Class 12: Transcriptomics and the analysis of RNA-Seq data

Nicholas Yousefi

In this lab, we will analyze the data from the paper by Himes et al.

Importing the Data

We will use `read.csv()` to read the two things we need for this analysis: - count data - col data (metadata)

We will then look at the data and metadata.

```
counts <- read.csv("airway_scaledcounts.csv", row.names=1)
metadata <- read.csv("airway_metadata.csv")

head(counts)
```

	SRR1039508	SRR1039509	SRR1039512	SRR1039513	SRR1039516
ENSG00000000003	723	486	904	445	1170
ENSG00000000005	0	0	0	0	0
ENSG00000000419	467	523	616	371	582
ENSG00000000457	347	258	364	237	318
ENSG00000000460	96	81	73	66	118
ENSG00000000938	0	0	1	0	2

	SRR1039517	SRR1039520	SRR1039521
ENSG00000000003	1097	806	604
ENSG00000000005	0	0	0
ENSG00000000419	781	417	509
ENSG00000000457	447	330	324
ENSG00000000460	94	102	74
ENSG00000000938	0	0	0

```
head(metadata)
```

```
         id     dex celltype     geo_id
1 SRR1039508 control   N61311 GSM1275862
2 SRR1039509 treated   N61311 GSM1275863
3 SRR1039512 control  N052611 GSM1275866
4 SRR1039513 treated  N052611 GSM1275867
5 SRR1039516 control  N080611 GSM1275870
6 SRR1039517 treated  N080611 GSM1275871
```

Q1. How many genes are in this dataset?

```
nrow(counts)
```

```
[1] 38694
```

Q2. How many 'control' cell lines do we have?

```
table(metadata$dex)
```

```
control treated
      4       4
```

First, we should do a sanity check to make sure the metadata matches with the actual data. We want to make sure the ID column in the metadata matches the col names of the counts and that they are in the same order.

To do this, we use the `==` test.

```
all(metadata$id == colnames(counts))
```

```
[1] TRUE
```

The `all()` function returns `TRUE` if and only if all the elements in the vector are `TRUE`. If even one element is `FALSE`, `all()` will return `FALSE`.

Analysis via comparison of CONTROL vs. TREATED

The "treated" have the dex drug and the "control" do not. First, I need to be able to extract just the "control" columns in the `counts` dataset.

```
control <- metadata[metadata$dex == "control",]
control$id
```

```
[1] "SRR1039508" "SRR1039512" "SRR1039516" "SRR1039520"
```

Now I can use this to access just the `"control"` columns of my `counts` data…

```
control.counts <- counts[,control$id]
head(control.counts)
```

```
                SRR1039508 SRR1039512 SRR1039516 SRR1039520
ENSG00000000003        723        904       1170        806
ENSG00000000005          0          0          0          0
ENSG00000000419        467        616        582        417
ENSG00000000457        347        364        318        330
ENSG00000000460         96         73        118        102
ENSG00000000938          0          1          2          0
```

Find the mean count value for each transcript/gene by finding the `rowMeans()`.

```
control.mean <- rowMeans(control.counts)
head(control.mean)
```

```
ENSG00000000003 ENSG00000000005 ENSG00000000419 ENSG00000000457 ENSG00000000460
         900.75            0.00          520.50          339.75           97.25
ENSG00000000938
           0.75
```

Q3. How would you make the above code in either approach more robust?

This is what "the above code" refers to in this question (the code we did in class was slightly different):

```
control <- metadata[metadata[,"dex"]=="control",]
control.counts <- counts[ ,control$id]
control.mean <- rowSums( control.counts )/4
head(control.mean)
```

This code calculates the mean count across all 4 control samples for each gene. It does so by taking the sum of each row in `control.counts` and dividing by 4. As can be seen, the 4 is hardcoded into the program. The code `rowSums( control.counts )/4` can only take the average of a set of 4 numbers. But if we had more than 4 control samples, the code would add up the counts for all those samples and then divide by 4, giving an incorrect average.

To make this code more robust, we could change `rowSums( control.counts )/4` to `rowMeans(control.counts)` in the above code. The `rowMeans()` function calculates the average of each row in the data frame correctly, regardless of how many elements are in that row. Therefore, if we had more than 4 samples in `control.counts`, the `rowMeans()` function would calculate the correct means.

