Alternative splicing analysis: STAR

Matthew Taliaferro

Contact Info

Greetings experimentalist humans 👋

matthew.taliaferro@cuanschutz.edu

Learning Objectives

By the end of the class, you should be able to:

- Define the challenges faced by aligning RNAseq reads to the transcriptome
- Understand how STAR approaches these challenges
- Be able to run STAR to align RNAseq reads to the transcriptome
- Understand the SAM file format

Rigor & Reproducibility

As with all computational **experiments** (yes, they are experiments, don't let your pipette-toting friends tell you otherwise), keeping track of what you did is key. In the old days, I kept a written notebook of commands that I ran. Sounds silly, but there were many times that I went back to that notebook to see exactly what the parameters were for a given run using a piece of software.

Today, there are better options. You are using one of the better ones right now. Notebooks, including RMarkdown (mainly for R) and Jupyter (mainly for Python), are a great way to keep track of what you did as well as give justification or explanation for your analyses using plain 'ol English.

Trust me, one day you will be glad you used them. The Methods section of your paper is never fun to write without them.

Problem Set and Grading Rubric

Today's problem set is composed of 3 problems.

- In the first, you will use `STAR` to align some RNAseq reads to the mouse transcriptome. This problem is worth 30% of the total points.
- In the second, you will count how many reads in your alignment cross a splice junction. This problem is worth 35% of the total points.
- In the third, you will find the longest genomic span covered by a single read pair. This problem is worth 35% of the total points.

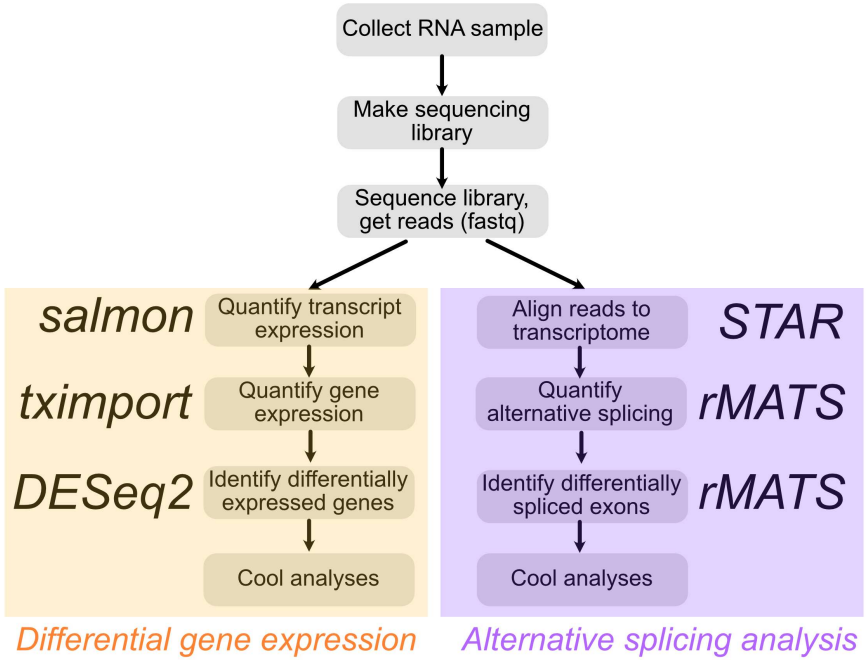
Further reading

If you are interested, here is a little more information about today's topic that you can read later:

- The paper describing STAR
- The documentation for the latest version of STAR
- In-depth information about what is contained within SAM files

Overview

Last week we talked about quantifying transcript expression with RNAseq and using it to identify genes that were differentially expressed across conditions.

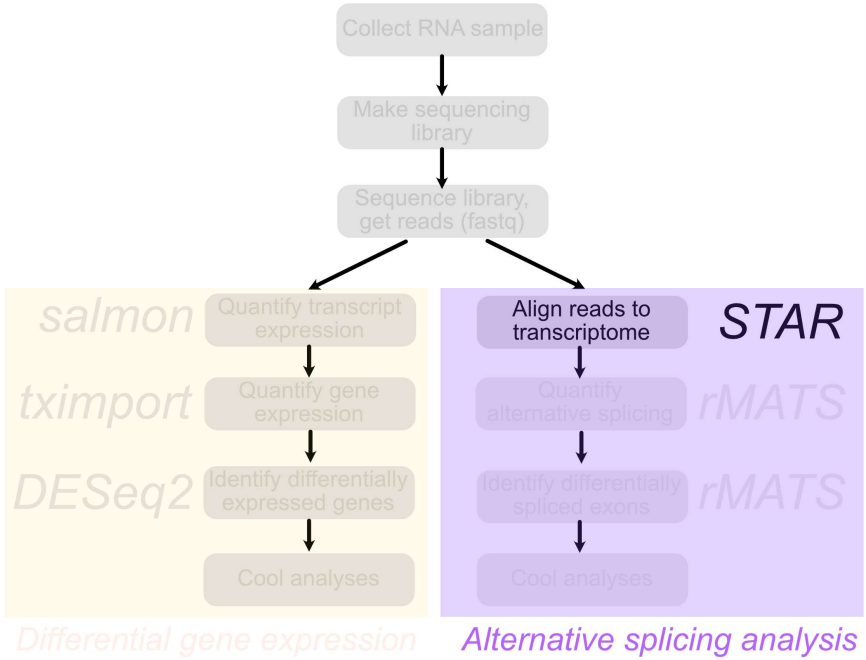


Overview

This week we will focus on the analysis of *alternative splicing*.

Unlike last week when we could move straight from raw reads in a `fastq` file to transcript quantifications, this week we will need an intermediate step: read alignment.

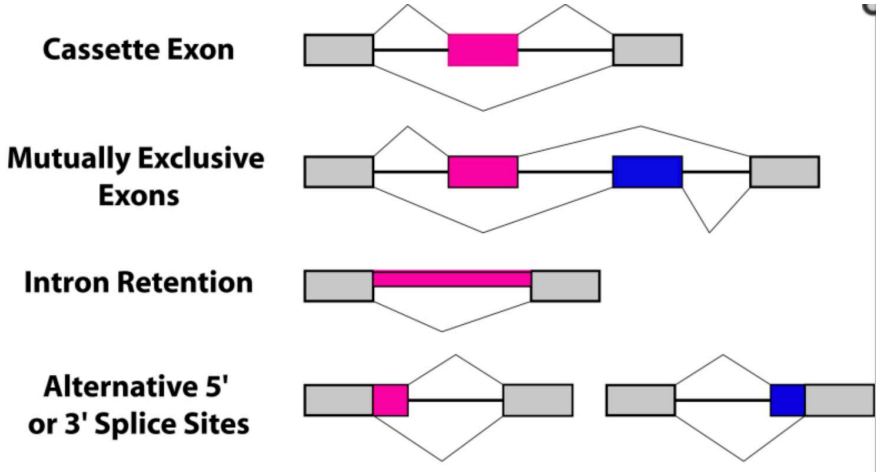
There are several software packages made for aligning RNAseq reads to the genome/transcriptome. One of the more popular ones, and the one we will use is called `STAR`.



Alternative splicing

As you may know, the vast majority of mammalian protein-coding genes contain introns. In fact most contain *many* introns, meaning they have many exons. Not every one of these exons, or pieces of an exon, has to be included in every transcript.

The inclusion of these exons is often regulated through the binding of RNA binding proteins to specific sequences in the exon and in the flanking introns.

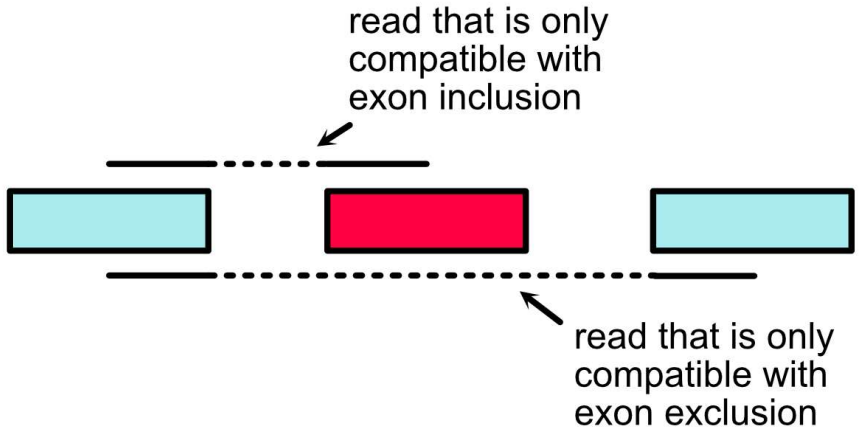


wormbook

Reads that inform us about alternative splicing

In order tell which exons were included in a transcript, we need to know more than just which transcript a read came from. We need to know *where* in the transcript it came from.

The most informative reads with regards to alternative splicing are those that **cross a splice junction**. Those reads tell us unambiguously that two exons were connected together in the transcript. If we are considering the inclusion of an alternative exon, a *junction read* could link the alternative exon to one of its neighbors, indicating that the alternative exon was included. Alternatively 😂, a junction read could link the two neighbor exons together, indiciating that the alternative exon was not included.



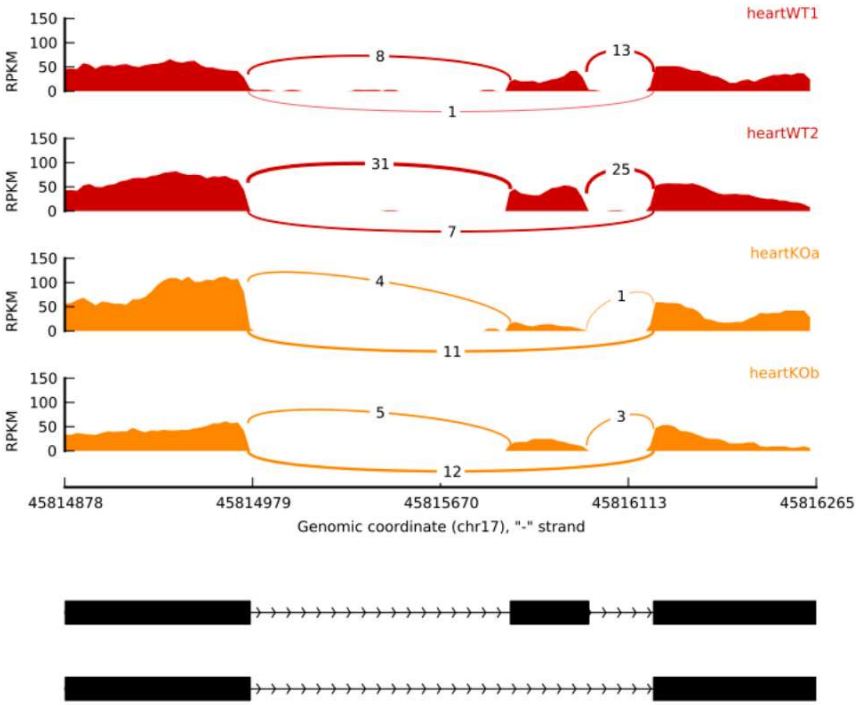
Junction reads in practice

We can use the relative number of reads that support inclusion or exclusion to learn something about how often an exon was included in a sample.

Consider the plots on the right. These are called **sashimi** plots.

We are quantifying the inclusion of the middle exon. Each row is an RNAseq sample where the height of the color indicates read depth at that location. The arcs that connect exons are junction reads. In the red samples, there are many more reads that support exon inclusion (i.e. go from the middle exon to a flanking exon) than support exclusion (i.e. go from one flanking exon to the other). In the orange samples, this trend is reversed.

We can therefore say that the exon is more often included in the red samples than the orange samples.



MISO documentation

Aligning RNAseq reads to the genome

If we are going to take this approach, we need to know *where* in the genome each read came from. What we want to do is take each read and **align** it against the genome, retrieving the genomic **coordinates** of its location. You have already done this in the DNA module using `bowtie`.

You might immediately see a problem, though. Many reads will not *contiguously* map to the genome. Specifically, the junction reads that we are so interested in will have two regions of alignment (one for each exon) separated by a gap (the intron).

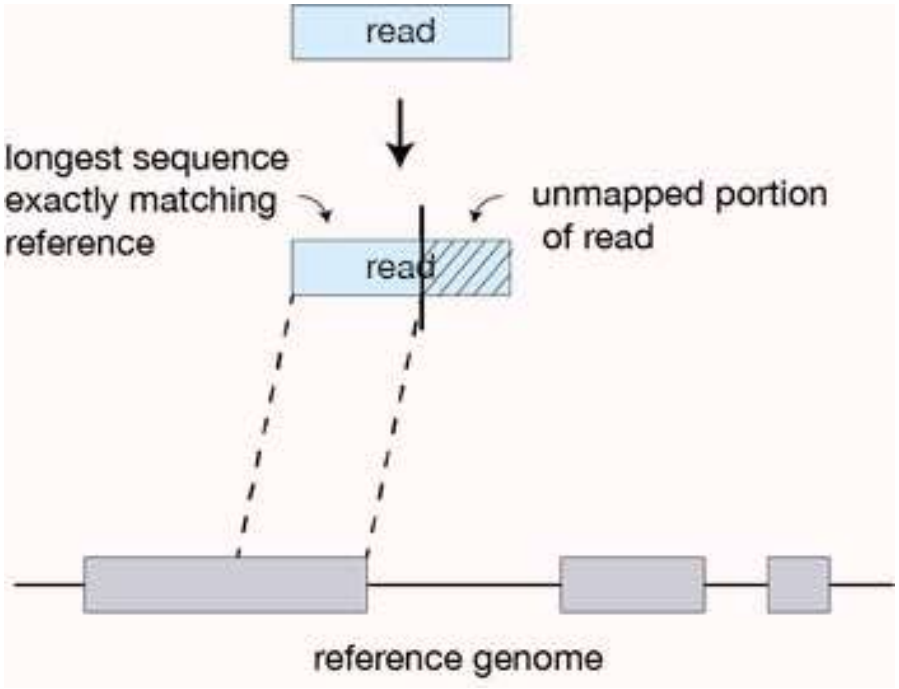
`bowtie` is not going to like this. However, an aligner made for RNAseq, like `STAR` will be able to handle this.

