Analysis of RNA-seq Data Generated Following Mouse Traumatic Brain Injury and Lymphatic Ablation

Arun Brendan Dutta

January 14, 2020

A step-by-step procedure for analysis of RNA-seq data generated by Ashley Bolte of the Lukens lab.

Contents

List of Figures

1 Introduction

The following instructions will walk you through analyzing your RNA-seq data. We will start from the .fastq files delivered by Genewiz and go all the way through plot generation. We will use both the command line and R to analyze your data. I would suggest copy and pasting all the code into a text editor program before using it in the terminal or in R. TextEdit is preinstalled on Macs, but I use Aquamacs for all my text editing (http://aquamacs.org). The reason for using a text editor is that copy and pasting code directly from a .pdf file into the terminal or into R can screw up the formatting and render the code inoperable. Moving into a text editor first means you can edit the code and copy and paste it into terminal / R knowing that it won't get altered in the transition. Note that Microsoft Word is not a text editor, it's a word processor that will also mess up your formatting. One last note is that whenever you see a line of code end with ', it means that command continues on the next line of the document.

2 Installing Software Packages

Before touching your data, you will have to install all the required software packages. This will probably be the hardest and most time-consuming part of the pipeline. Here's a list of all the software you'll need for the shell scripting in the terminal: twoBitToFa, hisat2, samtools, bedtools, python3, htseq-count.
Here's a list of all the software you'll need for the R code: lattice, DESeq2, GSA, seq2pathway, pheatmap.
Installing packages in R is pretty straightforward. In Rstudio you just go to the 'packages' tab on the right, click the 'install' button, and type in the name of the package you need. Make sure the 'Install dependencies' box is checked. After installation you can load packages into your R environment to use the programs contained therein. You can do this manually by checking the boxes next the names of the packages in the panel on the right, but we have that step automated in the scripts below.
The shell script packages are going to be a little more difficult. In general when installing software packages, I would Google the name of the package and go to their website. Look for a 'Manual' or 'Installation Guide' and follow the instructions there. For example, if you go to the bedtools website (`https://bedtools.readthedocs.io/en/latest`) and go to the 'Installation' page it tells you exactly what you need to type into the terminal to install the package. To see if a software package is successfully installed / accessible from the directory you're working in just type a command in the terminal or R (ex. hisat2) without anything else and you should get a little readout of how to run the command. If it says something like 'Command Not Found', then you know it's not installed / accessible. The commands below will hopefully walk you through the installation, but it's been a long time since I downloaded these so there may be steps I'm missing. This can be a frustrating process so let me know if you are having trouble.

```
#hisat2 installation
wget ftp://ftp.ccb.jhu.edu/pub/infphilo/hisat2/downloads/hisat2-2.1.0-OSX_x86_64.zip
mv hisat2-2.1.0 /usr/local/bin

#twoBitToFa installation
wget http://hgdownload.soe.ucsc.edu/admin/exe/macOSX.x86_64/twoBitToFa
chmod +x twoBitToFa
mv twoBitToFa /usr/local/bin

#for the other packages we're going to use a software package manager called homebrew
#homebrew makes installing and updating software on the command line much easier and quicker
#so first we're going to install homebrew, then the other packages

#homebrew installation
/usr/bin/ruby -e "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/master/install)"
#think you'll have to hit enter and enter your password at some point during the installation

#now install the specific packages
brew tap homebrew/science
brew install python3
brew install samtools
brew install bedtools

#if homebrew doesn't work, we can try getting the packages directly from the source
#samtools installation
wget https://github.com/samtools/samtools/releases/download/1.9/samtools-1.9.tar.bz2
tar -xvf samtools-1.9.tar.bz2
```

```
cd samtools-1.9
./configure --prefix=/usr/local/
make
make install

#bedtools installation
curl http://bedtools.googlecode.com/files/BEDTools.2.29.0.tar.gz > BEDTools.tar.gz
tar -zxvf BEDTools.tar.gz
cd BEDTools-2.29.0
make clean
make all
ls bin

#htseq-count can be installed with pip3, which is automatically installed with python3
pip3 install HTseq
```

3 Processing Sequencing Reads

3.1 Making a Directory

You'll want a dedicated directory, or folder, to hold all your associated files / code.

```
#mkdir creates directories
#we'll make one called RNAseq_analysis
mkdir RNAseq_analysis
#cd is used to move into directories
cd RNAseq_analysis
```

3.2 Concatenating Reads

Genewiz returned the sequencing reads as zipped .fastq files. There is one .fastq file per paired end per sample, so 2*16 samples = 32 .fastq files. This first code chunk renames the .fastq files to something more informative. You can see how we've organized the code to concatenate the different files based on the sample and whether it's paired end one ('R1') or paired end two ('R2'). The 'mv' or move command below can either rename files or move them to different directories.

```
#lines that start with a pound sign are just comments and aren't read as commands by the computer
#note how we use 'R1' and 'R2' to identify paired end 1 and paired end 2

#Rename no ablation + TBI group
mv V1_R1_001.fastq.gz noAb_TBI_rep1_PE1.fastq.gz
mv V1_R2_001.fastq.gz noAb_TBI_rep1_PE2.fastq.gz
mv V2_R1_001.fastq.gz noAb_TBI_rep2_PE1.fastq.gz
mv V2_R2_001.fastq.gz noAb_TBI_rep2_PE2.fastq.gz
mv V4_R1_001.fastq.gz noAb_TBI_rep3_PE1.fastq.gz
mv V4_R2_001.fastq.gz noAb_TBI_rep3_PE2.fastq.gz
mv V6_R1_001.fastq.gz noAb_TBI_rep4_PE1.fastq.gz
mv V6_R2_001.fastq.gz noAb_TBI_rep4_PE2.fastq.gz

#Rename ablation + TBI group
mv V-A1_R1_001.fastq.gz Ab_TBI_rep1_PE1.fastq.gz
mv V-A1_R2_001.fastq.gz Ab_TBI_rep1_PE2.fastq.gz
mv V-A2_R1_001.fastq.gz Ab_TBI_rep2_PE1.fastq.gz
mv V-A2_R2_001.fastq.gz Ab_TBI_rep2_PE2.fastq.gz
mv V-A5_R1_001.fastq.gz Ab_TBI_rep3_PE1.fastq.gz
mv V-A5_R2_001.fastq.gz Ab_TBI_rep3_PE2.fastq.gz
mv V-A6_R1_001.fastq.gz Ab_TBI_rep4_PE1.fastq.gz
mv V-A6_R2_001.fastq.gz Ab_TBI_rep4_PE2.fastq.gz

#Rename no ablation + sham group
mv V2-noTBI_R1_001.fastq.gz noAb_Sham_rep1_PE1.fastq.gz
mv V2-noTBI_R2_001.fastq.gz noAb_Sham_rep1_PE2.fastq.gz
mv V4-noTBI_R1_001.fastq.gz noAb_Sham_rep2_PE1.fastq.gz
mv V4-noTBI_R2_001.fastq.gz noAb_Sham_rep2_PE2.fastq.gz
mv V5-noTBI_R1_001.fastq.gz noAb_Sham_rep3_PE1.fastq.gz
mv V5-noTBI_R2_001.fastq.gz noAb_Sham_rep3_PE2.fastq.gz
mv V6-noTBI_R1_001.fastq.gz noAb_Sham_rep4_PE1.fastq.gz
mv V6-noTBI_R2_001.fastq.gz noAb_Sham_rep4_PE2.fastq.gz

#Rename ablation + sham group
mv V-A2-noTBI_R1_001.fastq.gz Ab_Sham_rep1_PE1.fastq.gz
mv V-A2-noTBI_R2_001.fastq.gz Ab_Sham_rep1_PE2.fastq.gz
mv V-A4-noTBI_R1_001.fastq.gz Ab_Sham_rep2_PE1.fastq.gz
mv V-A4-noTBI_R2_001.fastq.gz Ab_Sham_rep2_PE2.fastq.gz
mv V-A5-noTBI_R1_001.fastq.gz Ab_Sham_rep3_PE1.fastq.gz
mv V-A5-noTBI_R2_001.fastq.gz Ab_Sham_rep3_PE2.fastq.gz
```

```
mv V-A6-noTBI_R1_001.fastq.gz Ab_Sham_rep4_PE1.fastq.gz
mv V-A6-noTBI_R2_001.fastq.gz Ab_Sham_rep4_PE2.fastq.gz
```

3.3 Building a Genomic Index

We will align to the latest assembly of the mouse genome, mm10. We can retrieve this assembly from UCSC. This is a zipped .fasta file of the entire mouse genome. UCSC also has genomes of a ton of other species, so make sure you're using the right one for your experiment.
The hisat2 software package contains a number of toolss for working with sequencing data, including one for aligning sequencing reads to long reference sequences. We have to build the genome index with hisat2 before we can align to it. Indexing a genome is like indexing a book. If you're looking for something in a book it's easier to narrow down where it could be in a pre-built index rather than go through page by page. This saves a lot of time and memory in the long run. This only has to be performed once per genome.

```
#download the mm10 genome file
wget http://hgdownload.soe.ucsc.edu/goldenPath/mm10/bigZips/mm10.2bit
#convert it to the correct format
twoBitToFa mm10.2bit mm10.fa
#build genome index
hisat2-build mm10.fa mm10
```

3.4 Extracting Splice Sites

A mature RNA molecule will not include any introns. As a result, we want the software to ignore any introns in the genome when aligning the reads. To do this we have to tell the software where the splice sites are located in the mouse genome. We can get this information from the mouse .gtf file from Ensembl. This file details the features of the mouse genome, such as what genes are where, where the introns and exons are, etc. We will use this .gtf file later on to assign reads to specific genes. We have to download and reformat the .gtf file before pulling the splice sites. We use a python script that comes with hisat2 for extacting splice sites from the .gtf file.

```
#download .gtf file
wget ftp://ftp.ensembl.org/pub/release-98/gtf/mus_musculus/Mus_musculus.GRCm38.98.gtf.gz

#the original gtf file is missing 'chr' prefix for the chromosome column, which will cause an error with htseq-count later on
#add 'chr' prefix to the beginning of every line, except the header
awk '{ if($1 !~ /^#/){print "chr"$0} else{print $0} }'  Mus_musculus.GRCm38.98.gtf > Mus_musculus.GRCm38.98.corrected.gtf

python3 /usr/local/bin/hisat2-2.10.0/hisat2_extract_splice_sites.py \
    Mus_musculus.GRCm38.98.corrected.gtf > Mus_musculus.GRCm38.98.splicesites.txt
```