The fixed code would look like this:

```
control <- metadata[metadata[,"dex"]=="control",]
control.counts <- counts[ ,control$id]
control.mean <- rowMeans(control.counts)
head(control.mean)
```

The code we did in class already had this issue fixed.

Let's do the same thing for the treated.

> Q4. Follow the same procedure for the treated samples (i.e. calculate the mean per gene across drug treated samples and assign to a labeled vector called treated.mean)

```
treated <- metadata[metadata$dex == "treated",]
treated.counts <- counts[treated$id]
treated.mean <- rowMeans(treated.counts)
head(treated.mean)
```

```
ENSG00000000003 ENSG00000000005 ENSG00000000419 ENSG00000000457 ENSG00000000460
         658.00            0.00          546.00          316.50           78.75
ENSG00000000938
           0.00
```

Now I have `control.mean` and `treated.mean`. Lets put them together for safekeeping and ease of use later.

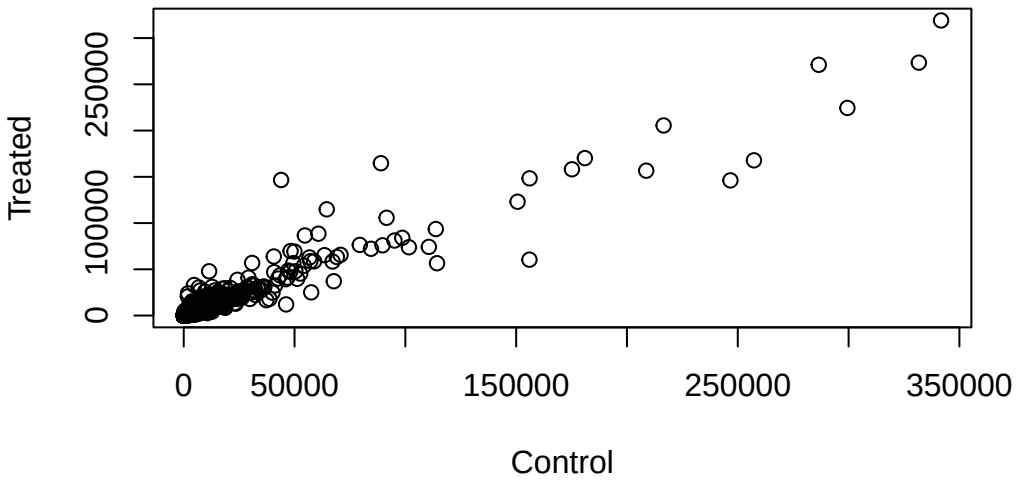
```
meancounts <- data.frame(control.mean, treated.mean)
head(meancounts)
```

```
                control.mean treated.mean
ENSG00000000003       900.75       658.00
ENSG00000000005         0.00         0.00
ENSG00000000419       520.50       546.00
ENSG00000000457       339.75       316.50
ENSG00000000460        97.25        78.75
ENSG00000000938         0.75         0.00
```

Let's do a quick plot to see how our data looks.

Q5 (a). Create a scatter plot showing the mean of the treated samples against the mean of the control samples. Your plot should look something like the following.

```
plot(meancounts, xlab="Control", ylab="Treated")
```



Q5 (b).You could also use the ggplot2 package to make this figure producing the plot below. What geom_?() function would you use for this plot?

`geom_point()`

This data is very heavily skewed and over a wide range. So, we must take the log transform of it so we can see it better.