Yes, I recently learned how to insert emojis

How STAR works

STAR begins by finding matches (either unique or nonunique) between a portion of a read and the reference. This matching region of the query is extended along the reference until the two start to disagree.

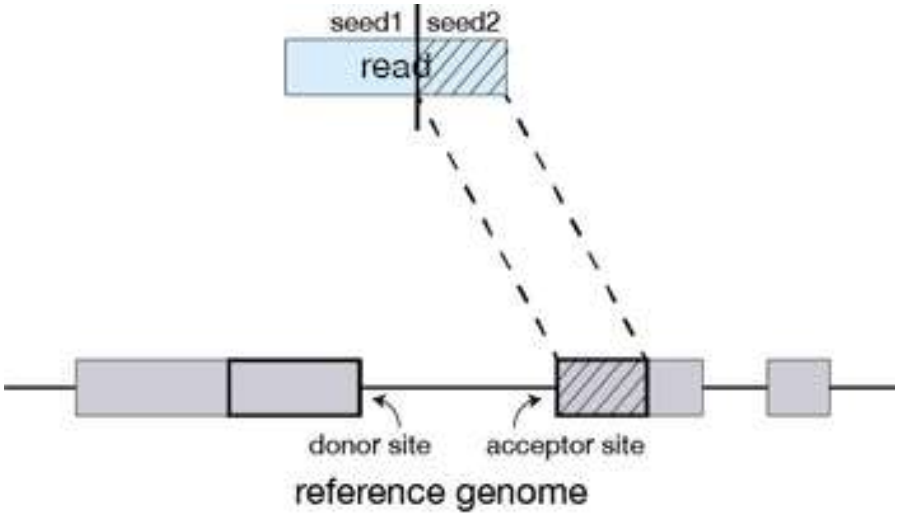
If this match extends all the way through to the end of the read, then the read lies completely within one exon (or intron, or I guess intergenic region if you are bad at making RNAseq libraries) and we are done. If the match ends before the end of the read, the part that has matched so far defines one *seed*.



Harvard Chan Bioinformatics Core

How STAR works

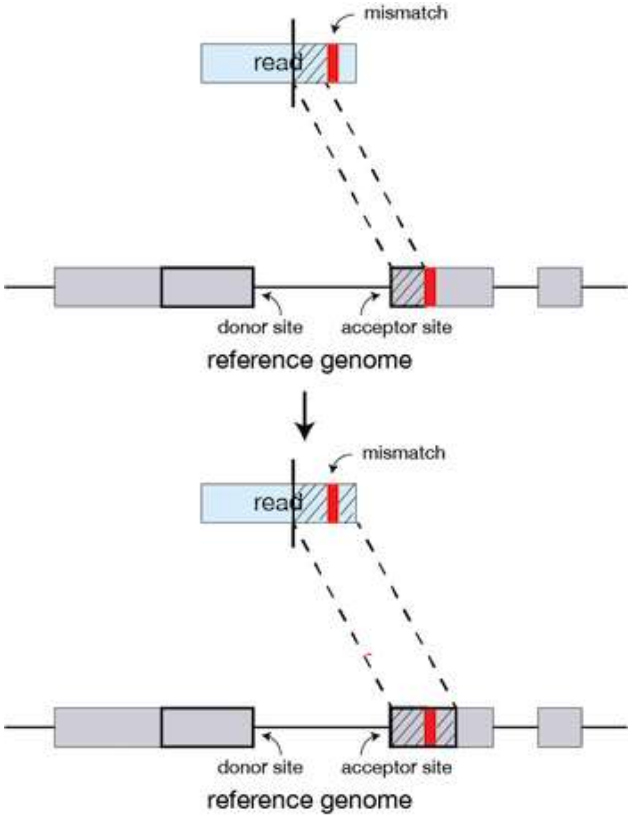
STAR then takes the rest of the query and uses it to find the best match to its sequence in the reference, defining another **seed**.



Harvard Chan Bioinformatics Core

How STAR works

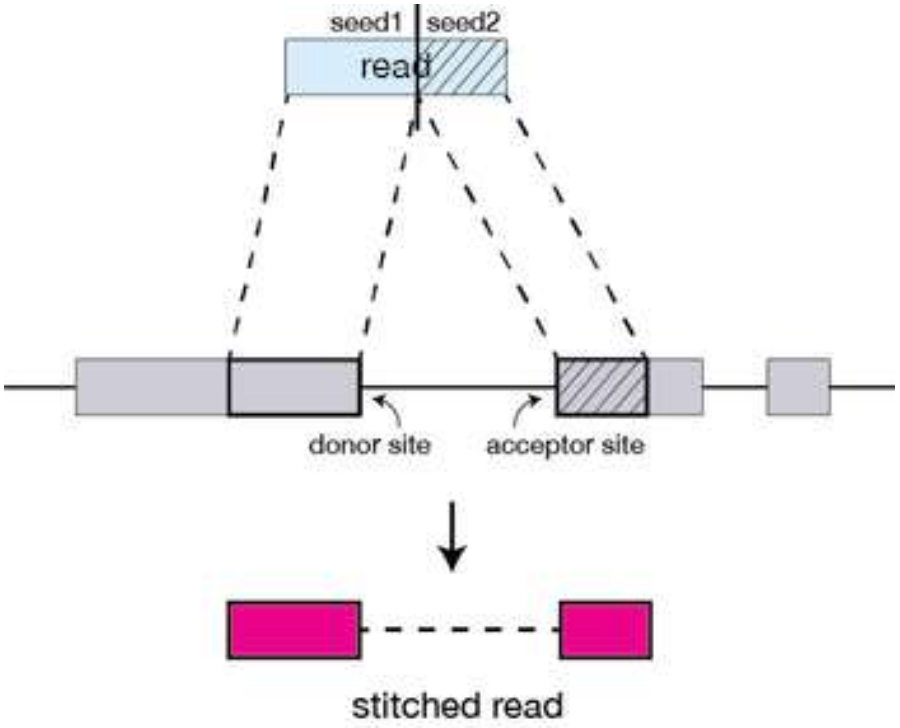
If, during the extension of a match a small region of mismatch or discontinuity occurs, these can be identified as mutations or indels if high-quality matches between the query and reference resume later in the read.



How STAR works

After aligning seeds, they can be stitched together.

The stitching of seeds with high alignment quality (low number of indels, mismatches) is prefered over the stitching of seeds with low alignment quality (high number of indels, mismatches).



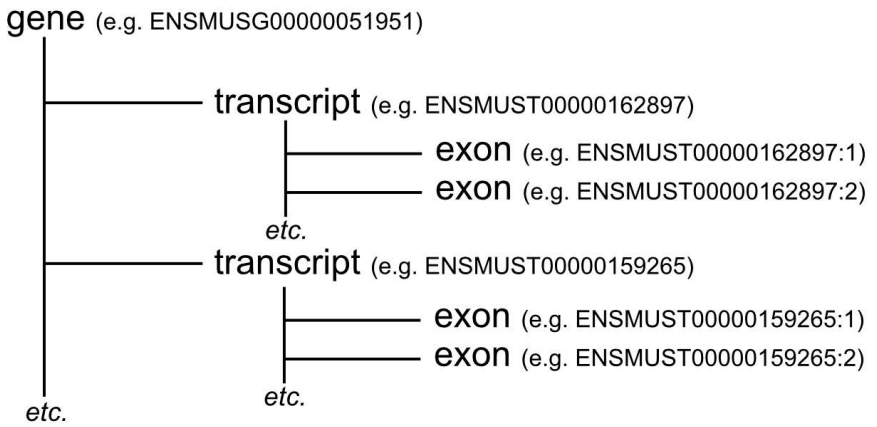
Harvard Chan Bioinformatics Core

Genome annotations

So if STAR is going to do this, it needs to know where the introns are. Although it can find "new" (i.e. unannotated) introns, unless we were specifically looking for new ones, we would probably have much more confidence in introns that had been seen before.

We can provide those introns that people agree exist in the form of a genome annotation. There are a few flavors of genome annotation, but the one we will use is called **GFF** (General Feature Format). General information about the GFF format can be found here.

GFF files are hierarchical in nature. There can be many levels in this hierarchy. Parents have children, grandchildren, and so on. For our purposes, the top level feature will be a gene. Genes have children transcripts which in turn have children exons.



Where are the introns in this diagram?

GFF file format

Let's take a look at one of these files. This is drawn from the annotation for mouse chromosome 19.

```
head -n 4 data/dummySTAR/MOLB7950.gff3
```

```
#> ##sequence-region chr19 1 61431566
#> chr19    HAVANA    gene    3065711    3197714    .    -    .    ID=ENSMUSG00000100969.6;gene_id=E
#> chr19    HAVANA    transcript    3065711    3197714    .    -    .    ID=ENSMUST00000190575.6;Par
#> chr19    HAVANA    exon    3197605    3197714    .    -    .    ID=exon:ENSMUST00000190575.6:1;Pa
```

We can see that each line corresponds to one feature. The first feature is a gene. It:

- Lives on chromosome 19.
- Begins at position 3065711 (just to add confusion, GFF files start at 1, BED files start at 0)
- Ends at position 3197714
- Is on the minus strand (this isn't DNA anymore...strand matters!)

The gene feature contains a child transcript feature.

Its properties are similarly outlined (*scroll right*).

Running STAR -- creating an index

The first thing we need to do is have STAR create an **index**, storing information about the annotation and where things are. The relevant options we will need to pay attention to when doing this are shown below:

- **--runMode** genomeGenerate (we are making an index, not aligning reads)
- **--genomeDir** /path/to/genomeDir (where you want STAR to put this index we are making)
- **--genomeFastaFiles** /path/to/genomesequence (genome sequence as fasta, either one file or multiple)
- **--sjdbGTFfile** /path/to/annotations.gff (yes it says gtf, but we are going to use a gff format)
- **--sjdbOverhang** 100 (100 will usually be a good value here, the recommended value is readLength - 1)
- **--sjdbGTFtagExonParentTranscript** Parent (we have to specify this because we are using a gff annotation and this is how gff files denote relationships)
- **--genomeSAindexNbases** 11 (don't worry about this one, we are specifying it because we are using an artifically small genome in this example) Now we are ready to make our index. This should take a couple minutes.

```
/Users/mtaliaferro/miniconda2/envs/three/bin/STAR --runMode genomeGenerate /
--genomeDir data/dummySTAR/dummySTARindex --genomeFastaFiles /
data/dummySTAR/chr19.fasta --sjdbGTFfile /
data/dummySTAR/MOLB7950.gff3 --sjdbOverhang 100 --sjdbGTFtagExonParentTranscript Parent --genomeSAi
```

Running STAR -- aligning reads

Now that we have our index we are ready to align our reads. The options we need to pay attention to here are:

- **--runMode** alignReads (we are aligning this time)
- **--genomeDir** /path/to/genomeDir (a path to the index we made in the previous step)
- **--readFilesIn** /path/to/forwardreads /path/to/reversereads (paths to our fastqs, separated by a space)
- **--readFilesCommand** gunzip -c (our reads our gzipped so we need to tell STAR how to read them)
- **--outFileNamePrefix** path/to/outputdir (where to put the results)
- **--outSAMtype** BAM SortedByCoordinate (the format of the alignment output, more on this later) Now we are ready to align our reads.

```
/Users/mtaliaferro/miniconda2/envs/three/bin/STAR --runMode alignReads --genomeDir /
data/dummySTAR/dummySTARindex/ --readFilesIn /
data/dummySTAR/MOLB7950_1.fastq.gz /
data/dummySTAR/MOLB7950_2.fastq.gz /
--readFilesCommand gunzip -c --outFileNamePrefix data/dummySTAR/MOLB7950/dummy /
--outSAMtype BAM SortedByCoordinate
```

What does `STAR` give us as an output? We'll look at that in the exercises.

Alternative splicing analysis: rMATS

Matthew Taliaferro

Contact Info

Greetings experimentalist humans 👋

matthew.taliaferro@cuanschutz.edu

Learning Objectives

By the end of the class, you should be able to:

- Define the meaning and rationale behind the "percent spliced in" (PSI) metric for alternative splicing analysis
- Use `rMATS` to analyze alternative splicing using RNAseq data
- Parse and filter `rMATS` output to derive meaningful insights

Rigor & Reproducibility

As with all computational **experiments** (yes, they are experiments, don't let your pipette-toting friends tell you otherwise), keeping track of what you did is key. In the old days, I kept a written notebook of commands that I ran. Sounds silly, but there were many times that I went back to that notebook to see exactly what the parameters were for a given run using a piece of software.

Today, there are better options. You are using one of the better ones right now. Notebooks, including RMarkdown (mainly for R) and Jupyter (mainly for Python), are a great way to keep track of what you did as well as give justification or explanation for your analyses using plain 'ol English.

Trust me, one day you will be glad you used them. The Methods section of your paper is never fun to write without them.

Problem Set and Grading Rubric

Today's problem set is composed of 2 problems.

- In the first, you will receive an `rMATS` output file. You will be tasked with filtering the events by read coverage and then plotting the difference in PSI values and their statistical significance using a volcano plot. This problem is worth 50% of the total points.
- In the second, you will take the filtered (by read coverage) `rMATS` output and plot the PSI values statistically significant events using a heatmap. This will allow you to easily judge the consistency of PSI values across replicates in these events. This problem is worth 50% of the total points.