3.5 Aligning .fastq Files to Genome with hisat2

Next code chunk will loop through the .fastq files will align the reads to the genome. This for loop will iterate through each of the paired end 1 .fastq files made earlier and align both the paired end 1 and 2 files to the mouse genome. Aligning to the genome takes a long time for big mammalian genomes.
Hisat2 requires a few inputs that are indicated by the dashed characters below. The `-x` argument is the name of all your indexed genome files. The `-rna-strandness` argument tells the program which paired end maps to the coding strand and which paired end maps to the template strand. The `-known-splicesite-infile` is where we input our defined splice sites. The `-1` and `-2` arguments are for the paired end reads. The output of the bowtie2 command is piped ('|') directly to the next set of commands which sorts the file by genomic location and saves it as a BAM file.

```
for i in *PE1.fastq.gz
do
    name=$(echo $i | awk -F"/" '{print $NF}' | awk -F"_PE." '{print $1}')
    echo $name
    hisat2 -x mm10 --rna-strandness RF --known-splicesite-infile Mus_musculus.GRCm38.98.splicesites.txt \
        -1 $i -2 ${name}_PE2.fastq.gz | samtools view -bS - | samtools sort -n - -o $name.sorted.bam
done
```

3.6 Further Formatting

We will clean up the data for the next couple steps. The first set of commands loops through the BAM files to remove 'junk' reads, which are reads that align to things like 'ChrUn' and alternative builds of chromosomes. This code here includes the mitochondrial reads, which is NOT what I did in the original analysis. If you run this code, you will retain these reads and incorporate the mitchondrial genes in the downstream analysis. To remove the junk, we have to resort the BAM file by read name.

```
for i in *sorted.bam
do
 name=$(echo $i | awk -F"/" '{print $NF}' | awk -F".sorted." '{print $1}')
 echo $name
 samtools sort $i -o $i
 samtools index $i
 samtools view -bh $i chr1 chr2 chr3 chr4 chr5 chr6 chr7 chr8 chr9 chr10 \
            chr11 chr12 chr13 chr14 chr15 chr16 chr17 chr18 chr19 chr20 chr21 chr22 chrX chrY chrM > $name.noJunk.bam
done
```

Then we remove PCR duplicates, which we identify as reads that start and end at the exact same genomic coordinates.

```
for i in *.noJunk.bam
do
 name=$(echo $i | awk -F"/" '{print $NF}' | awk -F".noJunk." '{print $1}')
 echo $name
 echo sorting and removing dups
 samtools sort -n $i -o $name.noJunk.bam
        #fixmate requires the bam be sorted by name
 samtools fixmate -m $name.noJunk.bam $name.fixmate.bam
        #markdup requires the bam be sorted by position (default sorting method)
 samtools sort $name.fixmate.bam -o $name.fixmate.bam
 samtools markdup -rs $name.fixmate.bam $name.noDups.bam
 rm $name.fixmate.bam
done
```

3.7 Sort Reads Into Genes

We use htseq-count to sort our reads into genomic features i.e. genes. The options below include `-r` which tells the program the bam file is sorted by position, `-f` which tells the program the input is a BAM file, and `-stranded` tells the program the strand information of the file. Notice the .gtf file is included. The output of this command is a .txt file that contains the gene counts for that sample. We will get one .txt file per sample.

```
for i in *.noDups.bam
do
    name=$(echo $i | awk -F"/" '{print $NF}' | awk -F".noDups." '{print $1}')
    echo $name
    samtools view -bf 1 $i | htseq-count -r pos -f bam \
        --stranded=reverse - Mus_musculus.GRCm38.98.corrected.gtf > $name.gene.counts.txt
done
```

3.8 Quality Control

It is good to take a look at several quality control metrics of your data to make sure nothing is awry. The metrics we will consider are the total number of reads, the amount of 'junk' reads that align to uninformative regions of the genome, the proportion of PCR duplicates, and the proportion of reads that actually align to genes. This last metric acts as a proxy for signal to noise. To calculate these metrics, we need the read counts for various intermediate BAM files we generated while progressing through the pipeline. In the code above, mitchondrial reads are kept in the analysis. However, the plot below was generated from a data set that classified the mitochondrial reads as 'junk', leading to a larger proportion of 'junk' reads. The first step is to pull read counts for the different samples at different steps of the pipeline and save them into a tab-delimited .txt file that we can upload into R.

```
echo -e "name\tall.counts\twithout.junk\tunique.counts" >> qc_metrics.txt

for bam in *sorted.bam
do
    name=$(echo $bam | awk -F".sorted.bam" '{print $1}')
    echo $name
```

```
    ALL_COUNTS=`samtools view -c $name.sorted.bam`
    WITHOUT_JUNK=`samtools view -c $name.noJunk.bam`
    UNIQUE_COUNTS=`samtools view -c $name.noDups.bam`
    echo -e "${name}\t$ALL_COUNTS\t$WITHOUT_JUNK\t$UNIQUE_COUNTS" >> qc_metrics.txt
done
```

After creating the .txt file, we will upload the read count data into R to calculate and plot our QC metrics. We can calculate the total reads, proportion not 'junk', and the proportion not PCR duplicates from this file. To calculate our proportion of reads in features / genes we have to use the 'all.counts' object generated when we first loaded the full counts table into R. The output of the code below is four bar charts displaying the how the metrics vary across the samples (figure 1). The goal of this analysis is to look for outliers in any of the metrics. We will also cross reference these data with the PCA potentially draw conclusions about outlier observations. For example, perhaps one sample has a low signal to noise ratio and that explains why it clusters away from its replicates on the PCA.

```
library(lattice)

qc <- read.table('qc_metrics.txt',sep='\t',header=TRUE)

rownames(qc) = c('Ab Sham 1','Ab Sham 2','Ab Sham 3','Ab Sham 4',
                 'Ab TBI 1','Ab TBI 2','Ab TBI 3','Ab TBI 4',
                 'noAb Sham 1','noAb Sham 2','noAb Sham 3','noAb Sham 4',
                 'noAb TBI 1','noAb TBI 2','noAb TBI 3','noAb TBI 4')
qc = qc[,-1]

qc$prop.junk = 1 - (qc$without.junk / qc$all.counts)
qc$prop.dups = 1 - (qc$unique.counts / qc$without.junk)

load('all.counts.Rdata')
qc$prop.in.features = colSums(all.counts[1:55421,]) / colSums(all.counts)

#qc$order = 1:25

qc$colors = c(rep('dark grey',4),rep('orange',4),rep('light blue',4),rep('blue',4))

pdf('total.reads.pdf', width=10, height=4)
print(barchart(all.counts ~ rownames(qc),  data = qc,
               main = "Total Reads",
               xlab = "Sample",
               ylab = "Read Count",
               col = qc$colors,
               ylim = c(0,100000000),
               scales = list(x = list(rot = 60)),
               horiz = FALSE))
dev.off()

pdf('proportion.junk.pdf', width=10, height=4)
print(barchart(prop.junk ~ rownames(qc),  data = qc,
               main = "Proportion Junk",
               xlab = "Sample",
               ylab = "Proportion Junk",
               col = qc$colors,
               ylim = c(0,1),
               scales = list(x = list(rot = 60)),
               horiz = FALSE))
dev.off()

pdf('proportion.duplicates.pdf', width=10, height=4)
print(barchart(prop.dups ~ rownames(qc),  data = qc,
               main = "Proportion PCR Duplicates",
               xlab = "Sample",
               ylab = "Proportion PCR Duplicates",
               col = qc$colors,
               ylim = c(0,1),
               scales = list(x = list(rot = 60)),
               horiz = FALSE))
dev.off()

pdf('proportion.in.features.pdf', width=10, height=4)
print(barchart(prop.in.features ~ rownames(qc),  data = qc,
               main = "Proportion In Features",
               xlab = "Sample",
               ylab = "Proportion In Features",
               col = qc$colors,
               ylim = c(0,1),
               scales = list(x = list(rot = 60)),
               horiz = FALSE))
dev.off()
```