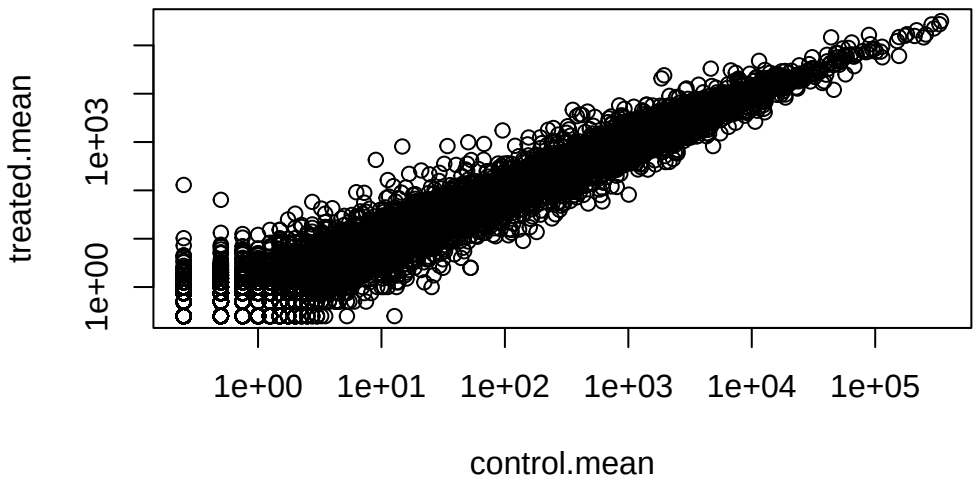
> Q6. Try plotting both axes on a log scale. What is the argument to plot() that allows you to do this?

`log="xy"`

```
plot(meancounts, log="xy")
```

```
Warning in xy.coords(x, y, xlabel, ylabel, log): 15032 x values <= 0 omitted
from logarithmic plot
```

```
Warning in xy.coords(x, y, xlabel, ylabel, log): 15281 y values <= 0 omitted
from logarithmic plot
```



We like working with log transformed data as it can help make things more straightforward to interpret.

```
log2(20/20) # if we take log2 of this, it is 0 if there is no change
```

```
[1] 0
```

What about if we had a doubling

```
log2(40/20) # gives us a log2 fold change of 1
```

```
[1] 1
```

```
log2(10/20) # halving gives a log2 fold change of -1
```

```
[1] -1
```

```
log2(80/20) # larger changes have  higher values
```

```
[1] 2
```

The thing in parentheses is called the fold change. When we take log2 of it, it is called log2(fold change)

We like working with log2(fold-change) values. Let's calculate them for our data.

```
meancounts$log2fc <- log2(meancounts$treated.mean / meancounts$control.mean)
head(meancounts)
```

```
                control.mean treated.mean      log2fc
ENSG00000000003       900.75       658.00 -0.45303916
ENSG00000000005         0.00         0.00         NaN
ENSG00000000419       520.50       546.00  0.06900279
ENSG00000000457       339.75       316.50 -0.10226805
ENSG00000000460        97.25        78.75 -0.30441833
ENSG00000000938         0.75         0.00        -Inf
```

We want to filter out any genes (that is the rows) where we have ZERO count data.

```
to.keep.inds <- rowSums(meancounts[,1:2] == 0) == 0
head(to.keep.inds)
```

```
ENSG00000000003 ENSG00000000005 ENSG00000000419 ENSG00000000457 ENSG00000000460
           TRUE           FALSE            TRUE            TRUE            TRUE
ENSG00000000938
          FALSE
```

```
mycounts <- meancounts[to.keep.inds,]
nrow(mycounts)
```

```
[1] 21817
```

> Q7. What is the purpose of the arr.ind argument in the which() function call above? Why would we then take the first column of the output and need to call the unique() function?

Again, in class we used slightly different code from what is in the lab instructions This is what was in the lab instructions (which the question is referring to):

```
zero.vals <- which(meancounts[,1:2]==0, arr.ind=TRUE)

to.rm <- unique(zero.vals[,1])
mycounts <- meancounts[-to.rm,]
head(mycounts)
```

If we left the `arr.ind` argument as its default value, `FALSE`, the `which()` function would have returned a 1 dimensional vector whose length is the number of zeros in the `meancounts` data frame. This vector would have essentially contained the index numbers of the positions of the zeros if column 2 were added to the bottom/end of column 1 in `meancounts`. This would have been harder to work with, so we set `arr.ind` to `TRUE` instead. Setting `arr.ind` to `TRUE` causes the `which()` function to return a 2 column matrix containing the row and column positions of each zero in the `meancounts` data frame. This is much easier to work with.

Our goal is to remove any row in `meancounts` that contains a zero in either column. Therefore, we only care about the row coordinates of the zeros, not the column coordinates. Thus, we can just take the first column of `zero.vals` which contains the row coordinates of all the zeros.