Further reading

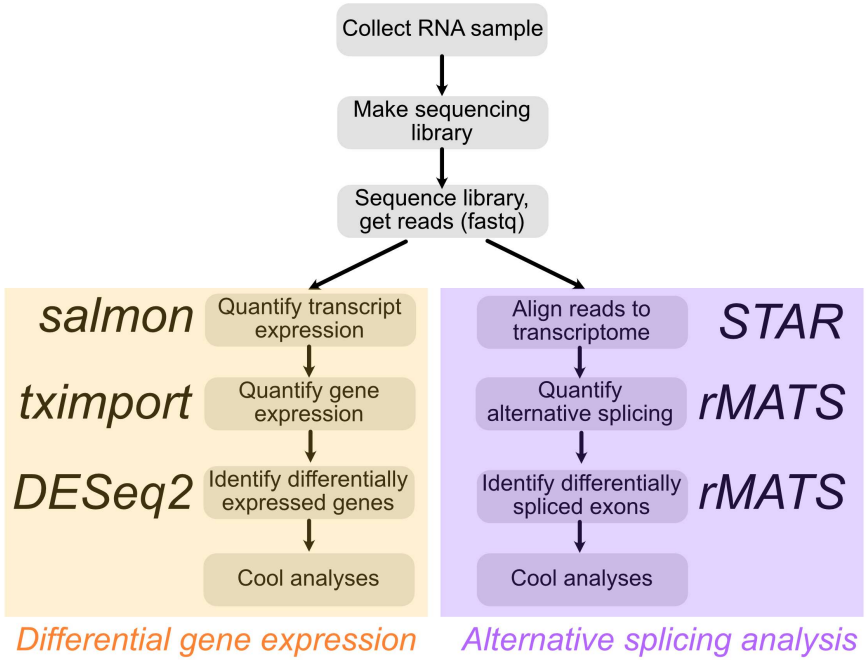
If you are interested, here is a little more information about today's topic that you can read later:

- The paper describing `rMATS`
- The documentation for the latest version of `rMATS`

Overview

Last time we talked about using STAR to align RNAseq reads to the transcriptome. We talked about why this was important for alternative splicing analysis given that we need to know exactly *where* in transcripts these reads are mapping.

Today we will talk about taking the alignments that we produced (OK not those exact alignments, but still) and using them to determine how the inclusion of alternative exons in transcripts varies between conditions.



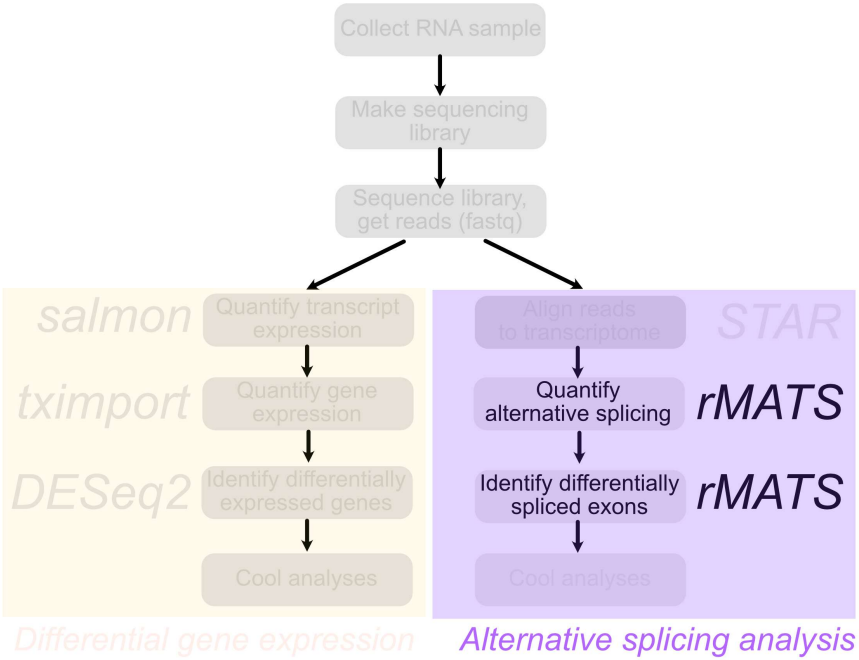
Overview

Today we will continue our focus on the analysis of *alternative splicing*.

As is the case for many computational biology questions, there are many software choices when it comes to alternative splicing. The one that we will use is called `rMATS`.

`rMATS` will take data about where reads align and where alternative exons are to determine how often an exon is included in a sample. It will quantify this using a metric called PSI (Percent Spliced In).

`rMATS` will then compare PSI values between replicates and across experimental conditions to identify alternative splicing events whoe PSI values are significantly different across conditions.

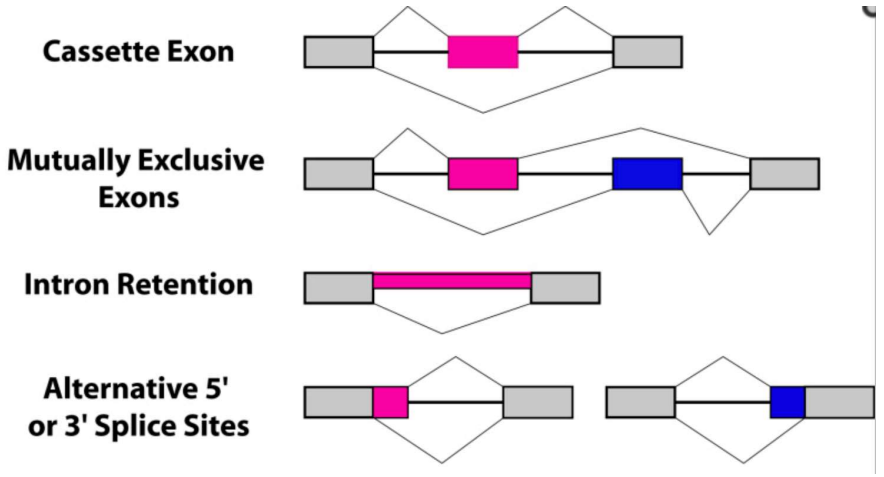


Alternative splicing

As you may know, the vast majority of mammalian protein-coding genes contain introns. In fact most contain *many* introns, meaning they have many exons. Not every one of these exons, or pieces of an exon, has to be included in every transcript.

The inclusion of these exons is often regulated through the binding of RNA binding proteins to specific sequences in the exon and in the flanking introns.

The data we will examine today came from an experiment probing the function of one of these RNA binding proteins, **RBFOX2**. More on that later.

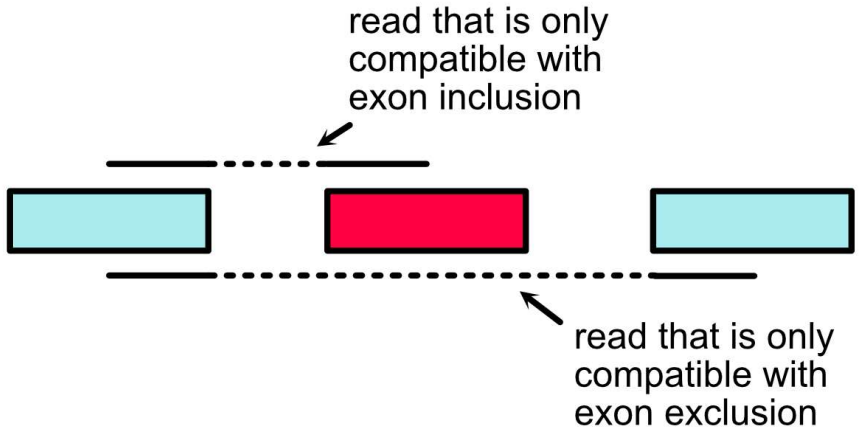


wormbook

Reads that inform us about alternative splicing

In order tell which exons were included in a transcript, we need to know more than just which transcript a read came from. We need to know *where* in the transcript it came from.

The most informative reads with regards to alternative splicing are those that **cross a splice junction**. Those reads tell us unambiguously that two exons were connected together in the transcript. If we are considering the inclusion of an alternative exon, a *junction read* could link the alternative exon to one of its neighbors, indicating that the alternative exon was included. Alternatively 😄 (was this funny this time?), a junction read could link the two neighbor exons together, indiciating that the alternative exon was not included.



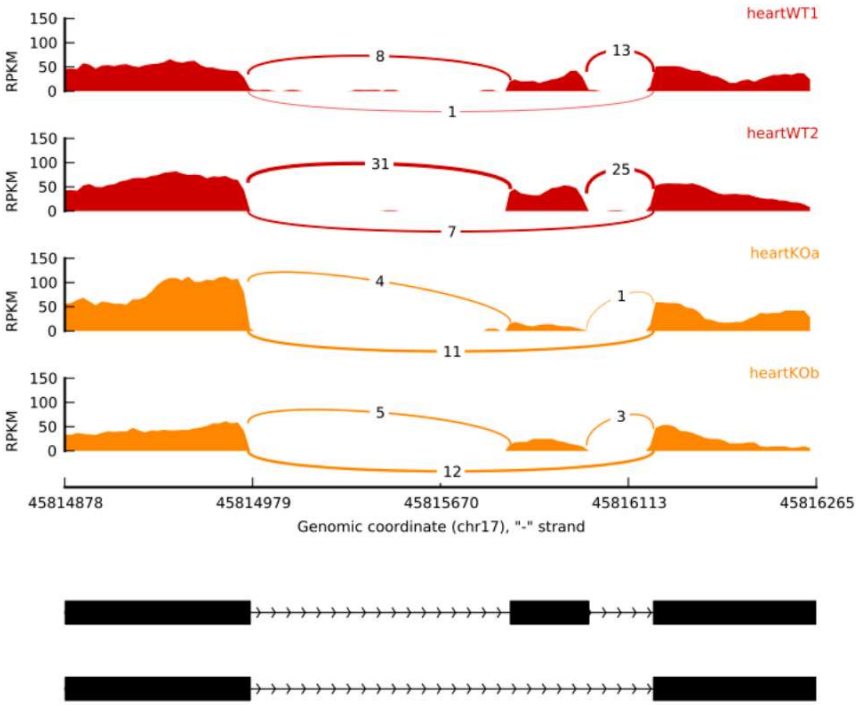
Junction reads in practice

We can use the relative number of reads that support inclusion or exclusion to learn something about how often an exon was included in a sample.

Consider the plots on the right. These are called **sashimi** plots.

We are quantifying the inclusion of the middle exon. Each row is an RNAseq sample where the height of the color indicates read depth at that location. The arcs that connect exons are junction reads. In the red samples, there are many more reads that support exon inclusion (i.e. go from the middle exon to a flanking exon) than support exclusion (i.e. go from one flanking exon to the other). In the orange samples, this trend is reversed.

We can therefore say that the exon is more often included in the red samples than the orange samples.



MISO documentation

Today's data

Today, we will use data produced as part of a study into the function of the RNA binding protein **RBFOX2**.

Rbfox2 controls autoregulation in RNA-binding protein networks

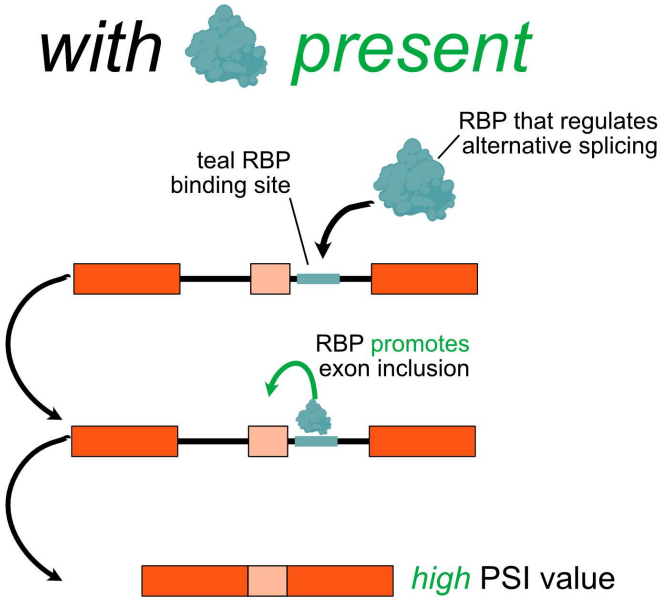
Mohini Jangi,[1,2] Paul L. Boutz,[1] Prakriti Paul,[3] and Phillip A. Sharp[1,2,4]

[1]David H. Koch Institute for Integrative Cancer Research, [2]Department of Biology, [3]Department of Biological Engineering, Massachusetts Institute of Technology, Cambridge, Massachusetts 02139, USA

Studying RNA regulation through RBP perturbation / RNAseq

A common strategy in the study of RNA processing is to take an RBP you are interested in, perturb it, and ask how the transcriptome looks before and after this perturbation. This is part of what the authors did in today's paper.

Imagine you were studying an RBP that promoted inclusion of an alternative exon by binding the intron downstream of exons that it controls.

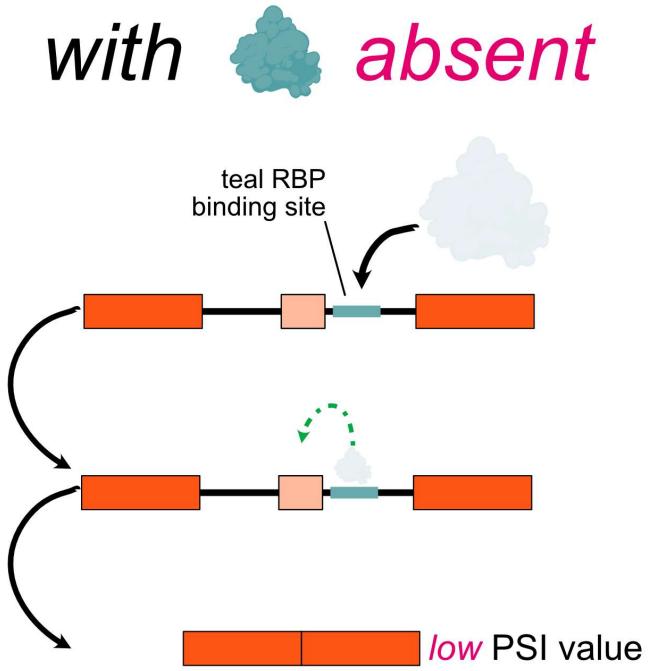


Studying RNA regulation through RBP perturbation / RNAseq

Now imagine that you perturbed that RBP, say, by knocking its expression down. What would happen?