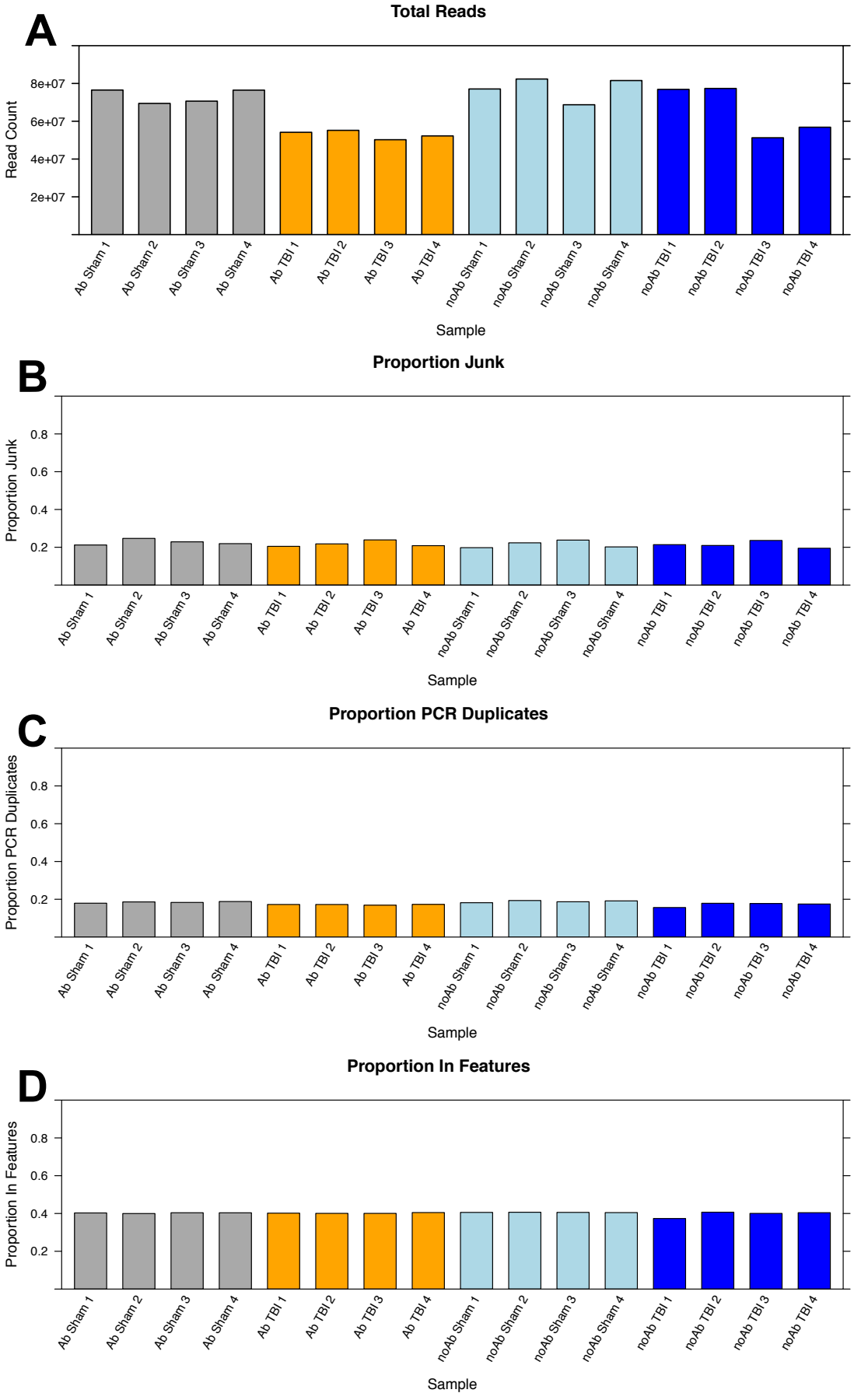

Figure 1: These four bar charts each present a different quality control metric for the experiment. (A) Total read counts show some expected variability, with the Ablation + TBI samples showing systemically low read depth. (B) Proportion of 'junk' reads are stable across the samples. 'Junk' is defined here as reads that align to 'ChrUn' or alternative builds of chromosomes. This plot includes mitochondrial reads as junk, although the code above will include them in the downstream analyses. (C) Proportion of PCR duplicates is also stable across the experiment. (D) The proportion of reads that align to genes also remains constant, indicating a stable signal to noise ratio.

4 Differential Expression Analysis in R

4.1 Prepare the Environment

We move into R to do the rest of the analysis. The first step is to load all the relevant software / functions into the R environment. This includes the software packages we installed earlier (lattice, DESeq2, pheatmap) as well as some functions we wrote and host on our lab github page. We also define some new functions here as well called 'categorize.deseq.df', 'run.deseq.list.any', and 'plotPCAlattice'. Lastly we move our working environment into the appropriate directory. One should run these steps every time they are working in R. We will use the 'categorize.deseq.df' function to classify our genes into 'activated', 'repressed', and 'not different' categories. As written, the function defines signicances as values with a adjusted p-value of less than 0.1 by default. The user can change the threshold when calling the function. In general, it is more prudent to use adjusted p-values as a cutoff threshold when doing a large number of significance tests to limit the number of false positives. Here is a good explanation: `http://www.nonlinear.com/support/progenesis/comet/faq/v2.0/pq-values.aspx`.

```
library(lattice)
library(DESeq2)
library(pheatmap)
library(GSA)
library(seq2pathway)

source('https://raw.githubusercontent.com/guertinlab/seqOutBias/master/docs/R/seqOutBias_functions.R')
source('https://raw.githubusercontent.com/mjg54/znf143_pro_seq_analysis/master/docs/ZNF143_functions.R')

categorize.deseq.df <- function(df, thresh = 0.1, log2fold = 0.0, treat
= 'Auxin') {

     df.activated = data.frame(matrix(nrow = 0, ncol = 0))
     df.repressed = data.frame(matrix(nrow = 0, ncol = 0))

     if (nrow(df[df$padj < thresh & !is.na(df$padj) & df$log2FoldChange > log2fold,]) != 0) {
        df.activated = df[df$padj < thresh & !is.na(df$padj) & df$log2FoldChange > log2fold,]
        df.activated$response = paste(treat, 'Activated')
        }

     if (nrow(df[df$padj < thresh & !is.na(df$padj) & df$log2FoldChange < -log2fold,]) != 0) {
        df.repressed = df[df$padj < thresh & !is.na(df$padj) & df$log2FoldChange < -log2fold,]
        df.repressed$response = paste(treat, 'Repressed')
        }

    df.unchanged = df[df$padj > 0.5 & !is.na(df$padj) & abs(df$log2FoldChange) < 0.25,]
    df.unchanged$response = paste(treat, 'Unchanged')

    df.dregs = df[!(df$padj < thresh & !is.na(df$padj) & df$log2FoldChange > log2fold) &
                  !(df$padj < thresh & !is.na(df$padj) & df$log2FoldChange < -log2fold) &
                  !(df$padj > 0.5 & !is.na(df$padj) &
                  abs(df$log2FoldChange) < 0.25), ]
    df.dregs$response = paste(treat, 'All Other Genes')

    df.effects.lattice =
    rbind(df.activated,
          df.unchanged,
          df.repressed,
          df.dregs)
    df.effects.lattice$response = factor(df.effects.lattice$response)
    df.effects.lattice$response = relevel(df.effects.lattice$response, ref = paste(treat, 'Unchanged'))
    df.effects.lattice$response = relevel(df.effects.lattice$response, ref = paste(treat, 'All Other Genes'))
    return(df.effects.lattice)
}

plotPCAlattice <- function(df, file = 'PCA_lattice.pdf') {
  perVar = round(100 * attr(df, "percentVar"))
  df = data.frame(cbind(df, sapply(strsplit(as.character(df$name), '_rep'), '[', 1)))
  colnames(df) = c(colnames(df)[1:(ncol(df)-1)], 'unique_condition')
  print(df)
  #get colors and take away the hex transparency
  color.x = substring(rainbow(length(unique(df$unique_condition))), 1,7)

  df$color = NA
  df$alpha.x = NA
  df$alpha.y = NA
  df$colpal = NA

  for (i in 1:length(unique(df$unique_condition))) {

    df[df$unique_condition == unique(df$unique_condition)[[i]],]$color = color.x[i]
    #gives replicates for unique condition
    reps_col<- df[df$unique_condition == unique(df$unique_condition)[[i]],]
    #gives number of replicates in unique condition
```

```
    replicates.x = nrow(reps_col)
    alx <- rev(seq(0.2, 1, length.out = replicates.x))

    #count transparency(alx), convert alx to hex(aly), combain color and transparency(cp)
    for(rep in 1:replicates.x) {

      na <- reps_col[rep, ]$name
      df[df$name == na, ]$alpha.x = alx[rep]
      aly = as.hexmode(round(alx * 255))
      df[df$name == na, ]$alpha.y = aly[rep]
      cp = paste0(color.x[i], aly)
      df[df$name == na, ]$colpal = cp[rep]
      #print(df)
    }
  }
  colpal = df$colpal
  df$name = gsub('_', ' ', df$name)
  df$name <- factor(df$name, levels=df$name, order=TRUE)
  pdf(file, width=6, height=6, useDingbats=FALSE)
  print(xyplot(PC2 ~ PC1, groups = name, data=df,
               xlab = paste('PC1: ', perVar[1], '% variance', sep = ''),
               ylab = paste('PC2: ', perVar[2], '% variance', sep = ''),
               par.settings = list(superpose.symbol = list(pch = c(20), col=colpal)),
               pch = 20, cex = 1.7,
               auto.key = TRUE,
               col = colpal))
  dev.off()
}

setwd('RNAseq_analysis')
```

4.2 Create Read Count Data Frame

Next step is to load the reads into R so we can do the rest of the analysis. We combine all the sample data into one data frame, which is essentially a big table. This includes some reformatting of the data. At the end of this step we save the counts table as an .Rdata object, so we can load the finished data frame into future R sessions without repeating any of these steps.

```
#first we initialize the data frame by defining the row names (i.e. all the different genes)
#we import one gene counts file - doesn't matter which one because they all use the same row names
x <- read.table("Ab_Sham_rep1.gene.counts.txt",header=FALSE)
#then we define a new data frame called 'all.counts' that is only made up of the row names from the counts file
#in other words, this is a data frame with just the genes without any sample data
all.counts <- data.frame(row.names = x$V1)

#define a vector with all the sample names
samples <- c("Ab_Sham_rep1","Ab_Sham_rep2","Ab_Sham_rep3","Ab_Sham_rep4",
             "Ab_TBI_rep1","Ab_TBI_rep2","Ab_TBI_rep3","Ab_TBI_rep4",
             "noAb_Sham_rep1","noAb_Sham_rep2","noAb_Sham_rep3","noAb_Sham_rep4",
             "noAb_TBI_rep1","noAb_TBI_rep2","noAb_TBI_rep3","noAb_TBI_rep4")

#loop through all the gene counts files add the read data to the 'all.counts' data frame
for (i in samples){
  j<-read.table(print(paste0(i,".gene.counts.txt")))
  all.counts <- cbind(all.counts,data.frame(i = j[2]))
}

#use the sample name vector to name the columns of the data frame
colnames(all.counts) <- samples

#save 'all.counts' object for QC later
save(all.counts,file='all.counts.Rdata')

#last five lines of count data are not in genes
#we get rid of them and save the data frame as a new object called 'merged.counts'
merged.counts <- all.counts[-(55422:55426),]

#the row names of the data frame are Ensembl IDs rather than gene names. We have to replace them

#generating Gene ID alongside EnsembleID. We'll remove these data frames when we're finished because they are very large
ensembl.all = read.table('Mus_musculus.GRCm38.98.corrected.gtf', sep='\t', header =F);
ensembl.gene.names = data.frame(sapply(strsplit(sapply(strsplit(
    as.character(ensembl.all[,9]),'gene_name '), "[", 2), ";"), "[", 1));
ensembl.id.names = data.frame(sapply(strsplit(sapply(strsplit(
    as.character(ensembl.all[,9]),'gene_id '), "[", 2), ";"), "[", 1));
ensembl.code = cbind(ensembl.gene.names, ensembl.id.names);
ensembl.code = ensembl.code[!duplicated(ensembl.code[,2]),];
rownames(ensembl.code) = ensembl.code[,2];
colnames(ensembl.code) = c('gene', 'id');

#data frame with the Ensembl IDs and their corresponding gene names
```

```
save(ensembl.code, file = "ensembl.code.Rdata")

#remove data frames that are no longer needed
rm(ensembl.all)
rm(ensembl.gene.names)
rm(ensembl.id.names)

#replace Ensembl IDs with gene names in 'merged.counts' row names
merged.counts = merge(merged.counts, ensembl.code, by="row.names", all.x=F)
rownames(merged.counts) <- make.names(merged.counts$gene, unique=TRUE)
merged.counts <- merged.counts[,-c(1,18,19)]

rm(ensembl.code)

#save the final counts table to disk so we can load it any time
save(merged.counts, file="merged.counts.Rdata")
```