There are some rows in `meancounts` that have zeros in both the `control.mean` and `treated.mean` columns. These rows appear twice in the `row` column of `zero.vals`. We want to create a vector containing only the numbers of the rows to remove from `meancounts`. We do not want any row indices to appear twice in this vector. The `unique()` function removes

any duplicate row indices, making `to.rm` contain the indices of the rows to be removed from `meancounts` without any duplicates.

A common threshold for calling genes as differentially expressed is a log2 fold-change of +2 or -2.

```
sum(mycounts$log2fc >= 2)
```

```
[1] 314
```

What percent is this?

```
round((sum(mycounts$log2fc >= 2) / nrow(mycounts)) * 100, 2)
```

```
[1] 1.44
```

Let's find the proportion that is downregulated.

```
round((sum(mycounts$log2fc <= -2) / nrow(mycounts)) * 100, 2)
```

```
[1] 2.22
```

> Q8. Using the up.ind vector above can you determine how many up regulated genes we have at the greater than 2 fc level?

```
up.ind <- mycounts$log2fc > 2
sum(up.ind)
```

```
[1] 250
```

> Q9. Using the down.ind vector above can you determine how many down regulated genes we have at the greater than 2 fc level?

```
down.ind <- mycounts$log2fc < -2
sum(down.ind)
```

```
[1] 367
```

> Q10. Do you trust these results? Why or why not?

Nope! The results could have big fold changes that are not statistically significant. We need to do statistical tests to see which results, of the ones with a log2(fold-change) > 2 or < -2, which ones are statistically significant.

The last thing we need is a statistical test to check if the drug induced difference is significant! We can do this using DESeq2.

Turn to DESeq2

Let's turn to doing this the correct way with the DESeq2 package.

```
library(DESeq2)
```

The main function in the DESeq2 package is called `deseq()`. It wants our count data and our colData (metadata) as input in a specific way.

```
dds <- DESeqDataSetFromMatrix(countData = counts,
                              colData = metadata,
                              design = ~dex)
```

```
converting counts to integer mode
```

```
Warning in DESeqDataSet(se, design = design, ignoreRank): some variables in
design formula are characters, converting to factors
```

Now I can run the DESeq analysis.

```
dds <- DESeq(dds) # the dds object has slots for the results, so save them to these slots
```

```
estimating size factors
```

```
estimating dispersions
```

```
gene-wise dispersion estimates
```

```
mean-dispersion relationship
```

```
final dispersion estimates
```

```
fitting model and testing
```

```
results(dds)
```

```
log2 fold change (MLE): dex treated vs control
Wald test p-value: dex treated vs control
DataFrame with 38694 rows and 6 columns
                 baseMean log2FoldChange     lfcSE      stat    pvalue
                <numeric>      <numeric> <numeric> <numeric> <numeric>
ENSG00000000003  747.1942     -0.3507030  0.168246 -2.084470 0.0371175
ENSG00000000005    0.0000             NA        NA        NA        NA
ENSG00000000419  520.1342      0.2061078  0.101059  2.039475 0.0414026
ENSG00000000457  322.6648      0.0245269  0.145145  0.168982 0.8658106
ENSG00000000460   87.6826     -0.1471420  0.257007 -0.572521 0.5669691
...                    ...            ...       ...       ...       ...
ENSG00000283115  0.000000             NA        NA        NA        NA
ENSG00000283116  0.000000             NA        NA        NA        NA
ENSG00000283119  0.000000             NA        NA        NA        NA
ENSG00000283120  0.974916      -0.668258   1.69456 -0.394354  0.693319
ENSG00000283123  0.000000             NA        NA        NA        NA
                     padj
                <numeric>
ENSG00000000003  0.163035
ENSG00000000005        NA
ENSG00000000419  0.176032
ENSG00000000457  0.961694
ENSG00000000460  0.815849
...                   ...
ENSG00000283115        NA
ENSG00000283116        NA
ENSG00000283119        NA
ENSG00000283120        NA
ENSG00000283123        NA
```

Now what we have got so far is the log2 fold-change and the adjusted p-value for the significance.