Exons that depended on that RBP for efficient inclusion would now be included in transcripts much less frequently. In this example, their PSI values would *decrease*.

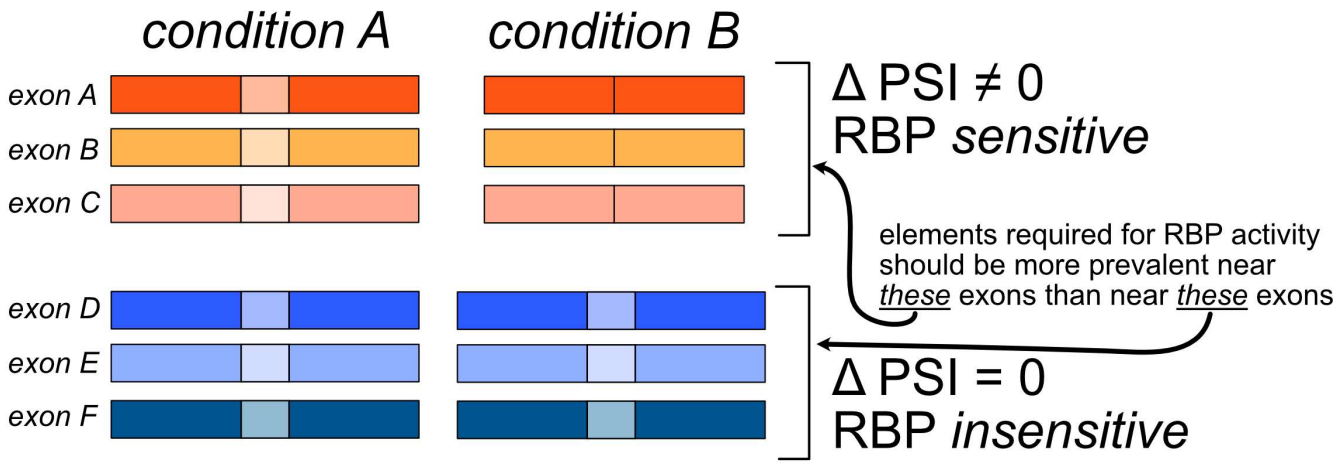
However, other exons don't really care about whether this RBP is around or not. Their PSI values will be generally unaffected.



Uncovering biochemical mechanism through genomics

Identifying RBP-sensitive exons in this manner therefore tells us two things:

- The functional splicing targets of the RBP
- Motifs that are associated with whether or not an exon is sensitive to RBP perturbation
  - These could be elements in the RNA that the RBP recognizes to direct its activity to this location



More on this in the next class. For now, let's focus on running `rMATS` and figuring out its output.

Run rMATS

`rMATS` is going to want two things:

- The locations of alternative exons (provided as a `gtf` genome annotation file)
- The locations of your read alignments, in `bam` format

See the documentation here. The relevant options we will need to pay attention to when doing this are shown below:

- **--b1** /path/to/b1.txt (path to a text file that contains paths to all BAM files for samples in condition 1, i.e. RBP perturbation)
- **--b2** /path/to/b2.txt (path to a text file that contains paths to all BAM files for samples in condition 2, i.e. Control perturbation)
- **--gtf** /path/to/gtf (path to the gtf genome annotation)
- **-t** readtype (single or paired)
- **--readlength** readlength
- **--od** /path/to/output (output directory)

rMATS output

rMATS produces several output files. The files we are interested in end in `.JC.txt`. There is one for every "type" of alternative splicing. Splicing quantification in these files was done using **only** reads that cross splice junctions. Although this reduces the statistical power of the tests that `rMATS` uses because it reduces the number of usable reads, junction reads are always completely unambiguous in relating how exons were connected together.

Let's look at one of these output files.

```
rmatsout <- read.table('data/rMATS/RBFOX2kd/rMATSouts/SE.MATS.JC.txt', header = T)

as_tibble(rmatsout) %>%
  head(n = 5)
```

```
#> # A tibble: 5 x 23
#>      ID GeneID geneSymbol chr   strand exonStart_0base exonEnd upstreamES
#>   <int> <fct>  <fct>      <fct> <fct>            <int>   <int>      <int>
#> 1     5 ENSMU… Neil1      chr9  -             57143757  5.71e7   57143450
#> 2     6 ENSMU… Wrap53     chr11 -             69562121  6.96e7   69561757
#> 3     9 ENSMU… Zfp28      chr7  +              6392181  6.39e6    6390399
#> 4    11 ENSMU… Pik3cd     chr4  -            149656603  1.50e8  149656380
#> 5    17 ENSMU… Cd81       chr7  +            143065942  1.43e8  143064673
#> # … with 15 more variables: upstreamEE <int>, downstreamES <int>,
#> #   downstreamEE <int>, ID.1 <int>, IJC_SAMPLE_1 <fct>, SJC_SAMPLE_1 <fct>,
#> #   IJC_SAMPLE_2 <fct>, SJC_SAMPLE_2 <fct>, IncFormLen <int>,
#> #   SkipFormLen <int>, PValue <dbl>, FDR <dbl>, IncLevel1 <fct>
```

Alternative splicing analysis: sequence analysis

Matthew Taliaferro

Contact Info

Greetings experimentalist humans 👋

matthew.taliaferro@cuanschutz.edu

Learning Objectives

By the end of the class, you should be able to:

- Understand how genome-scale experiments can inform biochemical mechanisms
- Convert genome coordinates into sequences programmatically
- Count the occurrences of small sequences (kmers) in a sequence file
- Identify kmers enriched in one sequence set vs. another

Rigor & Reproducibility

As with all computational **experiments** (yes, they are experiments, don't let your pipette-toting friends tell you otherwise), keeping track of what you did is key. In the old days, I kept a written notebook of commands that I ran. Sounds silly, but there were many times that I went back to that notebook to see exactly what the parameters were for a given run using a piece of software.

Today, there are better options. You are using one of the better ones right now. Notebooks, including RMarkdown (mainly for R) and Jupyter (mainly for Python), are a great way to keep track of what you did as well as give justification or explanation for your analyses using plain 'ol English.

Trust me, one day you will be glad you used them. The Methods section of your paper is never fun to write without them.

Problem Set and Grading Rubric

Today's problem set is composed of 1 problem, but it contains 2 parts. You are tasked with taking two fasta files of intronic sequences. We want to know the relative abundance of all 5mers in the two sequences.

- In part 1, you will calculate a relative enrichment for each 5mer between the two sequence files as a log2 fold ratio. This problem is worth 50% of the total points.
- In the second, for each 5mer, you will calculate a p value for this enrichment using a binomial test. You will then plot the results from both parts as a volcano plot to identify the 5mer that is *most* enriched in one set vs. another.

Further reading

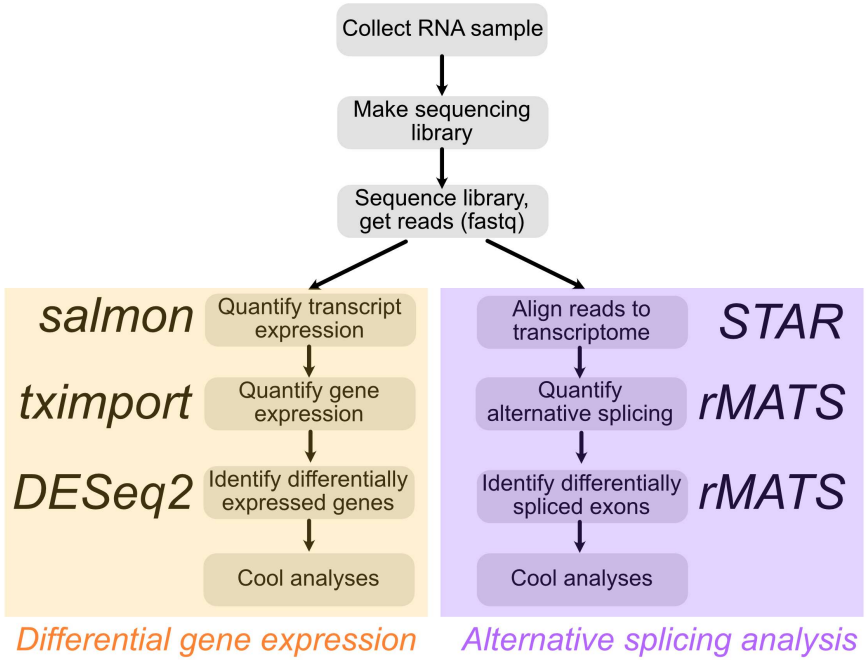
If you are interested, here is a little more information about today's topic that you can read later:

- The paper describing the RBFOX2 knockdown data we will be using today
- A paper where the authors derived preferred RNA sequence binding motifs for many RBPs, including RBFOX2

Overview

The last two sessions we have used `STAR` to align RNAseq reads to the genome (or transcriptome, if you prefer) and then used `rMATS` to take those alignments and quantify the inclusion of alternative exons in every sample.

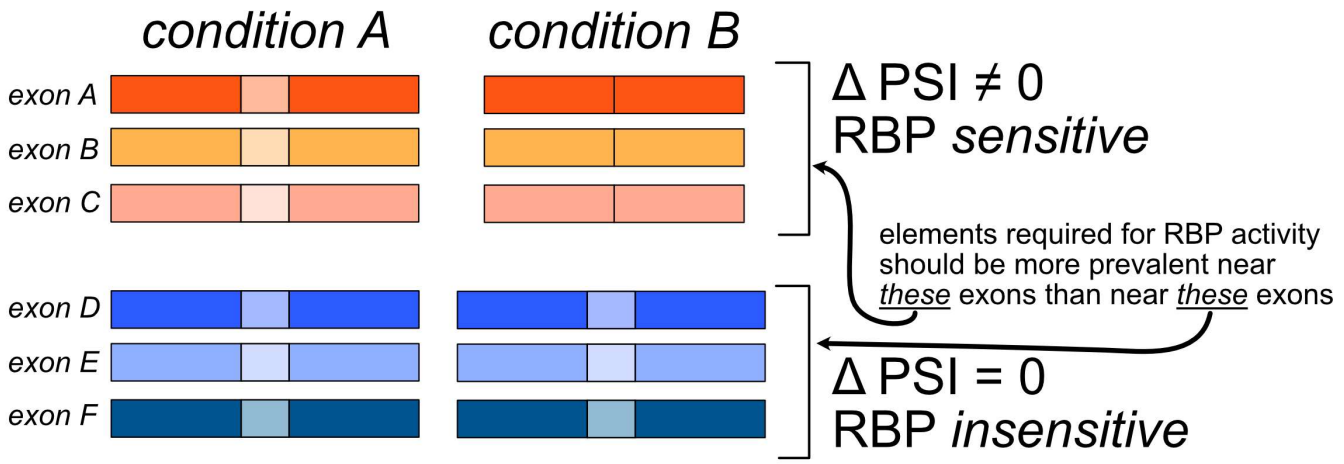
The data we used for this came from a study where the authors sequenced RNA from mouse ESCs that had been treated with either shRNAs against RBFOX2 or control shRNAs.



Overview

Today we will focus on the on the sequences that surround exons that we identified as sensitive to RBFOX2 loss.

As we discussed last time, many RBPs exert their function on RNAs by binding them, and many RBPs have a preferred RNA sequence that they like to bind. It then follows that the functional RNA targets of an RBP should be enriched for the RBP's preferred binding sequence relative to nontargets.