4.3 Principal Component Analysis

PCA is a dimensionality reduction technique that allows us to visualize how similar / different our samples are. By reducing all the variation in gene count information to two dimensions, we can easily evaluate clustering of different samples and identify batch effects and outliers. We use the 'plotPCA' function in the DESeq2 package to calculate our principal components. To do this we must convert our counts table to a DESeq object, which is a special type of data type that is specifically designed for use with the DESeq2 functions. Our 'plotPCAlattice' function (defined above) can be modified for different formatting options. Our PCA shows acceptable clustering (figure 2).

```
unt=4
trt=12
sample.conditions = factor(c(rep("untreated",unt), rep("treated",trt)), levels=c("untreated","treated"))

#convert counts table to a DESeq object
deseq.counts.table = DESeqDataSetFromMatrix(merged.counts, DataFrame(sample.conditions), ~ sample.conditions);
colData(deseq.counts.table)$condition<-factor(colData(deseq.counts.table)$sample.conditions, levels=c('untreated','treated'));
dds = DESeq(deseq.counts.table)
#take a log2 transformation of all the read counts
rld_HH = rlogTransformation(dds)
#calculate principal components
pca.dat = plotPCA(rld_HH, intgroup="condition", returnData=TRUE)
percentVar = round(100 * attr(pca.dat, "percentVar"))
#plot PCA
plotPCAlattice(pca.dat, file = 'PCA_RNAseq.pdf')
```

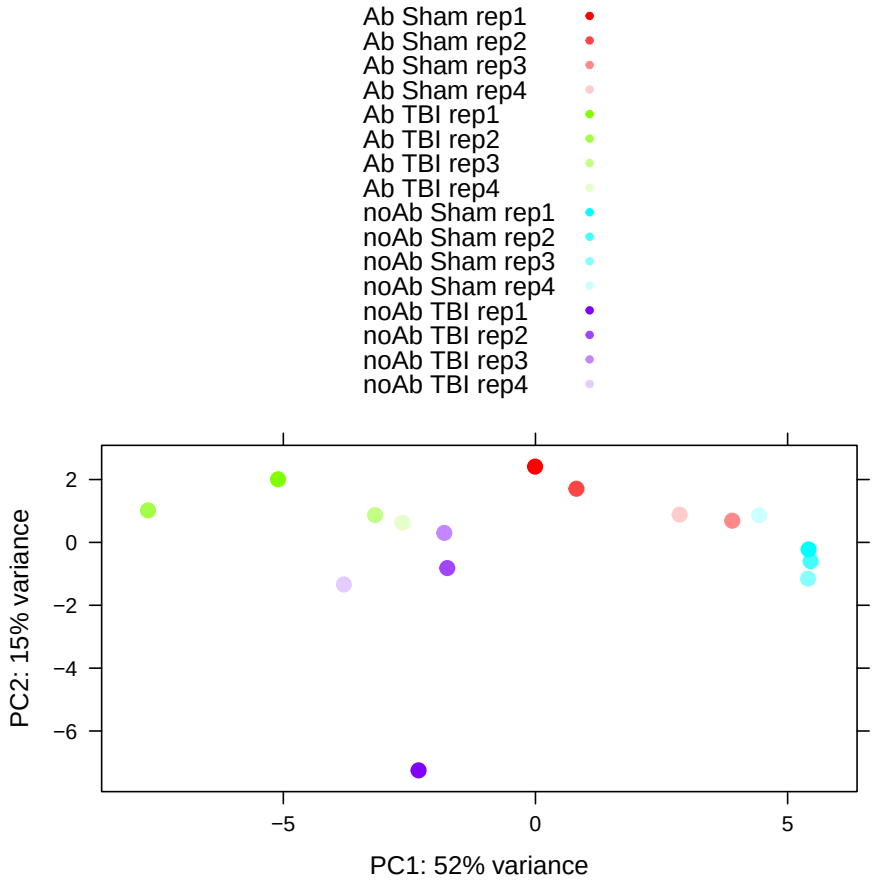

Figure 2: The PCA plot shows how similar your samples are to each other. Ideally all the data points from experimental replicates should cluster together. Looks like the TBI samples are clustering to the right on the PC1 axis and the sham samples are clustering to the left. Note that Principal Component 1 explains 52% of the variance.

4.4 Differential Expression Analysis

DESeq2 is one of several differential expression analysis software packages designed for high-throughput sequencing data. We are separating DESeq2 into individual function calls in case we want to change some of the parameters such as size factors, but overall we are following the generic DESeq2 analysis pipeline. This code chunk goes through four comparisons: TBI with and without ablation, ablation with and without TBI, no ablation with and without TBI, and no TBI with and without ablation. Notice that the only differences between the code for the different comparisons are what columns are pulled from the counts table and the names of the objects / plots. The output of this code for each comparison is an MA plot that shows how all the genes are responding to TBI / ablation and a saved .Rdata object that contains the results of the analysis. The MA plot for the TBI with ablation versus without ablation shows modest changes in global gene expression with 55 genes significantly increasing expression and 15 genes significnatly decreasing expression (Figure 3B). Comparing samples with and without TBI show a much greater extent of gene expression changes (Figure 3C-D).

```
###### noAb_TBI v. Ab_TBI ######

#define a smaller data frame composed of just the samples involved in this comparison
#the first four columns represent the control, the second four represent the treatment
merged.counts.small = merged.counts[,c(13:16,5:8)]

#establish the number of replicates per condition
unt = 4
trt = 4

#the following commands create a DESeq object, calculate average expression and fold change between different conditions,
#and run t-tests for each gene
sample.conditions = factor(c(rep("untreated",unt), rep("treated",trt)), levels=c("untreated","treated"))
mm.deseq.counts.table = DESeqDataSetFromMatrix(merged.counts.small, DataFrame(sample.conditions), ~ sample.conditions)

mm.atac = mm.deseq.counts.table
atac.size.factors = estimateSizeFactorsForMatrix(merged.counts.small)

sizeFactors(mm.atac) = atac.size.factors
mm.atac = estimateDispersions(mm.atac)
mm.atac = nbinomWaldTest(mm.atac)
res.mm.atac = results(mm.atac)

#our 'categorize.deseq.df' function adds a column to the results data frame that indicates whether
#a gene is activated, repressed, or unchanged and saves it as a lattice object
#this is also where we define the threshold for the comparison.
#If you want a more or less stringent threshold, then change it here.
noAb.tbi.v.ab.tbi.lattice =
    categorize.deseq.df(res.mm.atac,
                        thresh = 0.1, log2fold = 0.0, treat = 'Ablation')

#generate MA plot showing how all 55421 genes respond to either TBI or ablation, depending on the comparison
pdf("noAb.tbi.v.ab.tbi.MAplot.pdf", useDingbats = FALSE, width=4, height=3.33);
print(xyplot(noAb.tbi.v.ab.tbi.lattice$log2FoldChange ~
                log(noAb.tbi.v.ab.tbi.lattice$baseMean, base=10),
             groups=noAb.tbi.v.ab.tbi.lattice$response,
             col=c("grey90", "grey60", "red", "blue"),
             scales="free",
             aspect=1,
             ylim=c(-6.5, 6.5),
#             xlim=c(-1,4.2),
             par.strip.text=list(cex=1.0, font = 1),
             pch=20,
             cex=0.5,
             ylab=expression("+Ablation log"[2]~"Expression fold change"),
             xlab=expression("log"[10]~"Mean of Normalized Counts"),
             main = 'noAb_TBI v. Ab_TBI',
            par.settings=list(par.xlab.text=list(cex=1.1,font=2),
                              par.ylab.text=list(cex=1.1,font=2),
                              strip.background=list(col="grey85"))))
dev.off()

#save the lattice object so that DESeq2 does not need to be run again
save(noAb.tbi.v.ab.tbi.lattice, file='noAb.tbi.v.ab.tbi.lattice.Rdata')

###### Ab_Sham v. Ab_TBI ######

merged.counts.small = merged.counts[,1:8]

# number of replicates per condition
unt = 4
trt = 4

sample.conditions = factor(c(rep("untreated",unt), rep("treated",trt)), levels=c("untreated","treated"))
mm.deseq.counts.table = DESeqDataSetFromMatrix(merged.counts.small, DataFrame(sample.conditions), ~ sample.conditions)
```

```
mm.atac = mm.deseq.counts.table
atac.size.factors = estimateSizeFactorsForMatrix(merged.counts.small)

sizeFactors(mm.atac) = atac.size.factors
mm.atac = estimateDispersions(mm.atac)
mm.atac = nbinomWaldTest(mm.atac)
res.mm.atac = results(mm.atac)

ab.sham.v.ab.tbi.lattice =
    categorize.deseq.df(res.mm.atac,
                        thresh = 0.1, log2fold = 0.0, treat = 'TBI')

pdf("ab.sham.v.ab.tbi.MAplot.pdf", useDingbats = FALSE, width=4, height=3.33);
print(xyplot(ab.sham.v.ab.tbi.lattice$log2FoldChange ~
                log(ab.sham.v.ab.tbi.lattice$baseMean, base=10),
             groups=ab.sham.v.ab.tbi.lattice$response,
             col=c("grey90", "grey60", "red", "blue"),
             scales="free",
             aspect=1,
             ylim=c(-6.5, 6.5),
#             xlim=c(-1,4.2),
             par.strip.text=list(cex=1.0, font = 1),
             pch=20,
             cex=0.5,
             ylab=expression("+TBI log"[2]~"Expression fold change"),
             xlab=expression("log"[10]~"Mean of Normalized Counts"),
             main = 'Ab_Sham v. Ab_TBI',
            par.settings=list(par.xlab.text=list(cex=1.1,font=2),
                              par.ylab.text=list(cex=1.1,font=2),
                              strip.background=list(col="grey85"))))
dev.off()

save(ab.sham.v.ab.tbi.lattice, file='ab.sham.v.ab.tbi.lattice.Rdata')

###### noAb_Sham v. noAb_TBI ######

merged.counts.small = merged.counts[,c(9:12,13:16)]

# number of replicates per condition
unt = 4
trt = 4

sample.conditions = factor(c(rep("untreated",unt), rep("treated",trt)), levels=c("untreated","treated"))
mm.deseq.counts.table = DESeqDataSetFromMatrix(merged.counts.small, DataFrame(sample.conditions), ~ sample.conditions)

mm.atac = mm.deseq.counts.table
atac.size.factors = estimateSizeFactorsForMatrix(merged.counts.small)

sizeFactors(mm.atac) = atac.size.factors
mm.atac = estimateDispersions(mm.atac)
mm.atac = nbinomWaldTest(mm.atac)
res.mm.atac = results(mm.atac)

noAb.sham.v.noAb.tbi.lattice =
    categorize.deseq.df(res.mm.atac,
                        thresh = 0.1, log2fold = 0.0, treat = 'TBI')

pdf("noAb.sham.v.noAb.tbi.MAplot.pdf", useDingbats = FALSE, width=4, height=3.33);
print(xyplot(noAb.sham.v.noAb.tbi.lattice$log2FoldChange ~
                log(noAb.sham.v.noAb.tbi.lattice$baseMean, base=10),
             groups=noAb.sham.v.noAb.tbi.lattice$response,
             col=c("grey90", "grey60", "red", "blue"),
             scales="free",
             aspect=1,
             ylim=c(-6.5, 6.5),
#             xlim=c(-1,4.2),
             par.strip.text=list(cex=1.0, font = 1),
             pch=20,
             cex=0.5,
             ylab=expression("+TBI log"[2]~"Expression fold change"),
             xlab=expression("log"[10]~"Mean of Normalized Counts"),
             main = 'noAb_sham v. noAb_TBI',
            par.settings=list(par.xlab.text=list(cex=1.1,font=2),
                              par.ylab.text=list(cex=1.1,font=2),
                              strip.background=list(col="grey85"))))
dev.off()

save(noAb.sham.v.noAb.tbi.lattice, file='noAb.sham.v.noAb.tbi.lattice.Rdata')

###### noAb_Sham v. Ab_Sham ######

merged.counts.small = merged.counts[,c(9:12,1:4)]

# number of replicates per condition
unt = 4
trt = 4
```

```
sample.conditions = factor(c(rep("untreated",unt), rep("treated",trt)), levels=c("untreated","treated"))
mm.deseq.counts.table = DESeqDataSetFromMatrix(merged.counts.small, DataFrame(sample.conditions), ~ sample.conditions)

mm.atac = mm.deseq.counts.table
atac.size.factors = estimateSizeFactorsForMatrix(merged.counts.small)

sizeFactors(mm.atac) = atac.size.factors
mm.atac = estimateDispersions(mm.atac)
mm.atac = nbinomWaldTest(mm.atac)
res.mm.atac = results(mm.atac)

noAb.sham.v.Ab.sham.lattice =
    categorize.deseq.df(res.mm.atac,
                        thresh = 0.1, log2fold = 0.0, treat = 'Ablation')

pdf("noAb.sham.v.Ab.sham.MAplot.pdf", useDingbats = FALSE, width=4, height=3.33);
print(xyplot(noAb.sham.v.Ab.sham.lattice$log2FoldChange ~
                log(noAb.sham.v.Ab.sham.lattice$baseMean, base=10),
             groups=noAb.sham.v.Ab.sham.lattice$response,
             col=c("grey90", "grey60", "red", "blue"),
             scales="free",
             aspect=1,
             ylim=c(-6.5, 6.5),
#             xlim=c(-1,4.2),
             par.strip.text=list(cex=1.0, font = 1),
             pch=20,
             cex=0.5,
             ylab=expression("+Ablation log"[2]~"Expression fold change"),
             xlab=expression("log"[10]~"Mean of Normalized Counts"),
             main = 'noAb_sham v. Ab_sham',
            par.settings=list(par.xlab.text=list(cex=1.1,font=2),
                              par.ylab.text=list(cex=1.1,font=2),
                              strip.background=list(col="grey85"))))
dev.off()

save(noAb.sham.v.Ab.sham.lattice, file='noAb.sham.v.Ab.sham.lattice.Rdata')
```

