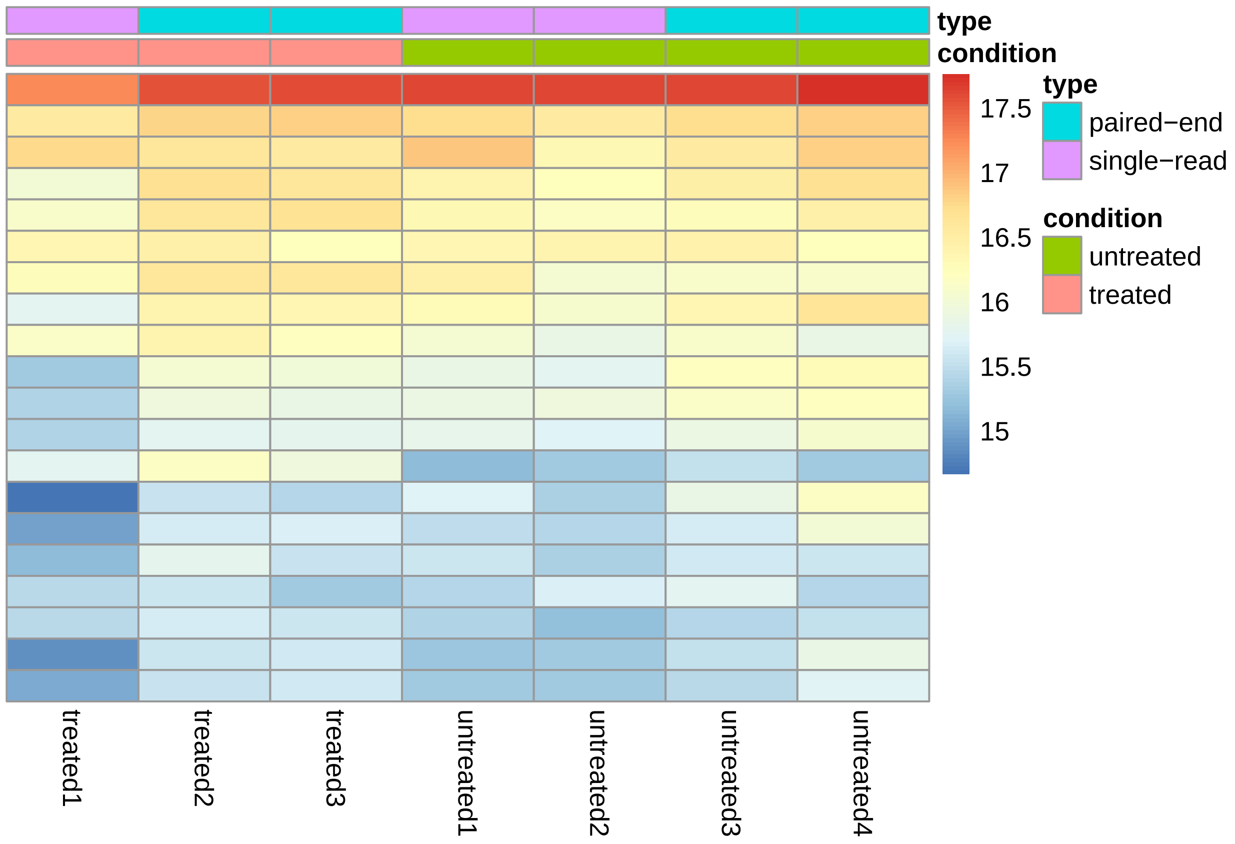
Another thing to do is to assess the quality of the data and remove bad data. I want to look in particular for samples where treatment showed abnormal results and would hurt downstream analyses.