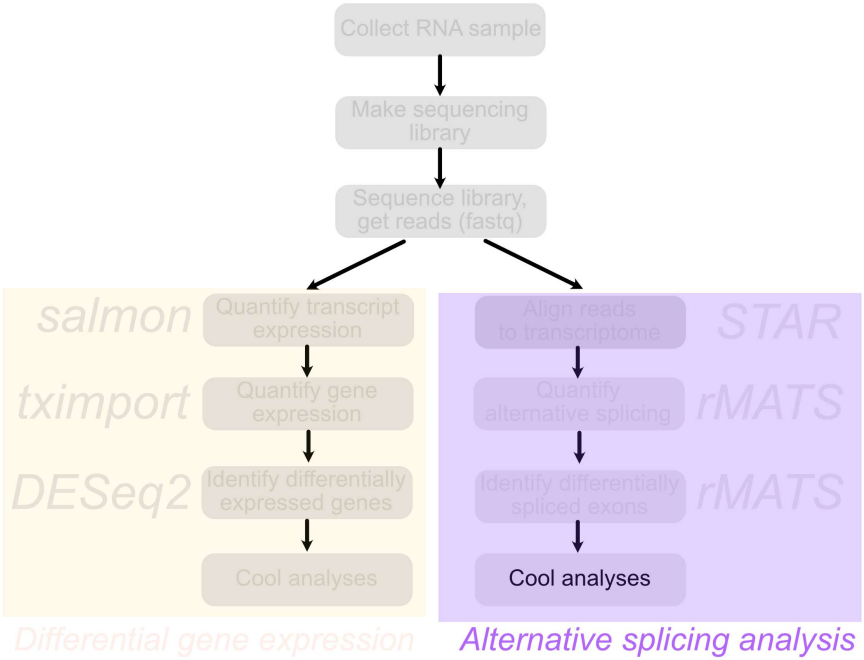


Overview

Another way to say what we are going to do today is that we are going to do some

COOL ANALYSES

😎 😎 😎 😎 😎

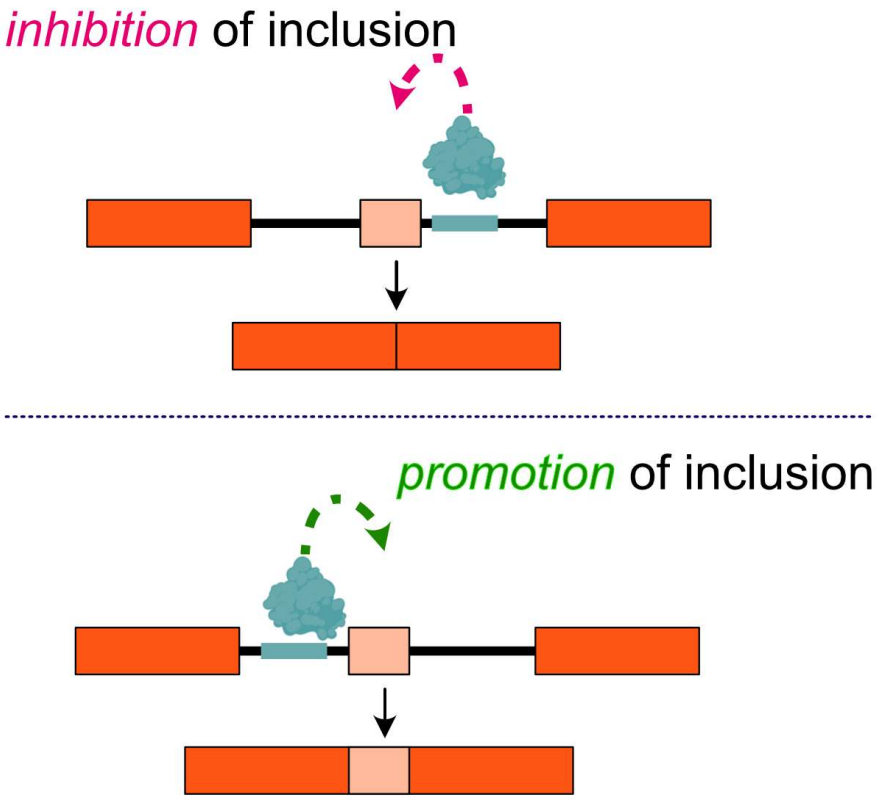


How RBPs regulate alternative splicing

As we've discussed, many RBPs regulate alternative splicing by binding sequences near alternative exons. However, it's slightly more complicated than that.

For many RBPs, *where* they bind relative to the exon can determine their activity.

For example, a given RBP might promote inclusion of the exon if it binds in the intron upstream of it and might promote exlcusion if it binds the intron downstream.



Same RBP, two different results!

This means we will have to treat the introns upstream and downstream of affected exons separately. We will also need to treat exons that become more and less included separately.

Relevant sequence regions

The sequences we are interested in will immediately flank exons that we have identified as sensitive to the loss of RBFOX2.

Specifically, we are interested in the 150 nt *upstream* (i.e. 5') of the exon and the 150 nt *downstream* (i.e. 3') of the exon.

> Note: You are dealing with RNA, not DNA. Strand matters. If the transcript is on the minus strand, the sequence upstream of the exon will have *higher* coordinates in genome space, and it will also be the *reverse complement* of the typical genome sequence given at that location.

Identifying RBFOX2-sensitive events

Our first step will be to identify exons that are sensitive to RBFOX2 loss. Let's read in an `rMATS` output table and then filter it based on read counts as we did last time.

```
rmats.filtered <- read.table('data/SE.MATS.JC.txt', header = T) %>%
  #Split the replicate read counts that are separated by commas into different columns
  separate(., col = IJC_SAMPLE_1, into = c('IJC_S1R1', 'IJC_S1R2', 'IJC_S1R3', 'IJC_S1R4'), sep = '
  separate(., col = SJC_SAMPLE_1, into = c('SJC_S1R1', 'SJC_S1R2', 'SJC_S1R3', 'SJC_S1R4'), sep = '
  separate(., col = IJC_SAMPLE_2, into = c('IJC_S2R1', 'IJC_S2R2', 'IJC_S2R3', 'IJC_S2R4'), sep = '
  separate(., col = SJC_SAMPLE_2, into = c('SJC_S2R1', 'SJC_S2R2', 'SJC_S2R3', 'SJC_S2R4'), sep = '
  #Calculate read counts per exon per sample
  mutate(., S1R1counts = IJC_S1R1 + SJC_S1R1) %>%
  mutate(., S1R2counts = IJC_S1R2 + SJC_S1R2) %>%
  mutate(., S1R3counts = IJC_S1R3 + SJC_S1R3) %>%
  mutate(., S1R4counts = IJC_S1R4 + SJC_S1R4) %>%
  mutate(., S2R1counts = IJC_S2R1 + SJC_S2R1) %>%
  mutate(., S2R2counts = IJC_S2R2 + SJC_S2R2) %>%
  mutate(., S2R3counts = IJC_S2R3 + SJC_S2R3) %>%
  mutate(., S2R4counts = IJC_S2R4 + SJC_S2R4) %>%
  #Filter on read counts
  filter(., S1R1counts >= 20 & S1R2counts >= 20 & S1R3counts >= 20 & S1R4counts >= 20 &
           S2R1counts >= 20 & S2R2counts >= 20 & S2R3counts >= 20 & S2R4counts >= 20) %>%
  #Get rid of extraneous columns
  dplyr::select(., c(geneSymbol, chr, strand, exonStart_0base, exonEnd, FDR, IncLevelDifference)) %
  as_tibble(.)
```

Identifying RBFOX2-sensitive events

```
head(rmats.filtered, n = 10)
```

```
#> # A tibble: 10 x 7
#>    geneSymbol chr   strand exonStart_0base
#>    <fct>      <fct> <fct>            <int>
#>  1 Cd81       chr7  +            143065942 1
#>  2 Smarca4    chr9  +             21632804
#>  3 Smarca4    chr9  +             21677952
#>  4 Ubxn2a     chr12 -              4891314
#>  5 Wbp2       chr11 -            116080545 1
#>  6 Eme1       chr11 -             94647290
#>  7 Eme1       chr11 -             94650222
#>  8 Sav1       chr12 -             69975967
#>  9 Rpl18a     chr8  -             70895915
#> 10 Rpl18a     chr8  -             70895915
```

From this table, we can see where each exon starts and ends, as well as the chromosome and strand that it is on. We can use this information to get the upstream and downstream intronic regions we want.

We also have the FDR and difference in PSI values between the two conditions here (RBFOX2 kd - Control kd). For the purposes of our analyses, we will focus on exons whose inclusion decreases upon RBFOX2 knockdown. This means we will want exons with positive values for `IncLevelDifference`.

Defining sensitive and insensitive exons

We need to define a set of exons whose inclusion decreases upon RBFOX2 knockdown and a set of exons whose inclusion doesn't change.

```
psis.sensitive <- filter(rmats.filtered, FDR <
psis.insensitive <- filter(rmats.filtered, FDR

nrow(psis.sensitive)
nrow(psis.insensitive)
```

```
#> [1] 114
```

```
#> [1] 4087
```

OK so we have 114 exons that are sensitive to RBFOX2 knockdown and 4087 exons that are insensitive.

Now we need to get the coordinates of the intronic regions we are interested in that surround these exons.

Get sequences of intronic regions flanking affected exons

There are many ways that we could do this, but we are going to use our old friend PYTHON .

First, let's write a file that contains the exonic coordinates for all of the exons in our sensitive and insensitive groups.

```
psis.sensitive %>%
  #We only want the columns that tell us about where a sequence is (chr, strand, exonic boundaries)
  dplyr::select(., -c(FDR, IncLevelDifference)) %>%
  write.table(., file = 'data/sensexons.coords.txt', sep = '\t', row.names = F, col.names = F, quot

psis.insensitive %>%
  dplyr::select(., -c(FDR, IncLevelDifference)) %>%
  write.table(., file = 'data/insensexons.coords.txt', sep = '\t', row.names = F, col.names = F, qu
```

Get sequences of intronic regions flanking affected exons

Let's take a look at those files to make sure they are what we want.

```
head -n 5 data/sensexons.coords.txt
```

```
#> Mff    chr1    +    82741817    82741976
#> Rps24    chr14    +    24495429    24495449
#> Egfl7    chr2    +    26590379    26590495
#> Scnm1    chr3    -    95130163    95130358
#> Fam49b    chr15    -    63956599    63956682
```

As a reminder, the columns of this tab-delimited file are:

- Gene name
- chromosome
- strand
- beginning of exon
- end of exon

From coordinate to sequence

A given nucleotide can be defined in the genome with 3 parameters: chromosome, coordinate, and strand. Luckily, rMATS records these data for the alternative exon as well as it's two neighbor exons, upstream and downstream. Consider the first event above from the gene Mff. The alternative exon begins at coordinate 82741817 in chr1 and is on the + strand. The exon extends until coordinate 82741976. OK so how can we go from this data to actual sequence, you know, As and Cs and Gs and Us and such?

When we were learning python in the bootcamp, we talked about slicing strings using indicies.

```
#A very short chrosomsome
chr = 'ACTGATCGATCATCGATCGGAATCG'
#My sequence of interest begins at
#(0-based) 4 and goes up to
#(but doesn't include!!) 12
myseq = chr[4:12]
print(myseq)
```

```
#>  ATCGATCA
```

From coordinate to sequence

So you can see what's going on here, but imagine that we had more than one chromosome, which all interesting organisms do (sorry E. coli). How would we deal with that? Here our old friend the dictionary will come to the rescue. The keys in our dictionary will be chromosome names and their values will be the sequence of the chromosome.

```
#A very short chrosomsome
chr1seq = 'ACTGATCGATCATCGATCGGAATCG'
chr2seq = 'TGATCGATCGATCGATCGATCGAGC'

genome = {}
genome['chr1'] = chr1seq
genome['chr2'] = chr2seq

#My sequence of interest begins at
#(0-based) 4 of chr1 and goes up
#to (but doesn't include!!) 12
myseq = genome['chr1'][4:12]
print(myseq)
```

```
#>  ATCGATCA
```

From coordinate to sequence

We are ready to take exonic coordinates and retrieve their flanking intronic sequences. The `biopython` library has a set of useful functions here. You can give it a genome sequence, and it will make a dictionary that is of the structure we described above where keys are chromosome names and values are sequences. For simplicity here, we will pretend that the entire of the genome is chromosome 19.

```
from Bio import SeqIO

genomefasta = 'data/chr19.fasta'
#Make a Biopython 'fasta object' of the genome
genomefasta_obj = SeqIO.parse(open(genomefasta, 'r'), 'fasta')
#Make a dictionary of the 'genome'
seq_dict = SeqIO.to_dict(genomefasta_obj)


#Now let's read in the coordinate files we made earlier.
#For each file, we will make two fastas: one of the upstream
#intronic 150 nt and one of the downstream intronic 150 nt
with open('data/sensexons.coords.txt', 'r') as coordfh, open('data/sensexons.chr19.upstream.fa',
  for line in coordfh:
    #Remove trailing newline characters and turn each line into a list
    line = line.strip().split('\t')
    chrm = line[1]
    strand = line[2]
    exonstart = int(line[3])
    exonstop = int(line[4])
    seqid = ('_'.join(line)
```

From coordinate to sequence

Let's take a look at the sequence file we made of the intronic regions that are *upstream* of RBFOX2-sensitive exons.

```
head data/sensexons.chr19.upstream.fa
```

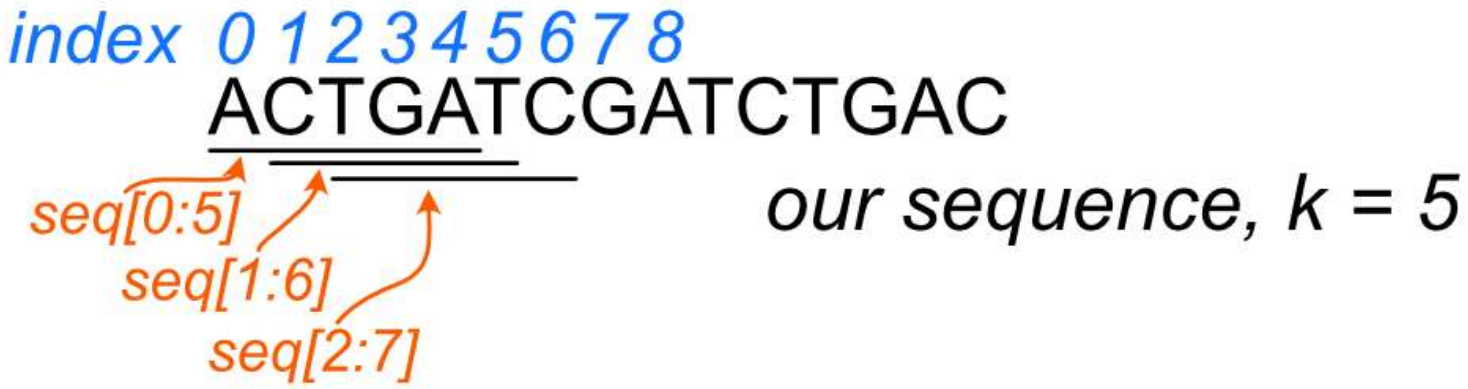
```
#> >Adrbk1_chr19_-_4288411_4288507
#> GCUCAGGAGGUAAAAGAAAGUCCUUUCUUUCGUUCCCUGGACUG
#> >Naa40_chr19_-_7229947_7230106
#> AGUGUCUGGAUUGGAGCCAGCCACUGUGGACUGGGCCUUCGACC
#> >Smc5_chr19_-_23215154_23215199
#> AUAAGUUCAGAAGAUAGUCUUGAUGGAGUUCUGUCUCAGUGAUU
```

Counting kmers

Now that we have our sequences, we want to ask which kmers are enriched in the sensexons sequences relative to the insensexons sequences. Kmers are just nucleotide sequences of length k. Often in these types of analyses, we will look for the enrichment of 5mers or 6mers (k = 5 or 6). Here, we will look at 5mers.