To determine how many activated and repressed genes there are for each comparison, simply run the following line of code on the lattice object for the comparison you are interested in.

```
print(table(noAb.tbi.v.ab.tbi.lattice$response))
```

For GO term analysis, we save lists of the significantly increased and decreased genes in the TBI plus and minus ablation comparison called '0.1.padj.inc.txt' and '0.1.padj.dec.txt' respectively. We input these into the Panther classification system (`http://www.pantherdb.org`).

```
#copy the appropriate lattice object
lattice = noAb.tbi.v.ab.tbi.lattice

write(rownames(lattice[lattice$response == 'Ablation Activated',]),file='0.1.padj.inc.txt',sep='\t')
write(rownames(lattice[lattice$response == 'Ablation Repressed',]),file='0.1.padj.dec.txt',sep='\t')
```

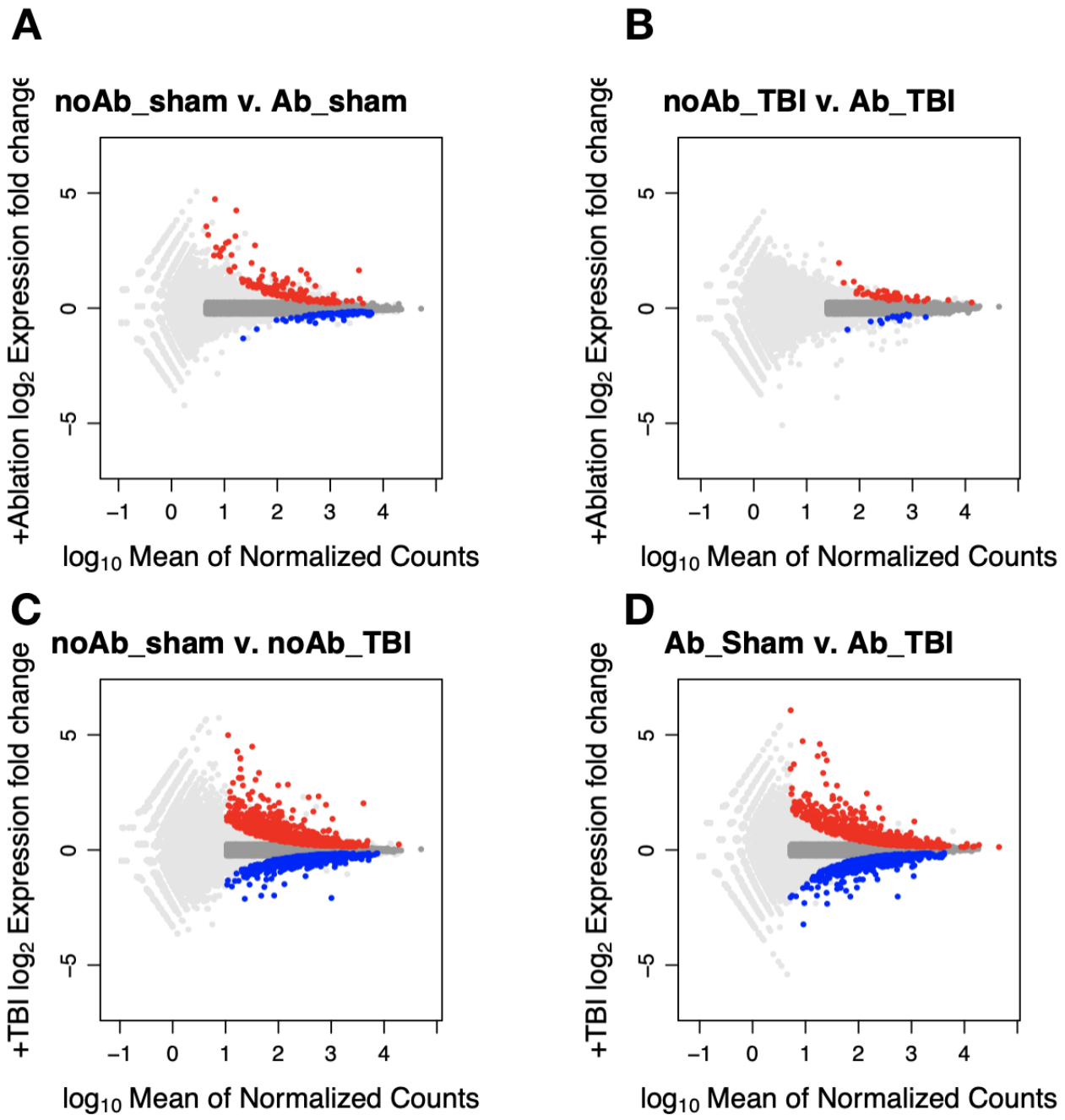

Figure 3: The x-axis of an MA plot represents mean expression of the gene across all samples, while the y-axis represents the log2 transformation of the fold change between the conditions. Genes that have a adjusted p-value below the threshold of 0.1 are marked as either 'activated' or 'repressed', depending on the direction of expression change, and colored red or blue respectively. The dark grey genes represent the 'not different' group, which are genes that have adjusted p-values greater than 0.5 and a log2 fold change of less than 0.25 in either direction. This control set represents genes that are expressed but not significantly affected by ablation. We specially define this control set to differentiate them from genes that may be insignificantly affected but are barely expressed (light grey genes). (A) No ablation sham treatment v. ablation sham treatment. This comparison returns 158 activated genes and 87 repressed genes. (B) No ablation TBI treatment v. ablation TBI treatment. This comparison returns 55 activated genes and 15 repressed genes. (C) No ablation sham treatment v. no ablation TBI treatment. This comparison returns 969 activated genes and 842 repressed genes. (D) Ablation sham treatment v. ablation TBI treatment. This comparison returns 897 activated genes and 813 repressed genes.