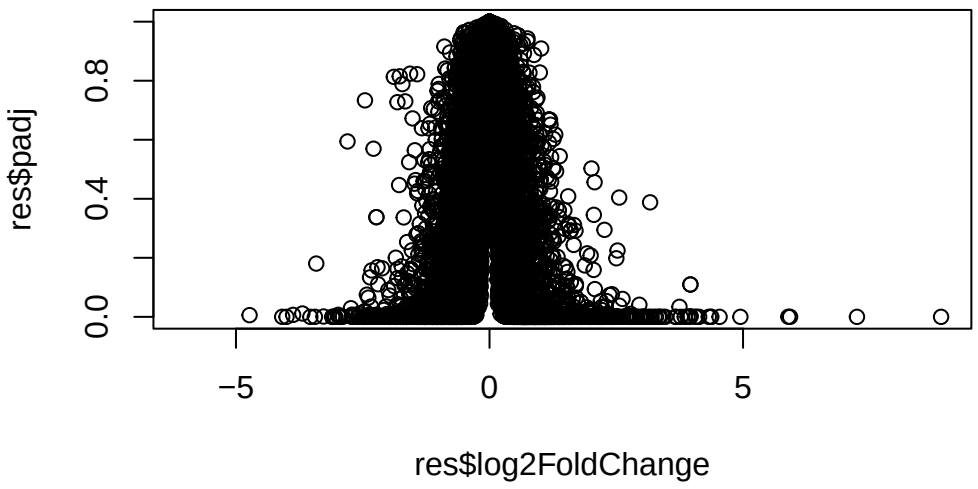
```
res <- results(dds)
head(res)
```

```
log2 fold change (MLE): dex treated vs control
Wald test p-value: dex treated vs control
DataFrame with 6 rows and 6 columns
```

```
                  baseMean log2FoldChange     lfcSE      stat    pvalue
                 <numeric>      <numeric> <numeric> <numeric> <numeric>
ENSG00000000003 747.194195     -0.3507030  0.168246 -2.084470 0.0371175
ENSG00000000005   0.000000             NA        NA        NA        NA
ENSG00000000419 520.134160      0.2061078  0.101059  2.039475 0.0414026
ENSG00000000457 322.664844      0.0245269  0.145145  0.168982 0.8658106
ENSG00000000460  87.682625     -0.1471420  0.257007 -0.572521 0.5669691
ENSG00000000938   0.319167     -1.7322890  3.493601 -0.495846 0.6200029
                     padj
                <numeric>
ENSG00000000003  0.163035
ENSG00000000005        NA
ENSG00000000419  0.176032
ENSG00000000457  0.961694
ENSG00000000460  0.815849
ENSG00000000938        NA
```

A first plot

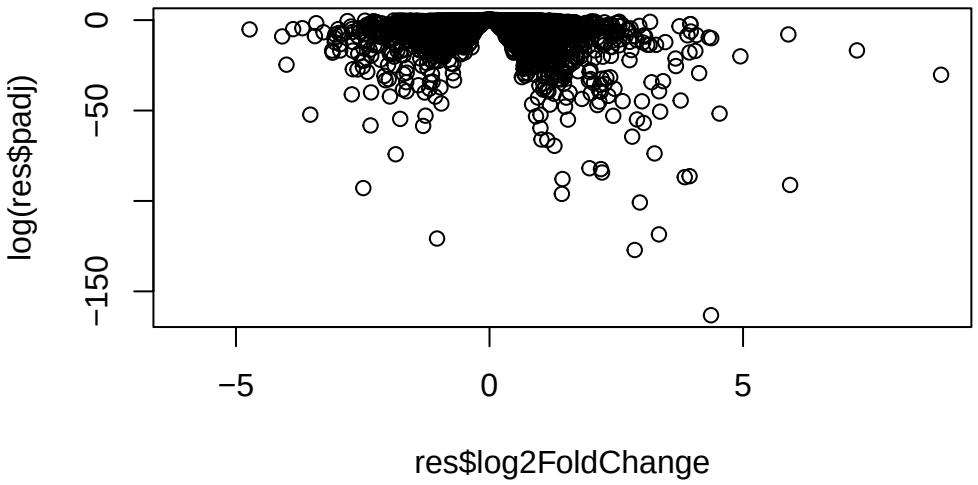
```
plot(res$log2FoldChange, res$padj)
```



Well that plot sucked. All the interesting P-values are down near zero

I am going to take the log of the p-value.

```
plot(res$log2FoldChange, log(res$padj))
```