Exploring count matrices for quality is best viewed as a heatmap. Below are heatmaps for the regular count matrix, the VST transformed count matrix, and the rlog transformed count matrix

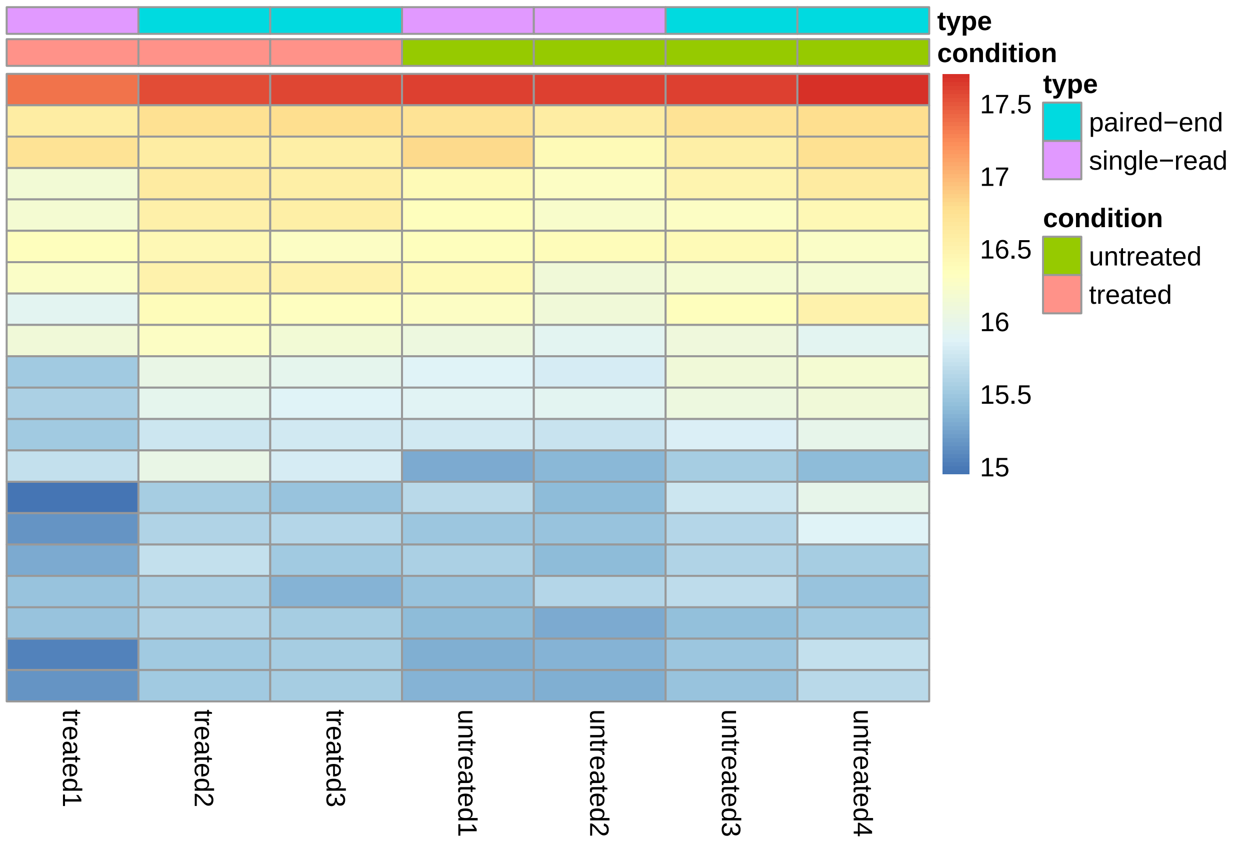
```
library(pheatmap)
select = order(rowMeans(counts(dds, normalized = T)),
               decreasing = T)[1:20]
df = as.data.frame(colData(dds)[,c("condition", "type")])
pheatmap(assay(ntd)[select,], cluster_rows = F, show_rownames = F,
         cluster_cols = F, annotation_col = df)
```



```
pheatmap(assay(vsd)[select,], cluster_rows = F, show_rownames = F,
         cluster_cols = F, annotation_col = df)
```



```r
pheatmap(assay(rld)[select,], cluster_rows = F, show_rownames = F,
         cluster_cols = F, annotation_col = df)
```