4.5 Gene Set Enrichment Analysis (GSEA)

We apply GSEA to study enrichment of our differentially expressed genes in immune-related gene sets. For GSEA, we use a more lenient threshold for differential expression - a p-value of 0.05. These annotated gene sets come from the Molecular Signatures Database (`http://software.broadinstitute.org/gsea/msigdb/index.jsp`) which contains eight major repositories of gene sets. The first step is to download the database as a .gmt file.

```
wget http://software.broadinstitute.org/gsea/msigdb/download_file.jsp?filePath=/resources/msigdb/7.0/msigdb.v7.0.symbols.gmt
```

We then import the data into R using a program from the GSA software package. This converts the .gmt file into a unique R object that can be used with GSEA software. For the enrichment analysis we use a program from the seq2pathway package, which performs Fisher's exact tests for each of the gene sets to evaluate enrichment. The output of this program is a large data frame that contains all the enrichment information. The software will determine enrichment of the gene set for thousands of pathways. We export the data for all these pathways but also choose a small subset of immune-related gene sets of particular interest to us. For this subset, we use the adjusted p-value from the Fisher's test to evaluate enrichment of our genes in that gene set (Figure 4). This high degree of enrichment suggests that these individual gene sets would be good candidates for heat map generation.

```
#load in MSig database using the 'GSA.read.gmt' command' and save it as an R object
msigdb = GSA.read.gmt('msigdb.v7.0.symbols.gmt')

#pull all genes with a p-value less than 0.05 and a log2FoldChange greater than 0
lattice = na.omit(noAb.tbi.v.ab.tbi.lattice)
lattice = lattice[lattice$pvalue < 0.05 & lattice$log2FoldChange > 0,]

#use the 'FisherTest_MsigDB' function to run the GSEA
inc.pval.0.05.gsea = FisherTest_MsigDB(msigdb,toupper(rownames(lattice)),'mm10')

#save the enriched gene sets and their associated adjusted p-values to a .txt file
x = data.frame(GeneSet = inc.pval.0.05.gsea$GeneSet, padj = inc.pval.0.05.gsea$FDR)
write.table(x,file='inc.gene.sets.pval.0.05.txt',sep='\t')

#repeat for the decreased genes
lattice = na.omit(noAb.tbi.v.ab.tbi.lattice)
lattice = lattice[lattice$pvalue < 0.05 & lattice$log2FoldChange < 0,]

dec.pval.0.05.gsea = FisherTest_MsigDB(msigdb,toupper(rownames(lattice)),'mm10')

x = data.frame(GeneSet = dec.pval.0.05.gsea$GeneSet, padj = dec.pval.0.05.gsea$FDR)
write.table(x,file='dec.gene.sets.pval.0.05.txt',sep='\t')

#save gene sets of interest to a 'gene.sets' vector
gene.sets = c("GO_REGULATION_OF_IMMUNE_SYSTEM_PROCESS","GO_POSITIVE_REGULATION_OF_IMMUNE_SYSTEM_PROCESS",
                    "GO_REGULATION_OF_IMMUNE_RESPONSE","GO_INNATE_IMMUNE_RESPONSE",
                    "GO_REGULATION_OF_IMMUNE_EFFECTOR_PROCESS","GO_IMMUNE_EFFECTOR_PROCESS",
                    "GO_REGULATION_OF_INNATE_IMMUNE_RESPONSE","REACTOME_INNATE_IMMUNE_SYSTEM",
                    "HALLMARK_COMPLEMENT","GO_CELL_ACTIVATION_INVOLVED_IN_IMMUNE_RESPONSE")

#pull corresponding rows from GSEA output
row.nums = c()
for (i in gene.sets) {
   row.nums = append(row.nums,grep(i,inc.pval.0.05.gsea$GeneSet))
}

#reorder based on adjusted p-value and plot
x = inc.pval.0.05.gsea[row.nums,]
x = x[order(x$FDR),]
x$order = 10:1

pdf(file = 'inc.gsea.0.05.pval.pdf',width=9)
print(barchart((-log2(FDR)) ~ reorder(GeneSet,order), data = x,
              main = "Enriched Gene Sets in Increased Genes (0.05 p-val)",
              xlab = "Gene Set",
              ylab = "-log2(padj)",
              col = 'red',
              #ylim = c(0,1),
              scales = list(x = list(rot = 60)),
              horiz = FALSE))
dev.off()
```

4.6 Heat Maps

Heat maps are an effective way of communicating how genes respond to perturbation across individual samples. The first heat map is the top 20 most significantly increased and the 20 most significantly

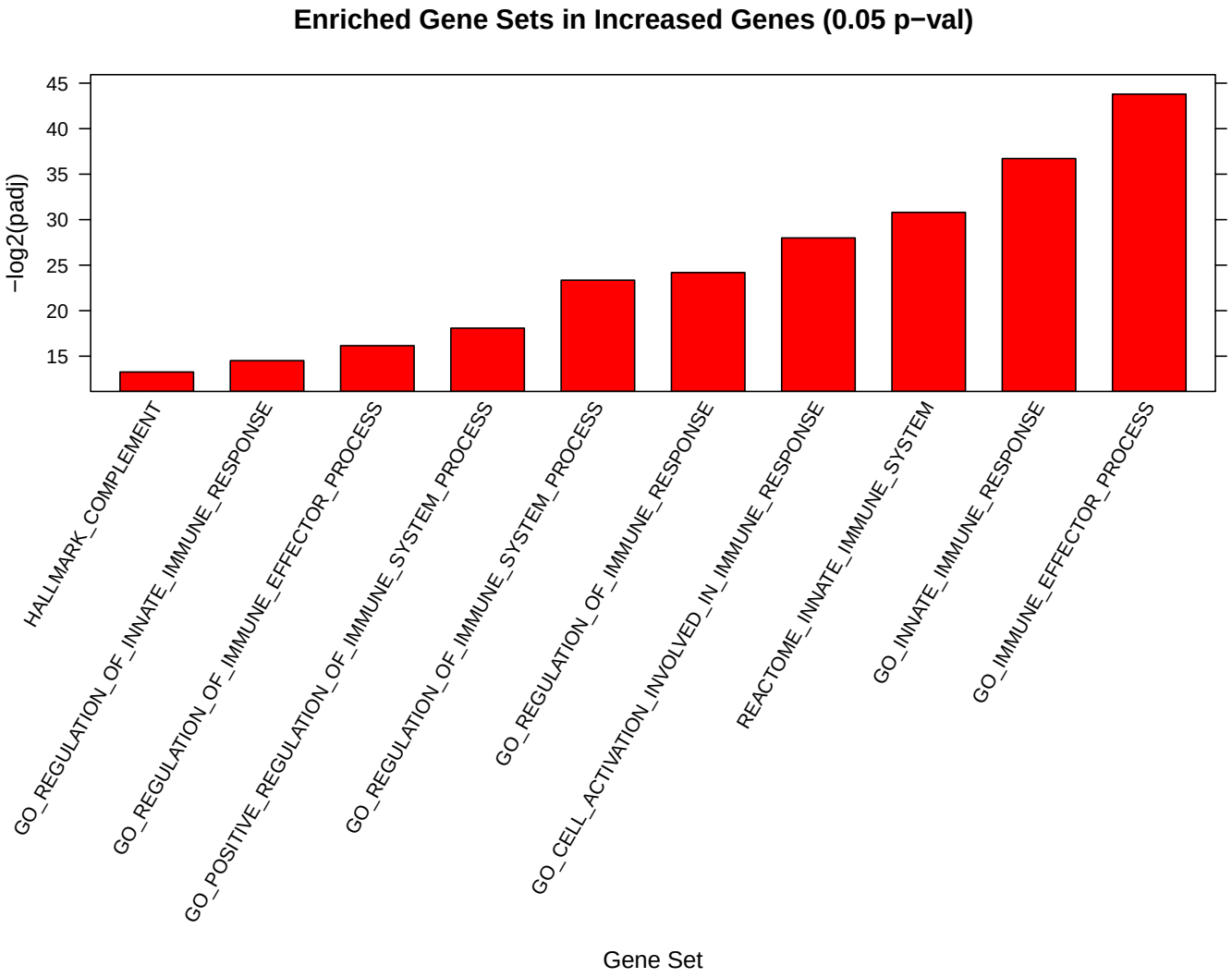

Figure 4: The x-axis represents different MSigDB pathways and their original source (ex. GO, Reactome, Broad Hallmark). The y-axis shows the negative log2 transformation of the adjusted p-value for each of the enrichments. The input for the GSEA are activated genes with a p-value of less than 0.05.

decreased genes following ablation in a TBI context (figure 5). We defined the 40 most significantly changed genes above, so now we just have to pull the relevant data from our counts table. The authors of DESeq2 recommend the following approach to generating heat maps from counts data: first perform a log2 transformation of the normalized counts data to minimize differences between highly expressed genes and lowly expressed genes. Then center the data by subtracting the mean expression for each gene from all the corresponding values, so that we we're really looking at how expression deviates from the average expression for each gene (`https://bioconductor.org/packages/release/workflows/vignettes/rnaseqGene/inst/doc/rnaseqGene.html`).

```
#make DEseq object in the same way as we did for PCA
unt=4
trt=4
sample.conditions = factor(c(rep("untreated",unt), rep("treated",trt)), levels=c("untreated","treated"))

deseq.counts.table = DESeqDataSetFromMatrix(merged.counts[,c(13:16,5:8)], DataFrame(sample.conditions), ~ sample.conditions);
colData(deseq.counts.table)$condition<-factor(colData(deseq.counts.table)$sample.conditions, levels=c('untreated','treated'));
dds = DESeq(deseq.counts.table)
#perform the log2 transformation
rld_HH = rlogTransformation(dds)

lattice = noAb.tbi.v.ab.tbi.lattice
lattice = na.omit(lattice)
```

```
#define a new object called 'y' that contains all genes that increase expression and sort y by adjusted p values
y = lattice[lattice$log2FoldChange > 0,]
y = y[order(y$padj),]
#the 'inc' object are the 20 genes with the lowest adjusted p value that increase expression
inc = rownames(y)[1:20]

#repeat for genes that decrease expression
y = lattice[lattice$log2FoldChange < 0,]
y = y[order(y$padj),]
dec = rownames(y)[1:20]

#define new object 'a' that contains the transformed, normalized read counts for the
#top 20 most significantly increased and decreased genes in just the noAb_TBI and the Ab_TBI samples
a = assay(rld_HH)[c(inc[1:20],dec[1:20]),]
#subtract average expression from each gene
a = a - rowMeans(a)

#produce heatmap
#the 'pheatmap' function has many options for formatting, including clustering of rows (genes)
#and/or columns (samples). Just change 'FALSE' to 'TRUE' to turn on these options
pdf(file='Ab.v.noAb.top20.heatmap.pdf')
print(pheatmap(a, cluster_rows=FALSE, show_rownames=TRUE,cluster_cols=FALSE))
dev.off()
```