So what we want to do is essentially ask, for every possible 5mer, what is the density of that 5mer in the sens sequences and how does it compare to the density in the insens sequences? Another way to ask that is "for all of the kmers in the sequences, what fraction of them are kmer X"? So we need to have a way to look at a sequence and count how many times each kmer occurs in that sequence. Luckily, we can do that fairly easily with `python` .

Our overall strategy is below:



Counting kmers

```
testseq = 'AUUAGCUAGCUAGCGACGCAGUACGUACGUAGCUAGCUAGCUAGUAUGCAUGAUGCUGACUG'

#All we need to do is take a window that is 5 nt wide
#and slide it along the sequence one nt at a time,
#recording every kmer that is in a window
kmercounts = {} #{kmer : number of times we observe that kmer}

k = 5

for i in range(len(testseq) - k + 1):
  kmer = testseq[i : i+k]
  #If we haven't seen this kmer before, its count is 1
  if kmer not in kmercounts:
    kmercounts[kmer] = 1
  #If we have seen this kmer before, add one to its count
  elif kmer in kmercounts:
```

```
#> {'GUAUG': 1, 'ACGCA': 1, 'UGAUG': 1, 'CGCAG': 1, 'GCUGA': 1, 'AGUAU': 1, 'CUGAC': 1, 'AGUAC': 1,
```

Now that we can count occurences of kmers in a sequence, all that is left is doing that for every sequence in a fasta file and then comparing those counts across files.

In today's exercises we will do just that. Remember, our goal is to learn something about RBFOX2 based on the sequences that are er

RNAseq: QC

Matthew Taliaferro

Not that long ago in

a lab not so far away...

Contact Info

Greetings experimentalist humans 👋

matthew.taliaferro@cuanschutz.edu

Learning Objectives

By the end of the class, you should be able to:

- Understand how `salmon` can be used to quantify gene expression given an RNAseq dataset
- Collapse transcript-level quantifications to gene level quantifications using `tximport`
- Deduce the relationships of samples to each other using hierarchical clustering and PCA

Rigor & Reproducibility

As with all computational **experiments** (yes, they are experiments, don't let your pipette-toting friends tell you otherwise), keeping track of what you did is key. In the old days, I kept a written notebook of commands that I ran. Sounds silly, but there were many times that I went back to that notebook to see exactly what the parameters were for a given run using a piece of software.

Today, there are better options. You are using one of the better ones right now. Notebooks, including RMarkdown (mainly for R) and Jupyter (mainly for Python), are a great way to keep track of what you did as well as give justification or explanation for your analyses using plain 'ol English.

Trust me, one day you will be glad you used them. The Methods section of your paper is never fun to write without them.

Problem Set and Grading Rubric

Today's problem set is composed of 2 problems. Each of them is worth 50% of the total points. For both of these problems, you will be presented with an RNAseq dataset. Transcript quantification will have already been performed using `salmon`.

- In the first, you must take those transcript-level quantifications and convert them to gene-level quantifcations using `tximport`. You will then filter genes based on whether or not they meet an expression level cutoff in all samples and then determine relationships between samples using hierarchical clustering.
- In the second, you will take the same dataset and determine relationships between samples using PCA.

Further reading

If you are interested, here is a little more information about today's topic that you can read later:

- The paper describing `salmon` and its approach to transcriptome quantification
- The documentation for `salmon`
- The documentation for `tximport`

Overview

In this class, we will examine RNAseq data collected over a timecourse of differentiation from mouse embryonic stem cells to cortical glutamatergic neurons (Hubbard et al, F1000 Research (2013)). In this publication, the authors differentiated mESCs to neurons using a series of *in vitro* culture steps over a period of 37 days. During this timecourse, samples were extracted at selected intervals for transcriptome analysis. Importantly, for each timepoint, either 3 or 4 samples were taken for RNA extraction, library preparation and sequencing. This allows us to efficiently use the statistical frameworks provided by the DESeq2 package to identify genes whose RNA expression changes across the timecourse.

DATA ARTICLE

Longitudinal RNA sequencing of the deep transcriptome during neurogenesis of cortical glutamatergic neurons from murine ESCs [version 1; peer review: 2 approved]

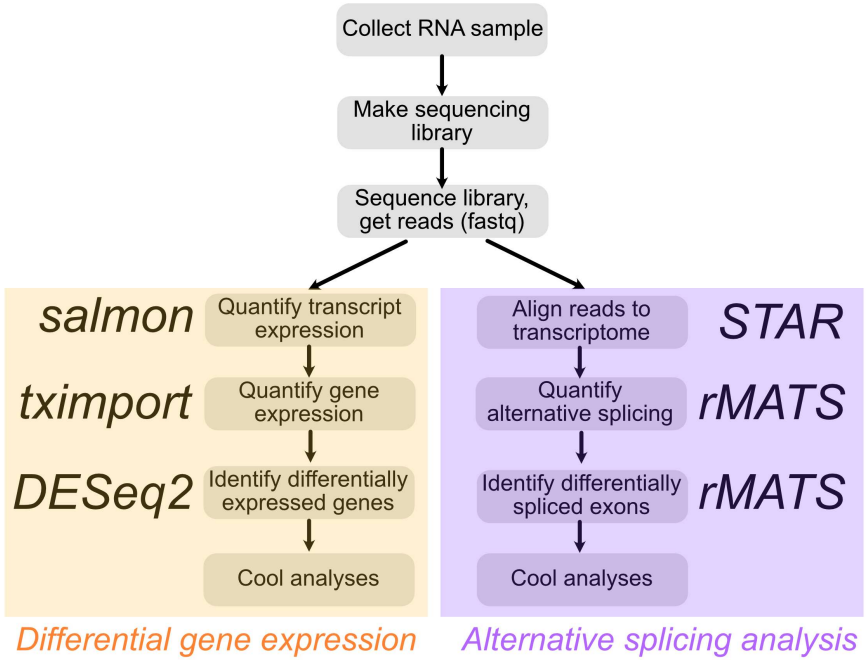
Kyle S Hubbard, Ian M Gut, Megan E Lyman, Patrick M McNutt

United States Army, Medical Research Institute of Chemical Defense, MD, 21010, USA

Overview

Cells were grown in generic differentiation-promoting media (LIF-) for 8 days until aggreates were dissociated and replated in neuronal differentiation media. This day of replating was designated as *in vitro* day 0 (DIV0). The timepoints taken before this replating therefore happened at "negative" times (DIV-8 and DIV-4). Because naming files with dashes or minus signs can cause problems, these samples are referred to as DIVminus8 and DIVminus4. Following the replating, samples were taken at days 1, 7, 16, 21, and 28 (DIV1, DIV7, DIV16, DIV21, and DIV28).

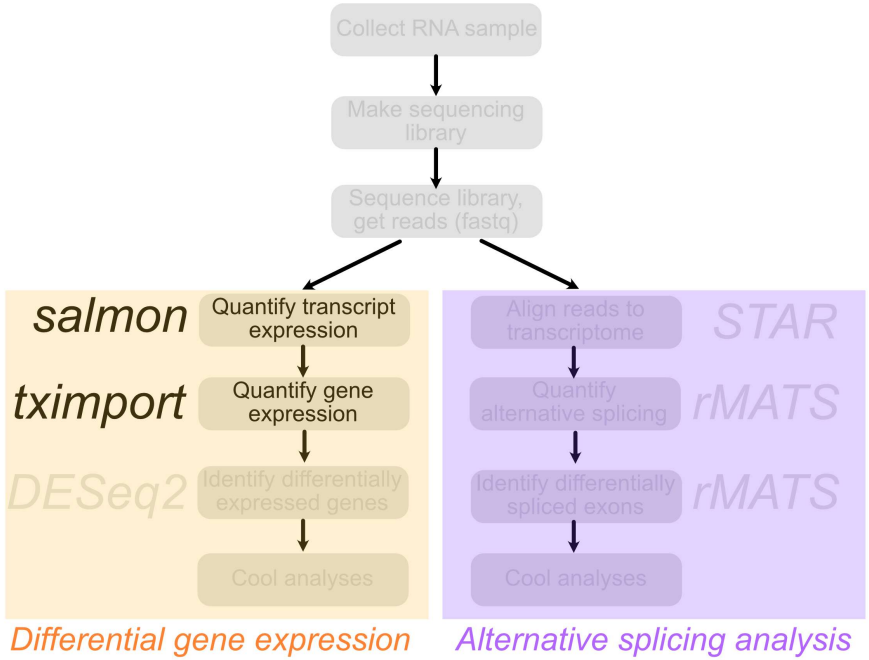
Today we will focus on some Quality Control steps that are good ideas to do for every RNAseq dataset you encounter, whether produced by yourself or someone else.



Overview

Cells were grown in generic differentiation-promoting media (LIF-) for 8 days until aggreates were dissociated and replated in neuronal differentiation media. This day of replating was designated as *in vitro* day 0 (DIV0). The timepoints taken before this replating therefore happened at "negative" times (DIV-8 and DIV-4). Because naming files with dashes or minus signs can cause problems, these samples are referred to as DIVminus8 and DIVminus4. Following the replating, samples were taken at days 1, 7, 16, 21, and 28 (DIV1, DIV7, DIV16, DIV21, and DIV28).

Today we will focus on some Quality Control steps that are good ideas to do for every RNAseq dataset you encounter, whether produced by yourself or someone else.



Quantifying transcript expression with salmon

We recently learned about the RNAseq quantification tool salmon. We won't rehash the details here about how salmon works. For our purposes, we just need to know that salmon reads in a fastq file of sequencing reads and a fasta file of transcript sequences to be quantified. Let's take a look at this fasta file of transcripts:

```
head -n 20 data/gencodecomprehensive.vM17.allc
```

```
#> >ENSMUST00000119854
#> CAAAAGCAACCAGACGGCCTACGTCCCGGCGCCTCTGAGAAAC
#> >ENSMUST00000147408
#> GAGTGGCTGACACGGGATATTGGAGTAACAAGACTTGGGCTGC
#> >ENSMUST00000070080
#> GGCGCTCCAGGGGCTAGGGCTCTTGGTTCCTCTTCTCCGGGCC
#> >ENSMUST00000033908
#> GGCCGCGGCCCGGGCCGCTCCGAGGTGATCGCGCGCGGCGCCC
#> >ENSMUST00000174016
#> CGGACGGAGCAGAACGGAAGCAGAGGGACAAAGTAGCGGATTC
```

Looks like we have ensembl transcript IDs, which is a good idea. I can tell because they start with 'ENS'. Using ensembl IDs as transcript names will allow us to later collate transcript expression levels into gene expression levels using a database that relates transcripts and genes. More on that later.

Making a transcriptome index

The first step in quantifying these transcripts is to make an index from them. This is done as follows:

```
salmon index -t <transcripts.fa> -i <transcripts.idx> --type quasi -k <k>
```

Here, transcripts.fa is a path to our fasta file, transcripts.idx is the name of the index that will be created, and k is the length of the kmers that will be used in the hash table related kmers and transcripts. k is the length of the minimum accepted match for a kmer in a read and a kmer in a transcript. Longer kmers (higher values of k) will therefore be more stringent, and lowering k may improve mapping sensitivity at the cost of some specificity. You may also see here how read lengths can influence what value for k you should choose.

Consider an experiment where we had 25 nt reads (this was true wayyyyyy back in the old, dark days of high-throughput sequencing). What's going to happen if I quantify these reads using an index where the kmer size was set to 29? Well, nothing will align. The index has represented the transcriptome in 29 nt chunks. However, no read will match to these 29mers because there are no 29mers in these reads! *As a general rule of thumb, for reads 75 nt and longer (which is the bulk of the data produced nowadays), a good value for k that maximizes both specificity and sensitivity is 31*. However, datasets that you may retrieve from the internet, particularly older ones, may have shorter read lengths, so keep this is mind when defining k.

Quantifying reads against this index

Once we have our index, we can quantify transcripts in the index using reads from our fastq files.

```
salmon quant --libType A -p 8 --seqBias --gcBias --validateMappings -1 <forwardreads.fastq> -2 <rev
```

In this command, our forward and reverse read fastq files are supplied to -1 and -2, respectively. If the experiment produced single end reads, -2 is omitted. is the path to the index produced in the previous step. I'm not going to go through the rest of the flags used here, but their meanings as well as other options can be found here.