Sample to sample distances

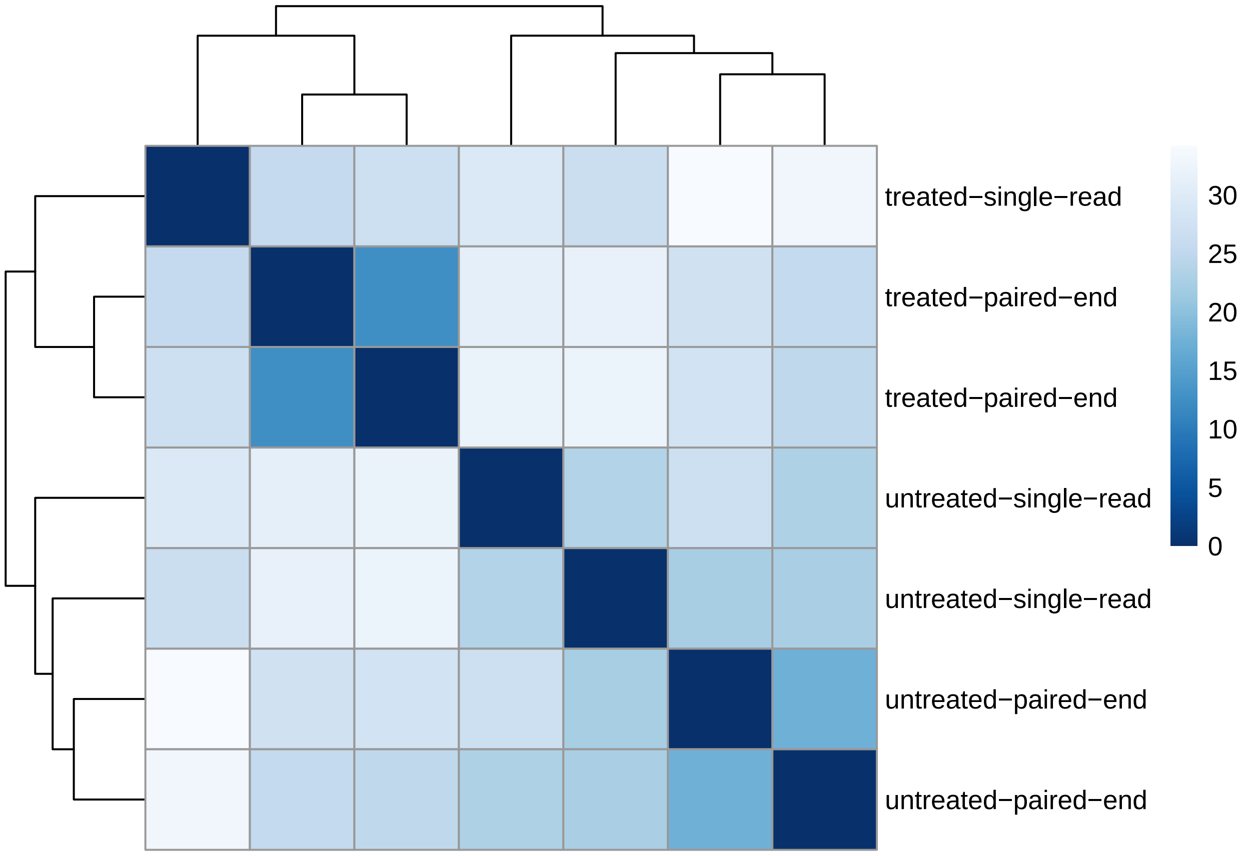
Another use of the transformed data is sample clustering. I can apply the `dist` function to the transpose of my transformed data to get distances between samples

```
(sampleDists = dist(t(assay(vsd))))
```

```
##            treated1 treated2 treated3 untreated1 untreated2 untreated3
## treated2   25.50386
## treated3   26.95680 12.56036
## untreated1 29.40565 31.01013 31.97611
## untreated2 26.55427 31.63020 32.23199   23.69489
## untreated3 34.19351 27.37454 27.85812   26.96435   22.49635
## untreated4 33.10003 25.39556 24.84995   23.17051   22.69880   17.51730
```