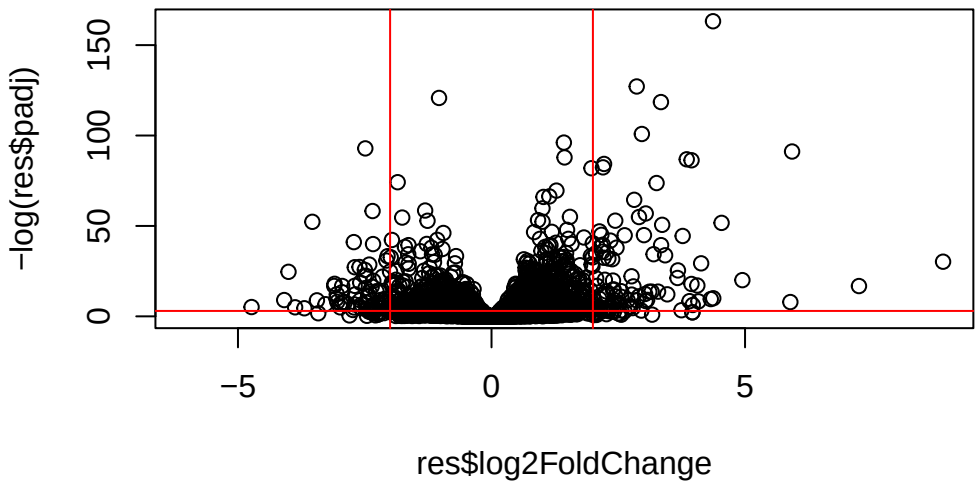



```
log(0.05)
```

```
[1] -2.995732
```

We can flip the y-axis so the plot does not look "upside down"

```
plot(res$log2FoldChange, -log(res$padj)) # volcano plot
abline(v=c(-2, 2), col="red")
abline(h=-log(0.05), col="red")
```



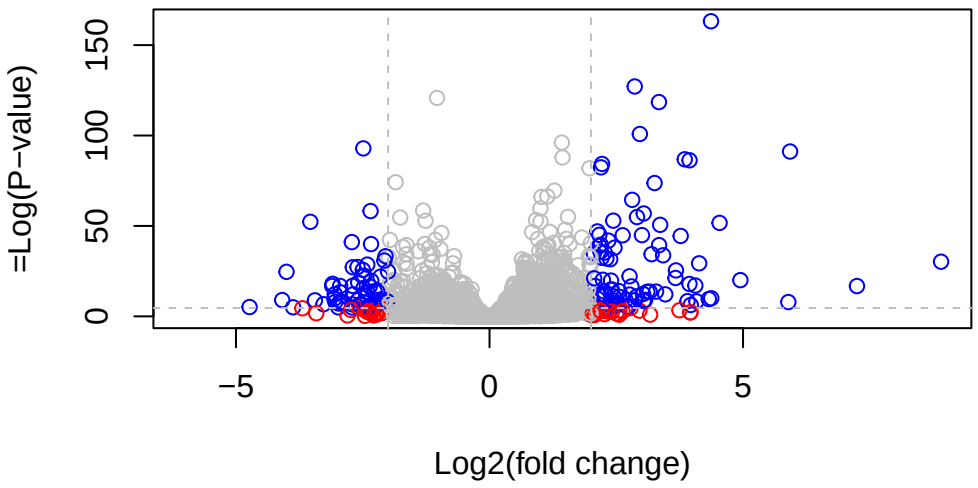
Let's color this plot to make it easier to visualize.

```
mycols = rep("gray", nrow(res))
mycols[abs(res$log2FoldChange) > 2] <- "red" # color anything with log2 fold change greate

inds <-  (res$padj < 0.01) & (abs(res$log2FoldChange) > 2)
mycols[inds] <- "blue"

# volcano plot with custom colors
plot(res$log2FoldChange, -log(res$padj), col=mycols, ylab="=Log(P-value)", xlab="Log2(fold

# add cutoff lines
abline(v=c(-2, 2), col="gray", lty=2)
abline(h=-log(0.01), col="gray", lty=2)
```



This is as far as we got in class on 11/3/2022. We will do the rest later.