Salmon outputs

Let's take a look at what salmon spits out. The first file we will look at is a log that is found at **/logs/salmon_quant.log**. This file contains info about the quantification, but there's one line of this file in particular that we are interested in. It lets us know how many of the reads in the fastq file that salmon found a home for in the transcriptome fasta.

```
less data/salmonouts/DIV0.Rep1/logs/salmon_quant.log
```

```
#> [2020-06-11 10:28:27.623] [jointLog] [info] Fragment incompatibility prior below threshold.  In
#> [2020-06-11 10:28:27.624] [jointLog] [info] Usage of --validateMappings implies use of range fa
#> [2020-06-11 10:28:27.624] [jointLog] [info] Usage of --validateMappings implies a default conse
#> [2020-06-11 10:28:27.624] [jointLog] [info] parsing read library format
#> [2020-06-11 10:28:27.624] [jointLog] [info] There is 1 library.
#> [2020-06-11 10:28:27.712] [jointLog] [info] Loading Quasi index
#> [2020-06-11 10:28:27.714] [jointLog] [info] Loading 32-bit quasi index
#> [2020-06-11 10:28:42.441] [jointLog] [info] done
#> [2020-06-11 10:28:42.441] [jointLog] [info] Index contained 133618 targets
#> [2020-06-11 10:28:44.299] [jointLog] [info] Automatically detected most likely library type as
#> [2020-06-11 10:30:05.185] [jointLog] [info] Thread saw mini-batch with a maximum of 0.88% zero
#> [2020-06-11 10:30:05.185] [jointLog] [info] Thread saw mini-batch with a maximum of 0.74% zero
#> [2020-06-11 10:30:05.185] [jointLog] [info] Thread saw mini-batch with a maximum of 0.80% zero
#> [2020-06-11 10:30:05.194] [jointLog] [info] Thread saw mini-batch with a maximum of 0.72% zero
#> [2020-06-11 10:30:05.203] [jointLog] [info] Thread saw mini-batch with a maximum of 0.74% zero
```

There are a lot of lines in this file, but really only one that we are interested in. We want the one that tells us the "Mapping rate." How co

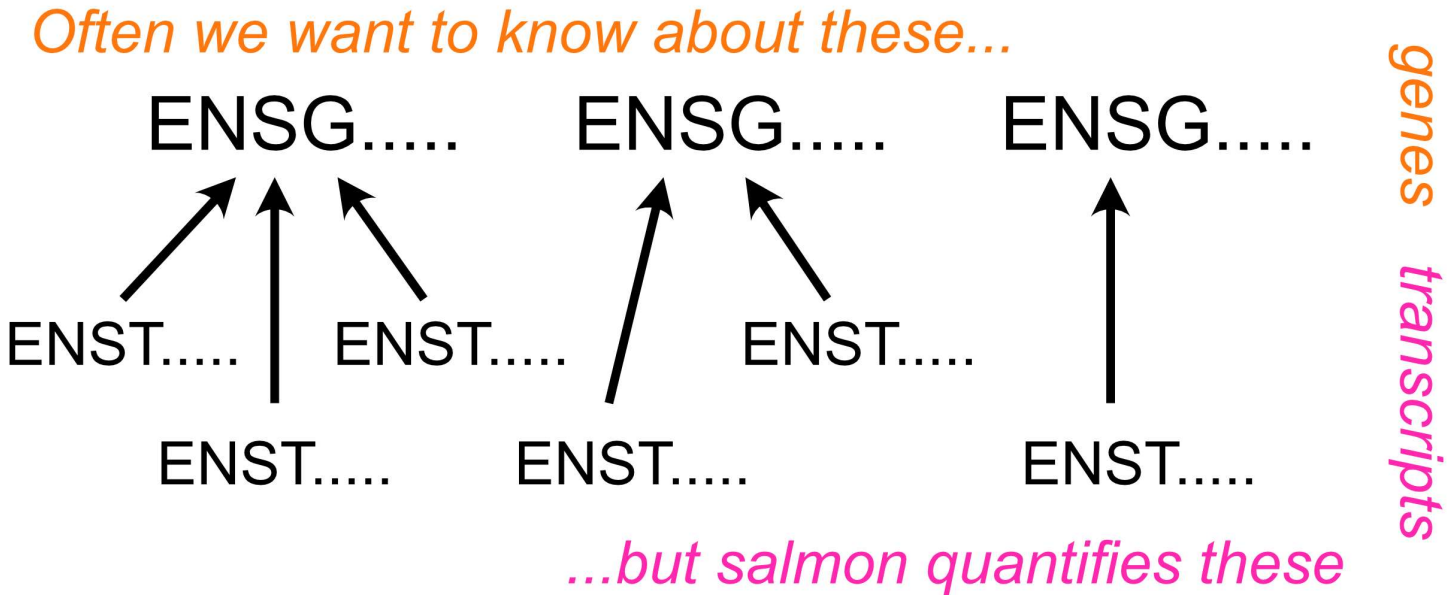
Salmon outputs

```
#Get the mapping rates for all samples
#In each log file, the line that we are interested in contains the string 'Mapping ' (notice the sp
grep 'Mapping ' data/salmonouts/*/logs/salmon_quant.log
```

```
#> data/salmonouts/DIV0.Rep1/logs/salmon_quant.log:[2020-06-11 10:30:05.575] [jointLog] [info] Mapp
#> data/salmonouts/DIV0.Rep2/logs/salmon_quant.log:[2020-06-11 10:33:26.879] [jointLog] [info] Mapp
#> data/salmonouts/DIV0.Rep3/logs/salmon_quant.log:[2020-06-11 10:37:13.255] [jointLog] [info] Mapp
#> data/salmonouts/DIV1.Rep1/logs/salmon_quant.log:[2020-06-11 10:41:37.780] [jointLog] [info] Mapp
#> data/salmonouts/DIV1.Rep2/logs/salmon_quant.log:[2020-06-11 10:45:18.193] [jointLog] [info] Mapp
#> data/salmonouts/DIV1.Rep3/logs/salmon_quant.log:[2020-06-11 10:48:33.355] [jointLog] [info] Mapp
#> data/salmonouts/DIV1.Rep4/logs/salmon_quant.log:[2020-06-11 10:51:45.439] [jointLog] [info] Mapp
#> data/salmonouts/DIV16.Rep1/logs/salmon_quant.log:[2020-06-11 11:14:36.361] [jointLog] [info] Map
#> data/salmonouts/DIV16.Rep2/logs/salmon_quant.log:[2020-06-11 11:18:16.602] [jointLog] [info] Map
#> data/salmonouts/DIV16.Rep3/logs/salmon_quant.log:[2020-06-11 11:22:21.462] [jointLog] [info] Map
#> data/salmonouts/DIV16.Rep4/logs/salmon_quant.log:[2020-06-11 11:26:13.888] [jointLog] [info] Map
#> data/salmonouts/DIV21.Rep1/logs/salmon_quant.log:[2020-06-11 11:30:06.704] [jointLog] [info] Map
#> data/salmonouts/DIV21.Rep2/logs/salmon_quant.log:[2020-06-11 11:34:10.867] [jointLog] [info] Map
#> data/salmonouts/DIV21.Rep3/logs/salmon_quant.log:[2020-06-11 11:38:15.799] [jointLog] [info] Map
#> data/salmonouts/DIV21.Rep4/logs/salmon_quant.log:[2020-06-11 11:42:41.865] [jointLog] [info] Map
#> data/salmonouts/DIV28.Rep1/logs/salmon_quant.log:[2020-06-11 11:46:53.420] [jointLog] [info] Map
#> data/salmonouts/DIV28.Rep2/logs/salmon_quant.log:[2020-06-11 11:50:52.802] [jointLog] [info] Map
#> data/salmonouts/DIV28.Rep3/logs/salmon_quant.log:[2020-06-11 11:54:37.736] [jointLog] [info] Map
#> data/salmonouts/DIV28.Rep4/logs/salmon_quant.log:[2020-06-11 11:58:17.045] [jointLog] [info] Map
```

Moving from transcript quantifications to gene quantifications

As we discussed, salmon quantifies *transcripts*, not *genes*. However, genes are made up of transcripts, so we can calculate gene expression values from transcript expression values if we knew which transcripts belonged to which genes.



Relating genes and transcripts

We can get this relationships between *transcripts* and *genes* through `biomaRt`.

`biomaRt` has many tables that relate genes, transcripts, and other useful data include gene biotypes and gene ontology categories, even across species. Let's use it here to get a table of genes and transcripts for the mouse genome.

```
#Load biomaRt
library(biomaRt)

#First we need to define a 'mart' to use.
#There are a handful of them that
#you can see here:
listMarts(mart = NULL,
          host = 'www.ensembl.org')
```

```
#>                  biomart                vers
#> 1 ENSEMBL_MART_ENSEMBL       Ensembl Genes 1
#> 2   ENSEMBL_MART_MOUSE       Mouse strains 1
#> 3     ENSEMBL_MART_SNP  Ensembl Variation 1
#> 4 ENSEMBL_MART_FUNCGEN Ensembl Regulation 1
```

I encourage you to see what is in each mart, but for now we are only going to use ENSEMBL_MART_ENSEMBL.

```
mart <- biomaRt::useMart("ENSEMBL_MART_ENSEMBL", host='www.ensembl.org')
```

Using biomaRt

Alright, we've chosen our mart. What datasets are available in this mart?

```
library(knitr)
datasets <- listDatasets(mart)

kable(datasets)
```

dataset	description
acalliptera_gene_ensembl	Eastern happy genes (fAst
acarolinensis_gene_ensembl	Anole lizard genes (AnoC
acchrysaetos_gene_ensembl	Golden eagle genes (bAqu
acitrinellus_gene_ensembl	Midas cichlid genes (Mida

A lot of stuff for a lot of species! Perhaps we want to limit it to see which ones are relevant to mouse.

```
mousedatasets <- filter(datasets, grepl('mmusc
kable(mousedatasets)
```

dataset	description	version
	Mouse genes	

Using biomaRt

Ah so we probably want the dataset called 'mmusculus_gene_ensembl'!

```
mart <- biomaRt::useMart("ENSEMBL_MART_ENSEMBL",
                         dataset = "mmusculus_gene_ensembl", host='www.ensembl.org')
```

Using biomaRt

OK what goodies are in this dataset?

```
goodies <- listAttributes(mart)
kable(goodies)
```

name	description	page
ensembl_gene_id	Gene stable ID	feature_page
ensembl_gene_id_version	Gene stable ID version	feature_page
ensembl_transcript_id	Transcript stable ID	feature_page
ensembl_transcript_id_version	Transcript stable ID version	feature_page
ensembl_peptide_id	Protein stable ID	feature_page
ensembl_peptide_id_version	Protein stable ID version	feature_page
ensembl_exon_id	Exon stable ID	feature_page

Using biomaRt

So there are 2885 rows of goodies about the mouse genome and its relationship to *many* other genomes. However, you can probably see that the ones that are most useful to us right now are right at the top: 'ensembl_transcript_id' and 'ensembl_gene_id'. We can use those attributes in our mart to make a table relating genes and transcripts.

I'm going to through one more attribute in: external_gene_name. Those are usually more informative than ensembl IDs.

```
t2g <- biomaRt::getBM(attributes = c('ensembl_
kable(t2g[1:20,])
```

ensembl_transcript_id	ensembl_gene_id	extern
ENSMUST00000082423	ENSMUSG00000064372	mt-Tp
ENSMUST00000082422	ENSMUSG00000064371	mt-Tt
ENSMUST00000082421	ENSMUSG00000064370	mt-Cy
ENSMUST00000082420	ENSMUSG00000064369	mt-Te
ENSMUST00000082419	ENSMUSG00000064368	mt-Nd
ENSMUST00000082418	ENSMUSG00000064367	mt-Nd
ENSMUST00000082417	ENSMUSG00000064366	mt-Tl2
ENSMUST00000082416	ENSMUSG00000064365	mt-Ts2

Using biomaRt

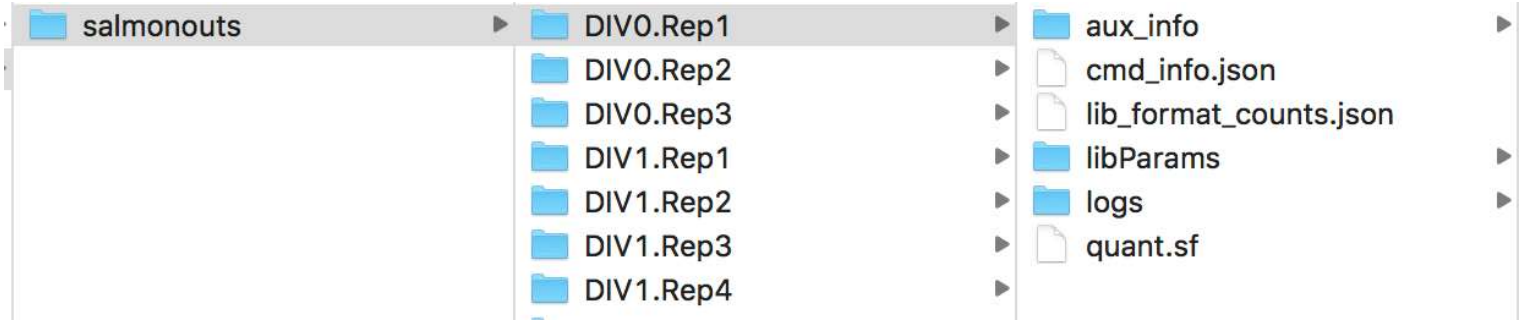
Alright this looks good! We are going to split this into two tables. One that contains transcript ID and gene ID, and the other that contains gene ID and gene name.

```
geneid2name <- dplyr::select(t2g, c(ensembl_gene_id, ensembl_transcript_id))
t2g <- dplyr::select(t2g, c(ensembl_transcript_id, ensembl_gene_id))
```

Getting gene level expression data with `tximport`

Now that we have our table relating transcripts and genes, we can give it to tximport to have it calculate gene-level expression data from our transcript-level expression data.

First, we have to tell it where the salmon quantification files (the quant.sf files) are. Here's what our directory structure that contains these files looks like:



Gene expression data with `tximport`

```
#The directory where all of the sample-specific salmon subdirectories live
base_dir <- 'data/salmonouts/'

#The names of all the sample-specific salmon subdirectories
sample_ids <- c('DIVminus8.Rep1', 'DIVminus8.Rep2', 'DIVminus8.Rep3', 'DIVminus8.Rep4',
                'DIVminus4.Rep1', 'DIVminus4.Rep2', 'DIVminus4.Rep3',
                'DIV0.Rep1', 'DIV0.Rep2', 'DIV0.Rep3',
                'DIV1.Rep1', 'DIV1.Rep2', 'DIV1.Rep3', 'DIV1.Rep4',
                'DIV7.Rep1', 'DIV7.Rep2', 'DIV7.Rep3', 'DIV7.Rep4',
                'DIV16.Rep1', 'DIV16.Rep2', 'DIV16.Rep3', 'DIV16.Rep4'
```

```
#>                              DIVminus8.Rep1
#> "data/salmonouts//DIVminus8.Rep1/quant.sf"
#>                              DIVminus8.Rep2
#> "data/salmonouts//DIVminus8.Rep2/quant.sf"
#>                              DIVminus8.Rep3
#> "data/salmonouts//DIVminus8.Rep3/quant.sf"
#>                              DIVminus8.Rep4
#> "data/salmonouts//DIVminus8.Rep4/quant.sf"
#>                              DIVminus4.Rep1
#> "data/salmonouts//DIVminus4.Rep1/quant.sf"
```

Gene expression data with `tximport`

You can see that we got a list of sample names and the absolute path to each sample's quantification file.