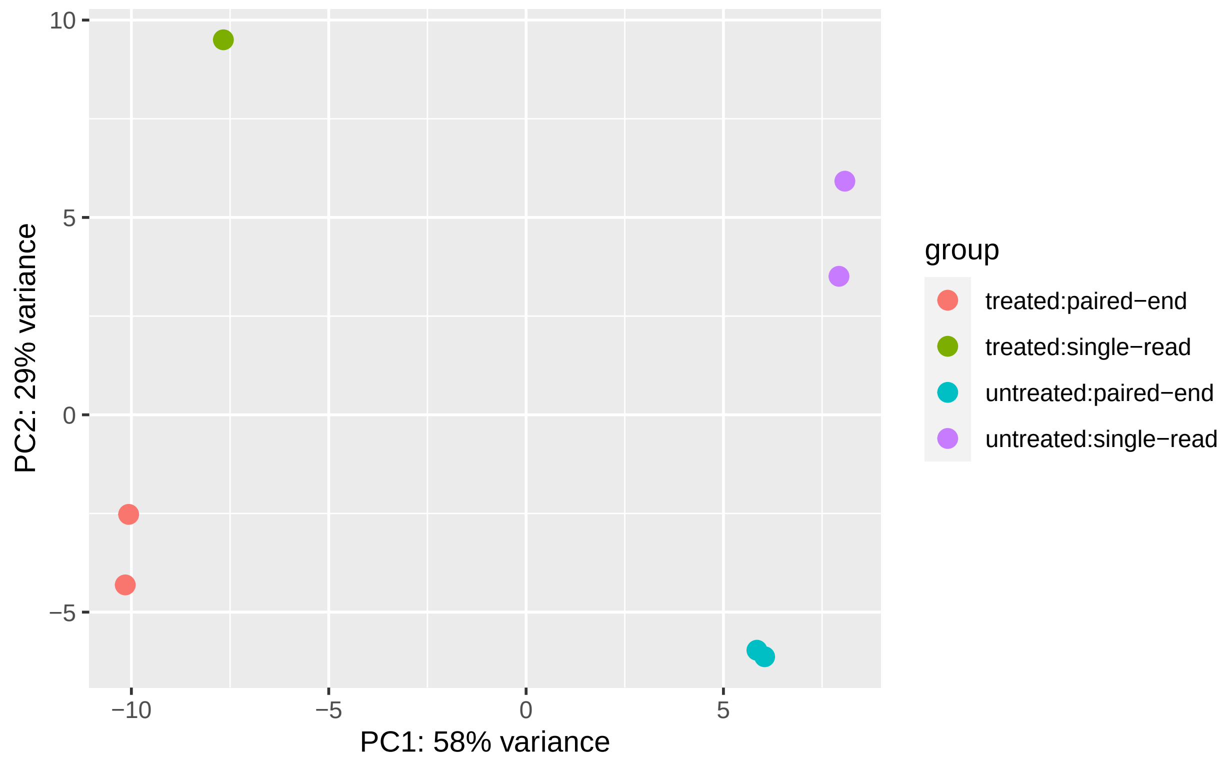
Then I can make a heatmap of the distances between samples.

```
library(RColorBrewer)
sampleDistMatrix = as.matrix(sampleDists)
rownames(sampleDistMatrix) = paste(vsd$condition, vsd$type, sep = "-")
colnames(sampleDistMatrix) = NULL
colors = colorRampPalette(rev(brewer.pal(9, "Blues")))(255)
pheatmap(sampleDistMatrix,
         clustering_distance_rows = sampleDists,
         clustering_distance_cols = sampleDists,
         col = colors)
```