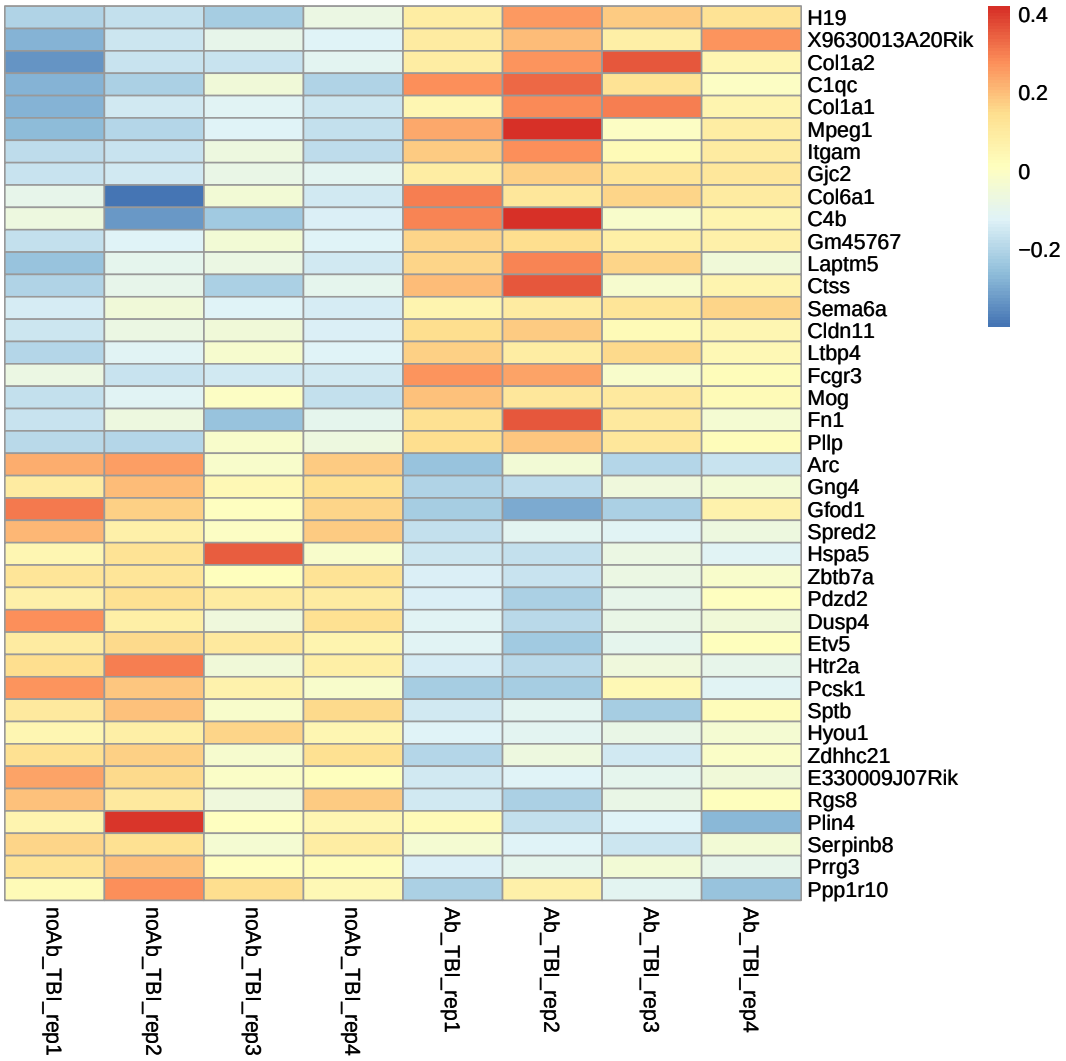

Figure 5: The 20 most significantly increased and the 20 most significantly decreased genes upon ablation in a TBI context are shown here. Significance was determined by adjusted p-value. The first 20 rows represent increased genes and the bottom 20 rows represent decreased genes. We performed a log2 transformation on the read counts for these genes. Then we took the average expression of each gene and subtracted that value from the respective counts to get the numbers represented here.

The next two heat maps examine how the Broad Hallmark sets of complement-related genes (figure 6) and GO's immune effector genes (figure 7) respond to ablation in a TBI context. Getting these gene lists requires some formatting on the command line and in R.

```
#download Hallmark complement gene set and rename it 'complement_hallmark.txt'
wget http://software.broadinstitute.org/gsea/msigdb/download_geneset.jsp?geneSetName=HALLMARK_COMPLEMENT
mv download_geneset.jsp?geneSetName=HALLMARK_COMPLEMENT complement_hallmark.txt

#download GO immune effectors gene set and rename it 'immune_effectors_GO.txt'
wget http://software.broadinstitute.org/gsea/msigdb/download_geneset.jsp?geneSetName=GO_IMMUNE_EFFECTOR_PROCESS
mv download_geneset.jsp?geneSetName=GO_IMMUNE_EFFECTOR_PROCESS immune_effectors_GO.txt
```

```
#complement gene set
#load complement gene set into R and get rid of the first two lines (unnecessary information)
x = read.table('complement_hallmark.txt',sep='\t')
x = as.vector(x[3:nrow(x),1])

#pull the row numbers corresponding to complement genes and put them in a list object called 'genes'
genes = c()
for (i in x) {
    genes = append(genes,grep(paste0('^',i,'$'),toupper(rownames(noAb.tbi.v.ab.tbi.lattice))))
}

#pull the appropriate rows from the lattice object and reorder them by padj
lattice = noAb.tbi.v.ab.tbi.lattice[genes,]
lattice = lattice[order(lattice$padj),]

#define object 'a' as the normalized, log2-transformed read counts of just the genes and samples of interest
a = assay(rld_HH)[rownames(lattice)[1:20],]
#center data by subtracting mean of each row
a = a - rowMeans(a)

#plot and save heat map
pdf(file='complement.top20.heatmap.pdf')
print(pheatmap(a, cluster_rows=FALSE, show_rownames=TRUE,cluster_cols=FALSE))
dev.off()

#repeat with immune effectors (GO)
x = read.table('immune_effectors_GO.txt',sep='\t')
x = as.vector(x[3:nrow(x),1])

genes = c()
for (i in x) {
    genes = append(genes,grep(paste0('^',toupper(i),'$'),toupper(rownames(noAb.tbi.v.ab.tbi.lattice))))
}

lattice = noAb.tbi.v.ab.tbi.lattice[genes,]
lattice = lattice[order(lattice$padj),]

a = assay(rld_HH)[rownames(lattice),]
a = a - rowMeans(a)
a = a[1:40,]

pdf(file='immune.effectors.GO.heatmap.pdf')
print(pheatmap(a, cluster_rows=FALSE, show_rownames=TRUE,cluster_cols=FALSE))
dev.off()
```

4.7 Volcano Plots

Volcano plots place log2FoldChange on the x-axis and -log10(padj) on the y-axis, so genes that are further toward the extremes on both axes are usually the most significantly differentially expressed. Our cutoff for significance here as above is a adjusted p-value of 0.1. The y-axis represents a -log10 transformation of the adjusted p-values, leading to a threshold of roughtly 1.3. The dashed horizontal grey line represents this threshold. Here we have volcano plots showing the response following TBI without ablation (figure 8A) and TBI with ablation (figure 8B), as well as the plot for ablation versus no ablation in TBI (figure 8C). The final volcano plot also shows the ablation versus no ablation comparison but includes six genes highlighted in green (figure 8D). The only one on the decreased side is Arc. The other five, from highest on the y-axis to lowest are as follows: C1qc, Itgam, C4b, Fn1, and C1qa. One can highlight whatever genes one chooses by changing the 'genes' vector in the code below.

```
volcano.plot <- function(lattice, thresh = 0.1, treat='Ablation', title="", highlights = c()) {
    name = strsplit(deparse(substitute(lattice)),'.lattice')[[1]][1]
```