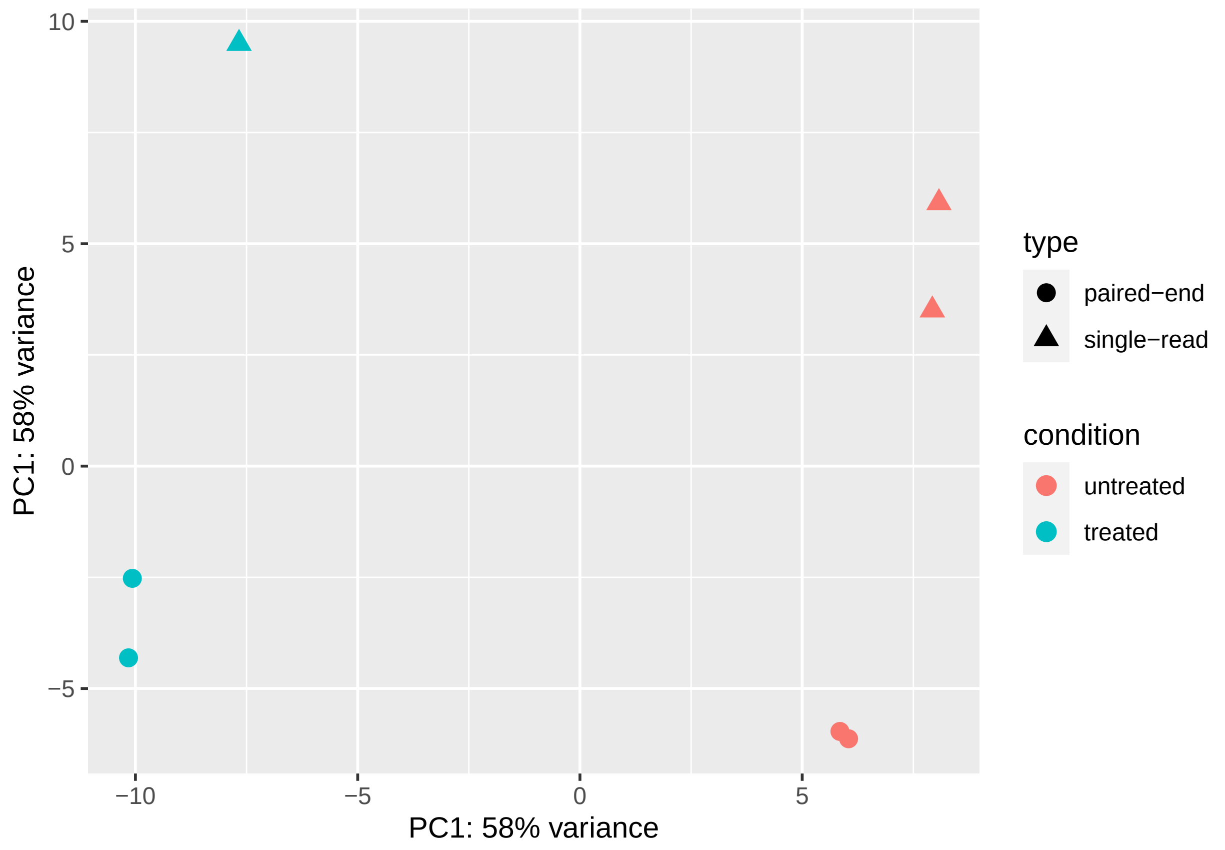


Related to the distance matrix is the PCA plot. This is useful for visualising the overall effect of experiment parameters.

```r
plotPCA(vsd, intgroup = c("condition", "type"))
```



```r
# Using ggplot
pcaData = plotPCA(vsd, intgroup = c("condition", "type"), returnData=T)
percentVar = round(100*attr(pcaData, "percentVar"))
ggplot(pcaData, aes(PC1, PC2, color = condition, shape = type)) +
  geom_point(size=3) +
  xlab(paste0("PC1: ", percentVar[1],"% variance")) +
  ylab(paste0("PC1: ", percentVar[1],"% variance")) +
  coord_fixed()
```



There's a wealth of even more information about `DESeq2`, but I think this is a good stopping place

Random Forest

The second test I'm using is the random forest. What I've found in my research is that this works a lot better as a selection algorithm of factors after running the results. I can take the average count of each gene, and build the random forest based on which treatment group gives the greatest change in expression. First thing I need to do is sort of clean up the data. I want to match up my original genes with there respective `baseMean` from the running the `DESeq` function. I converted everything to a tibble as it's a little bit easier to work with. Anywhere where the system didn't come up with a `baseMean` for a gene, I'm going to toss that out.

```
library(rsample)
library(randomForest)
library(randomForestExplainer)

RF = merge(cts, resMulti, by = "row.names", all.x = TRUE)
RF = as_tibble(RF)
(RF %<>% filter(!is.na(baseMean)) %>%
                  dplyr::select(`Row.names`,
                  untreated1, untreated2, untreated3, untreated4,
treated1, treated2, treated3,
baseMean))
```

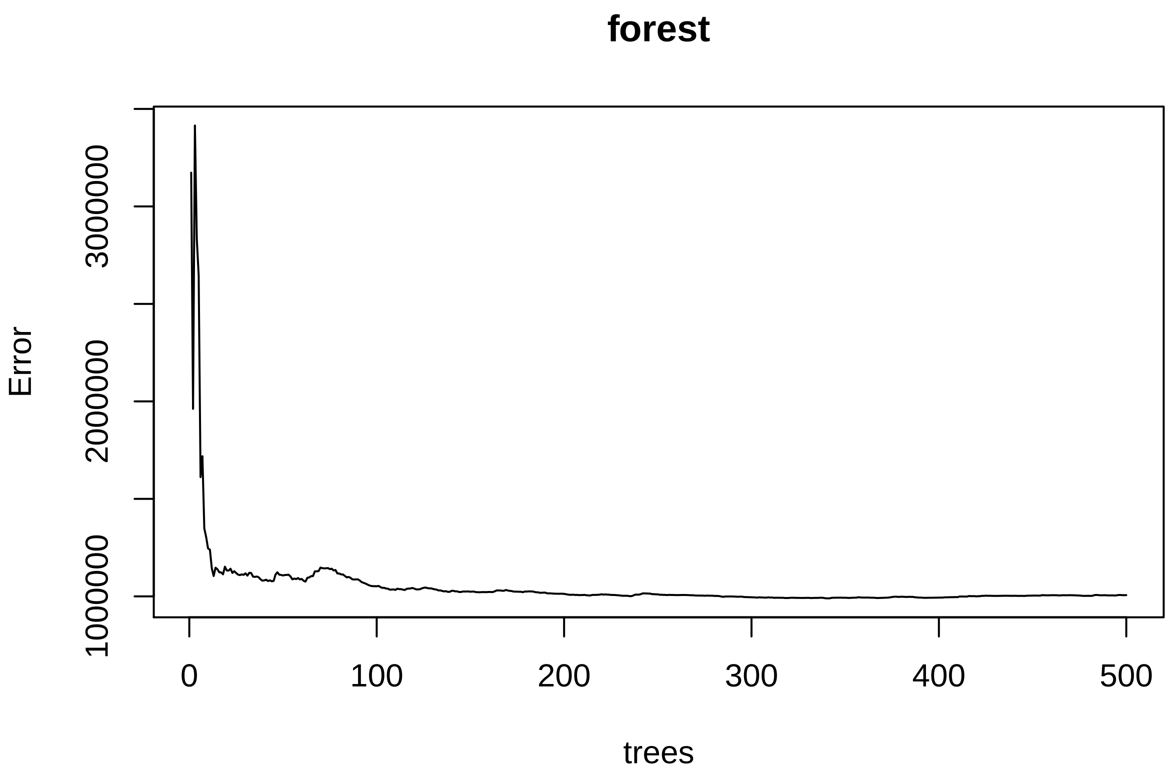
```
## # A tibble: 9,921 x 9
##    Row.names untreated1 untreated2 untreated3 untreated4 treated1 treated2
##    <I<chr>>       <int>      <int>      <int>      <int>    <int>    <int>
##  1 FBgn0000~         92        161         76         70      140       88
##  2 FBgn0000~          5          1          0          0        4        0
##   3 FBgn0000~       4664       8714       3564       3150      6205      3072
##   4 FBgn0000~        583        761        245        310       722       299
##   5 FBgn0000~         10         11          3          3        10         7
##   6 FBgn0000~       1446       1713        615        672      1698       696
##   7 FBgn0000~         15         25          9          5        20        14
##   8 FBgn0000~     101664     120163      45880      53201    127363     76099
##   9 FBgn0000~      33402      41118      16007      18360     56048     31421
##  10 FBgn0000~         21         63         15         13        64        28
## # ... with 9,911 more rows, and 2 more variables: treated3 <int>,
## #   baseMean <dbl>
```

I'll then take the counts for the respective treatment groups and build my random forest based on regression parameters. I'll then plot the amount of variance over the progression of the number of trees made.

```r
set.seed(5391)
(forest = randomForest(baseMean ~., data = RF))
```

```
##
## Call:
##  randomForest(formula = baseMean ~ ., data = RF)
##                Type of random forest: regression
##                      Number of trees: 500
## No. of variables tried at each split: 2
##
##           Mean of squared residuals: 1006190
##                     % Var explained: 95.48
```

```r
plot(forest)
```



I will admit, this one is still a work in progress, I'm still thinking about which data from the set I should use to build the data. In addition, the parameter of `baseMean` being estimated by the treatment groups may not be the best way of going about it.