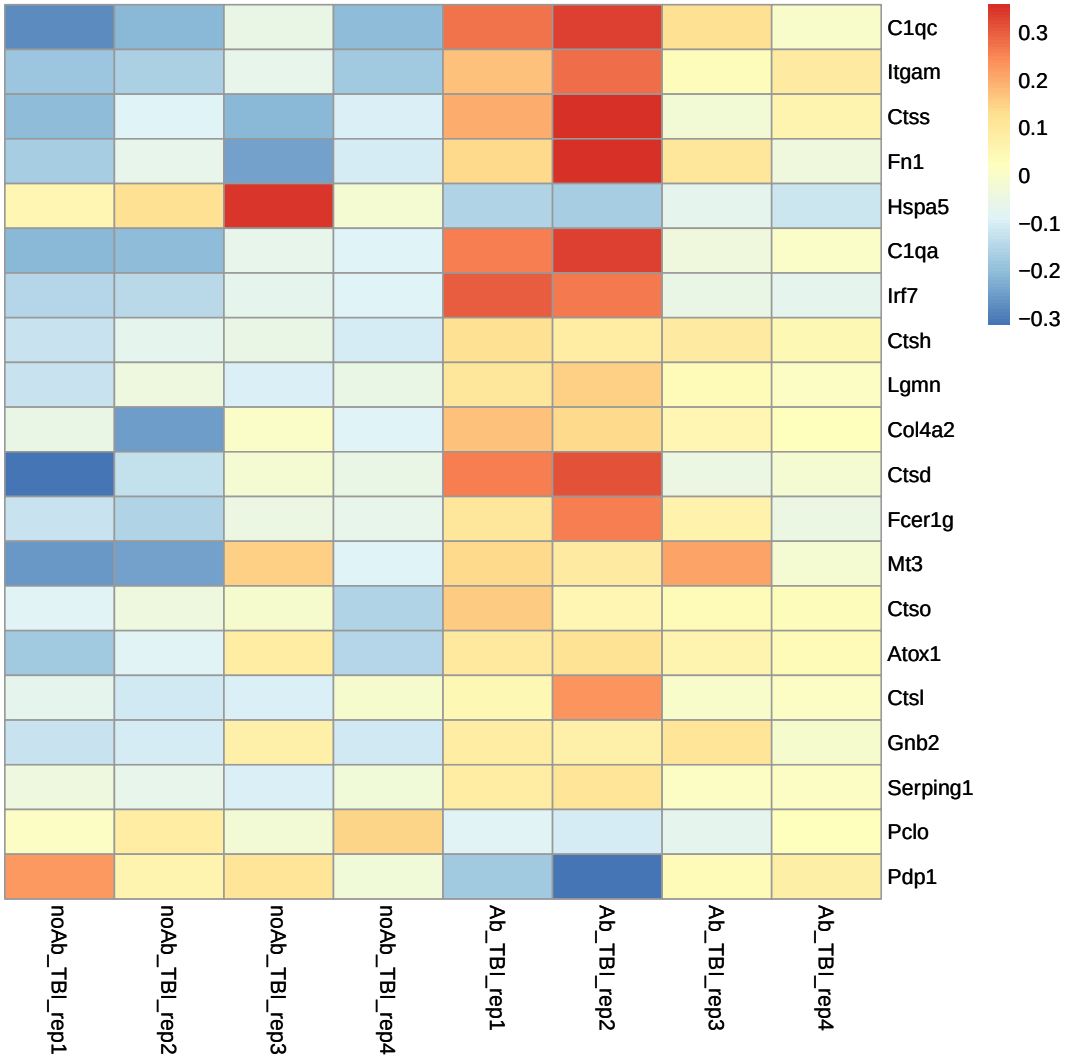

Figure 6: This heat map represents the 20 genes in the Broad Institute's Hallmark Complement gene set that are most significantly changed (either up or down) following ablation in a TBI context. The values were calculated using the same approach as described in the text.

```
#add a 'color' column that will dictate data point color
lattice$color = 'black'
lattice[lattice$response == paste0(treat,' Activated'),]$color = 'red'
lattice[lattice$response == paste0(treat,' Repressed'),]$color = 'blue'

#if a vector of 'highlight' genes is inputted, then iterate through vector
#and change 'color' value of those genes to 'green'
if (length(highlights) > 0) {
    name = paste0(name,'.highlights')
    #paste '^' and '$' onto gene name to ensure exact match
    highlights = paste0('^',highlights,'$')
    row.nums = c()
    for(i in highlights) {
        row.nums = append(row.nums, grep(i,rownames(lattice)))
    }
    lattice[row.nums,]$color = 'green'
}

pdf(file = paste0(name,'.volcano.plot.pdf'))

#plot the log2FoldChange on the x-axis and the -log10(padj) on the y-axis
plot(lattice$log2FoldChange, -log10(lattice$padj), xlim = c(-5,6), col = lattice$color, cex = 0.8, pch = 20,
 main = title, xlab = 'log2FoldChange', ylab = '-log10(padj)')
#add a line to signify the significance threshold
abline(h=-log(thresh,base=10), lty = 5, lwd = 4, col = '#C0C0C0')
```

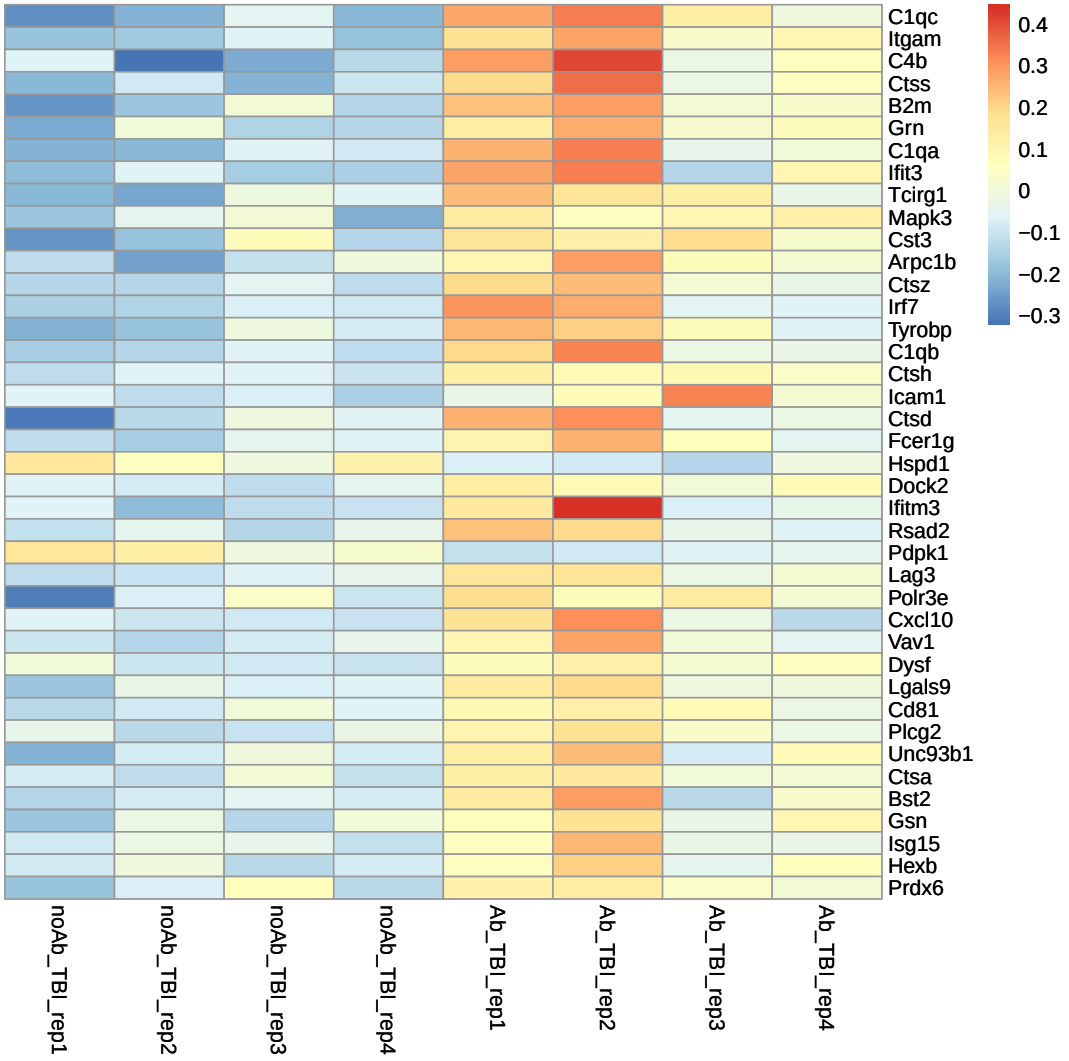

Figure 7: This heat map represents the 40 genes in GO's Immune Effector gene set that are most significantly changed (either up or down) following ablation in a TBI context. The values were calculated using the same approach as described in the text.

```
    dev.off()
}

volcano.plot(noAb.tbi.v.ab.tbi.lattice, treat='Ablation', title='TBI No Ablation v. Ablation')
volcano.plot(ab.sham.v.ab.tbi.lattice, treat='TBI', title='Ablation Sham v. TBI')
volcano.plot(noAb.sham.v.noAb.tbi.lattice, treat ='TBI', title='No ablation Sham v. TBI')

x = c('Arc', 'C1qc', 'C1qa', 'Itgam', 'C4b', 'Fn1')
volcano.plot(noAb.tbi.v.ab.tbi.lattice, treat='Ablation', title='TBI No Ablation v. Ablation', highlights = x)
```

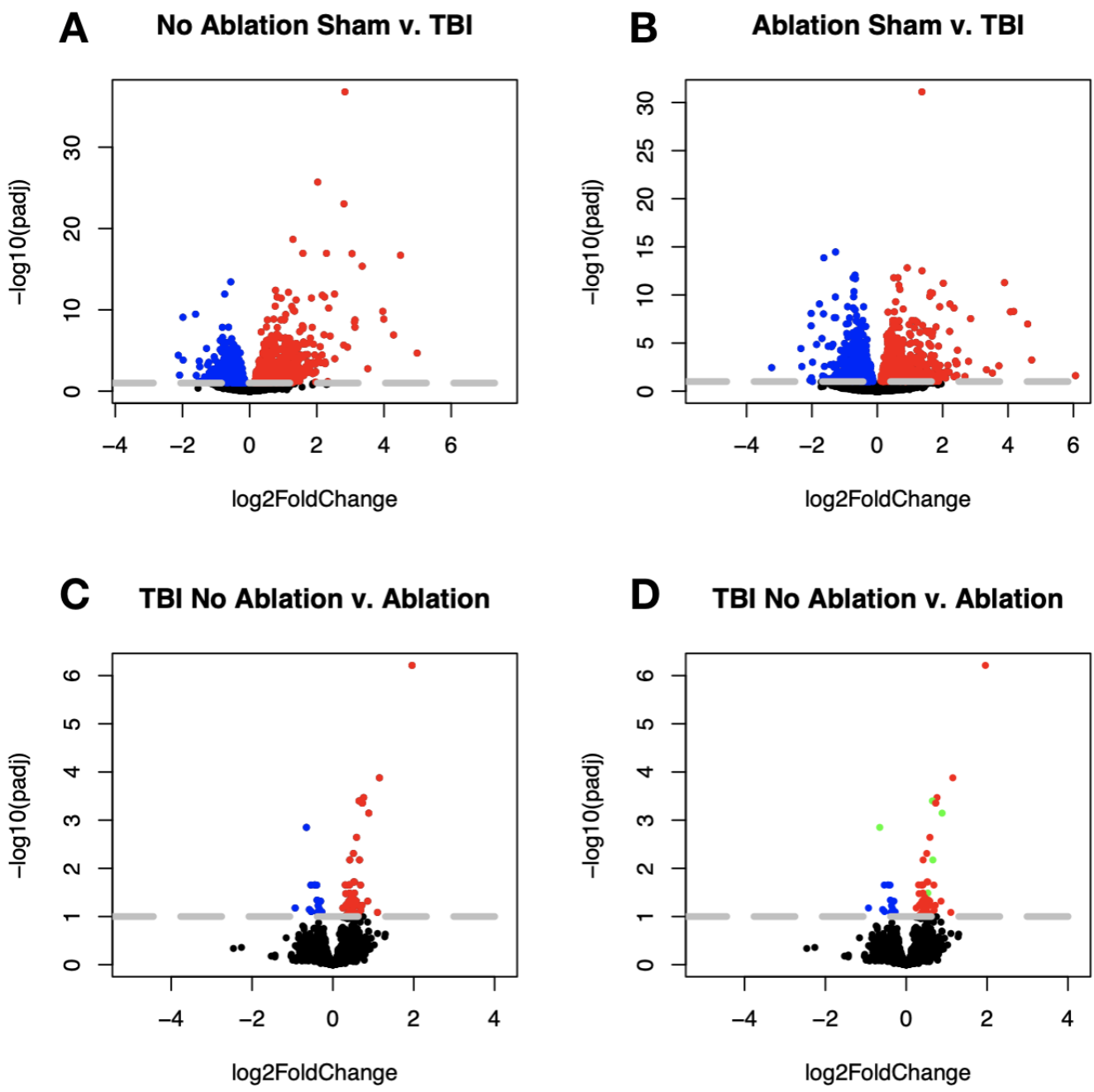

Figure 8: These four volcano plots represent different comparisons and different options for presenting the expression data. The x-values are log2FoldChange and the y-values are -log10(padj) for each gene. The dashed, horizontal, grey line marks the adjusted p-value cut off of -log10(0.1), which is roughly equivalent to 1.301. (A) Sham vs. TBI without ablation. (B) Sham vs. TBI with ablation. (C) No ablation versus Ablation with TBI. (D) No ablation versus Ablation with TBI with six genes highlighted. The six genes are described in the text.