Weighted Logged Odds

The first thing I did was to read the data into the system. I have a count matrix and a file with the annotations for the data.

```
library("pasilla")
library(tidyverse)
library(tidylo)
library(tidytuesdayR)
library(tidytext)

pasCTS = system.file("extdata",
                     "pasilla_gene_counts.tsv",
                     package = "pasilla", mustWork = TRUE)

pasAnno = system.file("extdata",
                      "pasilla_sample_annotation.csv",
                      package = "pasilla", mustWork = TRUE)
```

I'm going to read my count matrix as a data frame and then take a look at the data

```
cts = as.data.frame(read.csv(pasCTS, sep = "\t"))
head(cts, 4)
```

```
##       gene_id untreated1 untreated2 untreated3 untreated4 treated1 treated2
## 1 FBgn0000003          0          0          0          0        0        0
## 2 FBgn0000008         92        161         76         70      140       88
## 3 FBgn0000014          5          1          0          0        4        0
## 4 FBgn0000015          0          2          1          2        1        0
##   treated3
## 1        1
## 2       70
## 3        0
## 4        0
```

I'll now use the principles of tidy data to arrange the data properly

```
cts2 = cts%>%pivot_longer(untreated1:treated3, names_to = "condition", values_to = "counts")
head(cts2, 10)
```

```
## # A tibble: 10 x 3
##    gene_id     condition  counts
##    <chr>       <chr>       <int>
##  1 FBgn0000003 untreated1      0
##  2 FBgn0000003 untreated2      0
##  3 FBgn0000003 untreated3      0
##  4 FBgn0000003 untreated4      0
##  5 FBgn0000003 treated1        0
##  6 FBgn0000003 treated2        0
##  7 FBgn0000003 treated3        1
##  8 FBgn0000008 untreated1     92
##  9 FBgn0000008 untreated2    161
## 10 FBgn0000008 untreated3     76
```

Now, apply the weighted log odds from the `tidylo` package. Now, the vignette tells me to use the count function to count the data. However, I'm already given the exon counts, so I'm going to set my n to the counts column. Finally, I'll arrange the data by the greatest to least log odds ratio

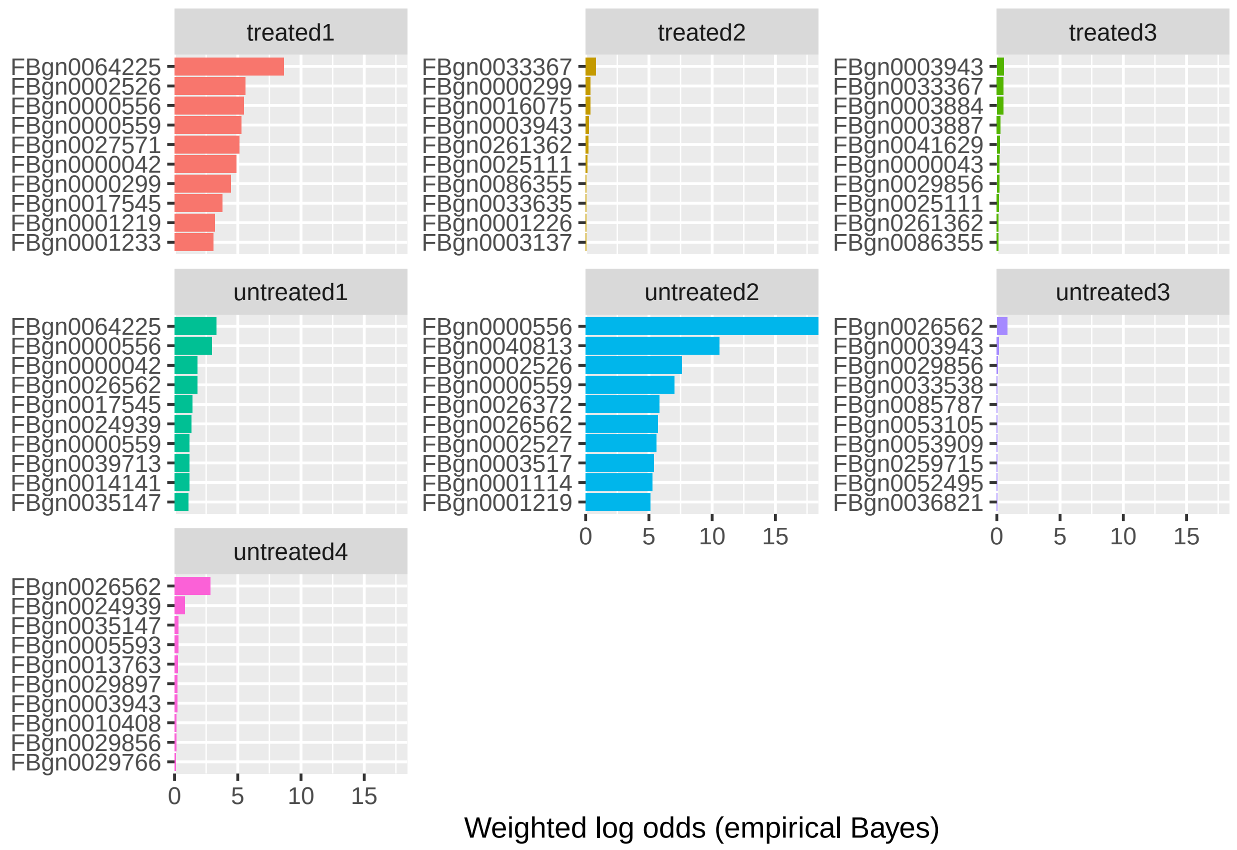
```r
n = cts2$counts
cts2 = cts2 %>% bind_log_odds(gene_id, condition, n)
cts2 %>% arrange(-log_odds_weighted)
```

```
## # A tibble: 102,193 x 4
##    gene_id     condition  counts log_odds_weighted
##    <chr>       <chr>       <int>             <dbl>
##  1 FBgn0000556 untreated2 360330             18.4
##  2 FBgn0040813 untreated2 139905             10.6
##  3 FBgn0064225 treated1   180986              8.62
##  4 FBgn0002526 untreated2 153771              7.60
##  5 FBgn0000559 untreated2 171656              7.03
##  6 FBgn0026372 untreated2 113692              5.84
##  7 FBgn0026562 untreated2  91503              5.73
##  8 FBgn0002527 untreated2  87970              5.58
##  9 FBgn0002526 treated1   136813              5.58
## 10 FBgn0000556 treated1   253500              5.48
## # ... with 102,183 more rows
```

And now to visualise the data, grouping by the sample with the largest log odds ratio based on treatment group

```r
conditions <- c("untreated1", "untreated2", "untreated3", "untreated4",
                "treated1", "treated2", "treated3")

cts2 %>%
  group_by(condition) %>%
  top_n(10) %>%
  ungroup() %>%
  mutate(weight = reorder_within(gene_id, log_odds_weighted, condition)) %>%
  ggplot(aes(log_odds_weighted, weight, fill = condition)) +
  geom_col(show.legend = FALSE) +
  facet_wrap(~condition, scales = "free_y") +
  scale_y_reordered() +
  scale_x_continuous(expand = c(0, 0)) +
  labs(y = NULL, x = "Weighted log odds (empirical Bayes)")
```


treated1
FBgn0064225
FBgn0002526
FBgn0000556
FBgn0000559
FBgn0027571
FBgn0000042
FBgn0000299
FBgn0017545
FBgn0001219
FBgn0001233
treated2
FBgn0033367
FBgn0000299
FBgn0016075
FBgn0003943
FBgn0261362
FBgn0025111
FBgn0086355
FBgn0033635
FBgn0001226
FBgn0003137
treated3
FBgn0003943
FBgn0033367
FBgn0003884
FBgn0003887
FBgn0041629
FBgn0000043
FBgn0029856
FBgn0025111
FBgn0261362
FBgn0086355
untreated1
FBgn0064225
FBgn0000556
FBgn0000042
FBgn0026562
FBgn0017545
FBgn0024939
FBgn0000559
FBgn0039713
FBgn0014141
FBgn0035147
untreated2
FBgn0000556
FBgn0040813
FBgn0002526
FBgn0000559
FBgn0026372
FBgn0026562
FBgn0002527
FBgn0003517
FBgn0001114
FBgn0001219
untreated3
FBgn0026562
FBgn0003943
FBgn0029856
FBgn0033538
FBgn0085787
FBgn0053105
FBgn0053909
FBgn0259715
FBgn0052495
FBgn0036821
untreated4
FBgn0026562
FBgn0024939
FBgn0035147
FBgn0005593
FBgn0013763
FBgn0029897
FBgn0003943
FBgn0010408
FBgn0029856
FBgn0029766
0
5
10
15
Weighted log odds (empirical Bayes)