Now we are ready to run `tximport`! `tximport` is going to want paths to all the quantification files (salm_dirs) and a table that relates transcripts to genes (t2g). Luckily, we happen to have those exact two things.

```
library(tximport)
txi <- tximport(salm_dirs, type = 'salmon', tx2gene = t2g, dropInfReps = TRUE, countsFromAbundance
```

Gene expression data with `tximport`

Notice how we chose *lengthscaledTPM* for our abundance measurement. This is going to give us TPM values (transcripts per million) for expression in the $abundance slot. Let's check out what we have now.

```
tpms <- txi$abundance %>%
  as.data.frame(.) %>%
  rownames_to_column(var = 'ensembl_gene_id')

kable(tpms[1:50,])
```

ensembl_gene_id	DIVminus8.Rep1	DIVminus8.Rep2	DIVminus8.Rep3	DIVminus8.Rep4	DIVminus4.Rep1	DIV
ENSMUSG00000000001	92.799612	93.563524	95.500337	101.951963	106.634101	
ENSMUSG00000000003	0.000000	0.000000	0.000000	0.000000	0.000000	
ENSMUSG00000000028	59.718765	57.981259	57.935839	41.663343	45.614077	
ENSMUSG00000000031	0.177770	0.215726	0.320948	36.535428	0.914481	

TPM as an expression metric

Alright, not bad!

Let's stop and think for a minute about what `tximport` did and the metric we are using (TPM). What does *transcripts per million* mean? Well, it means pretty much what it sounds like. For every million transcripts in the cell, X of them are this particular transcript. Importantly, this means when this TPM value was calculated from the number of *counts* a transcript received, this number had to be adjusted for both the total number of counts in the library and the length of a transcript.

If sample A had twice the number of total counts as sample B (i.e. was sequenced twice as deeply), then you would expect every transcript to have approximately twice the number of counts in sample A as it has in sample B. Similarly, if transcript X is twice as long as transcript Y, then you would expect that if they were equally expressed (i.e. the same number of transcript X and transcript Y molecules were present in the sample) that X would have approximately twice the counts that Y does. Working with expression units of TPM incorporates both of these normalizations.

So, if a TPM of X means that for every million transcripts in the sample that X of them were the transcript of interest, then the sum of TPM values across all species should equal one million, right?

Let's check and see if that's true.

TPM as an expression metric

```
sum(tpms$DIVminus8.Rep1)
sum(tpms$DIVminus8.Rep2)
sum(tpms$DIVminus8.Rep3)
```

```
#> [1] 995216.3
```

```
#> [1] 995244.1
```

```
#> [1] 995222.4
```

OK, not quite one million, but pretty darn close.

This notion that TPMs represent proportions of a whole also leads to another interesting insight into what `tximport` is doing here. If all transcripts belong to genes, then the TPM for a gene must be the sum of the TPMs of its transcripts. Can we verify that that is true?

TPM as an expression metric

```
#Redefine for clarity in comparisons
tpms.genes <- tpms

#Make a new tximport object, but this time instead
#of giving gene expression values, give transcript expression values
#This is controlled by the `txOut` argument
txi.transcripts <- tximport(salm_dirs, type = 'salmon', tx2gene = t2g, dropInfReps = TRUE,
                            countsFromAbundance = 'lengthScaledTPM', txOut = TRUE)

#Make a table of tpm values for every transcript
```

ensembl_transcript_id	ensembl_gene_id	DIVminus8.Rep1	DIVminus8.Rep2	DIVminus8.Rep3	DIVminus8.Rep4
ENSMUST00000082423	ENSMUSG00000064372	214.839964	199.081786	152.871667	270.905168
ENSMUST00000082422	ENSMUSG00000064371	16.850193	6.221307	0.000000	3.954820
ENSMUST00000082421	ENSMUSG00000064370	1426.656406	1506.843160	1505.511957	1130.763861
ENSMUST00000082420	ENSMUSG00000064369	4.044046	7.949446	14.048138	3.790036

OK so lets look at the expression of ENSMUSG00000020634 in the first sample (DIVminus8.Rep1).

TPM as an expression metric

```
#Get sum of tpm values for transcripts that belong to ENSMUSG00000020634
tpms.tx.ENSMUSG00000020634 <- filter(tpms.txs, ensembl_gene_id == 'ENSMUSG00000020634')
sumoftxtpm <- sum(tpms.tx.ENSMUSG00000020634$DIVminus8.Rep1)

#Get gene level tpm value of ENSMUSG00000020634
genetpm <- filter(tpms.genes, ensembl_gene_id == 'ENSMUSG00000020634')$DIVminus8.Rep1

#Are they the same?
sumoftxtpm
genetpm
```

```
#> [1] 46.71414
```

```
#> [1] 46.71414
```

Basic RNAseq QC

OK now that we've got expression values for all genes, we now might want to use these expression values to learn a little bit about our samples. One simple question is

> Are replicates similar to each other, or at least more similar to each other than to other samples?

If our data is worth anything at all, we would hope that differences between replicates, which are supposed to be drawn from the same condition, are smaller than differences between samples drawn from different conditions. If that's not true, it could indicate that one replicate is very different from other replciates (in which case we might want to remove it), or that the data in general is of poor quality.

Another question is

> How similar is each sample to every other sample?

In our timecourse, we might expect that samples drawn from adjacent timepoints might be more similar to each other than samples from more distant timepoints.

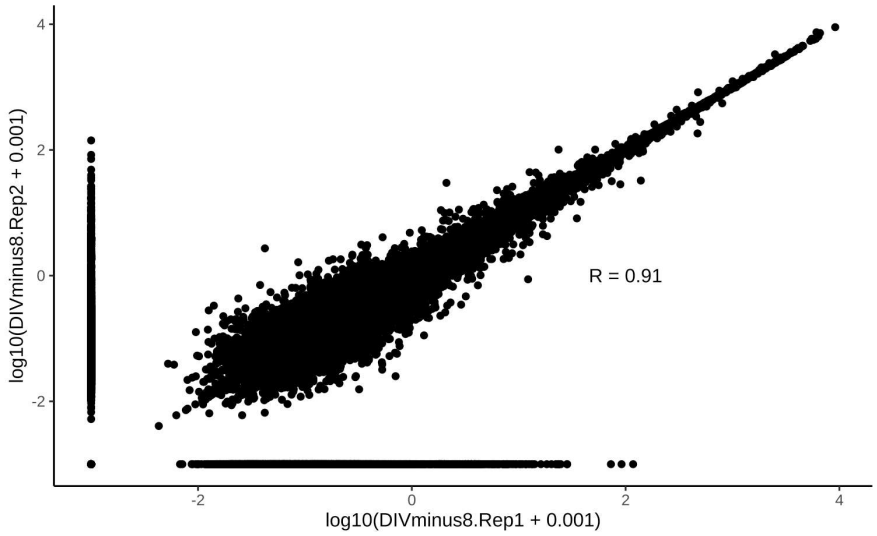
Hierarchical clustering

A simple way to think about this is to simply correlate TPM values for genes between samples. For plotting purposes here, let's plot the log(TPM) of two samples against each other. However, for the actual correlation coefficient we are going to be using the *Spearman* correlation method, which uses ranks, not absolute values. This means that whether or not you take the log will have no effect on the Spearman correlation coefficient.

```
#DIVminus8.Rep1 vs DIVminus8.Rep2

#Since we are plotting log TPM values, we need
#log(0) is a problem.

#Add pseudocounts and take log within ggplot f
r.spearman <- cor.test(tpms$DIVminus8.Rep1, tpm
                       method = 'spearman')$es
r.spearman <- signif(r.spearman, 2)
ggplot(tpms, aes(x = log10(DIVminus8.Rep1 + 1e
  geom_point() + theme_classic() +
  annotate('text', x = 2, y = 0, label = paste(
```



Hierarchical clustering

With RNAseq data, the variance of a gene's expression increases as the expression increases. However, using a pseudocount and taking the log of the expression value actually reverses this trend. Now, genes with the lowest expression have the most variance. Why is this a problem? Well, the genes with the most variance are going to be the ones that contribute the most to intersample differences. Ideally, we would like to therefore remove the relationship between expression and variance.

There are transformations, notably `rlog` and `vst`, that are made to deal with this, but they are best used when dealing with normalized **count** data, while here we are dealing with TPMs. We will talk about counts later, but not here.

So, for now, we will take another approach of simply using an expression threshold. Any gene that does not meet our threshold will be excluded from the analysis. Obviously where to set this threshold is a bit subjective. For now, we will set this cutoff at 1 TPM.

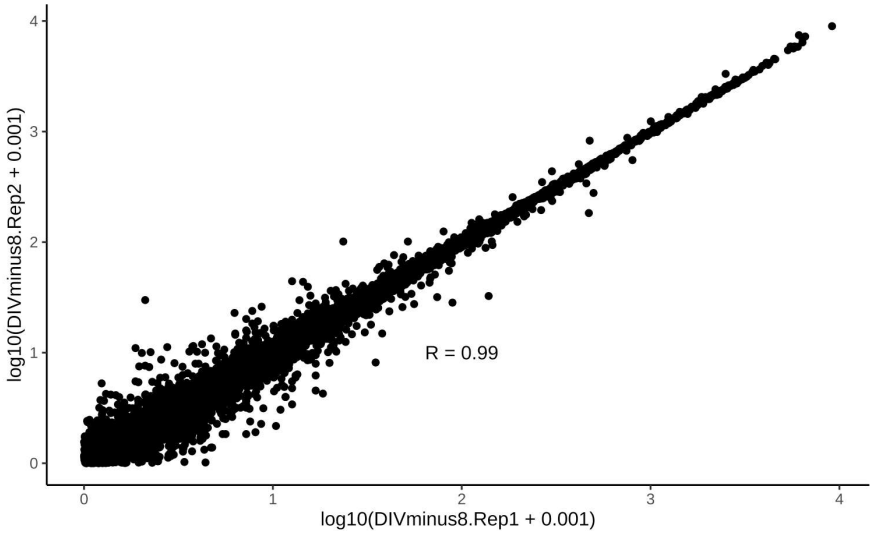
Hierarchical clustering

```
#DIVminus8.Rep1 vs DIVminus8.Rep2

#Since we are plotting log TPM values, we need
#log(0) is a problem.

#Filter for genes that have expression of at l
tpms.2samplecor <- dplyr::select(tpms, c(ensem
  filter(., DIVminus8.Rep1 >= 1 & DIVminus8.Re

#Add pseudocounts and take log within ggplot f
r.spearman <- cor.test(tpms.2samplecor$DIVminu
                       tpms.2samplecor$DIVminu
r.spearman <- signif(r.spearman, 2)
ggplot(tpms.2samplecor, aes(x = log10(DIVminus
  geom_point() + theme_classic() +
  annotate('text', x = 2, y = 1, label = paste
```



Filtering lowly expressed genes

OK that's two samples compared to each other, but now we want to see how **all** samples compare to **all** other samples. Before we do this we need to decide how to apply our expression cutoff across many samples. Should a gene have to meet the cutoff in only one sample? In all samples? Let's start by saying it has to meet the cutoff in at least half of the 30 samples.

```
#Make a new column in tpms that is the number of samples in which the value is at least 1
tpms.cutoff <- mutate(tpms, nSamples = rowSums(tpms[,2:31] > 1))%>%
  #Now filter for rows where nSamples is at least 15
  #Meaning that at least 15 samples passed the threshold
  filter(., nSamples >= 15) %>%
  #Get rid of the nSamples column
  dplyr::select(., -nSamples)

nrow(tpms)
nrow(tpms.cutoff)
```

```
#> [1] 52346
```

```
#> [1] 15061
```

Correlating gene expression values

Now we can use the `cor` function to calculate pairwise correlations in a *matrix* of TPM values.

```
tpms.cutoff.matrix <- dplyr::select(tpms.cutoff, -ensembl_gene_id) %>%
  as.matrix(.)

tpms.cor <- cor(tpms.cutoff.matrix, method = 'spearman')
head(tpms.cor)
```

```
#>                DIVminus8.Rep1 DIVminus8.Rep2 DIVminus8.Rep3 DIVminus8.Rep4
#> DIVminus8.Rep1      1.0000000      0.9874291      0.9874863      0.9656629
#> DIVminus8.Rep2      0.9874291      1.0000000      0.9887704      0.9663987
#> DIVminus8.Rep3      0.9874863      0.9887704      1.0000000      0.9672342
#> DIVminus8.Rep4      0.9656629      0.9663987      0.9672342      1.0000000
#> DIVminus4.Rep1      0.9613798      0.9618754      0.9611926      0.9386769
#> DIVminus4.Rep2      0.9609581      0.9599379      0.9609036      0.9377638
#>                DIVminus4.Rep1 DIVminus4.Rep2 DIVminus4.Rep3 DIV0.Rep1 DIV0.Rep2
#> DIVminus8.Rep1      0.9613798      0.9609581      0.9618412 0.7926824 0.7998363
#> DIVminus8.Rep2      0.9618754      0.9599379      0.9614563 0.7948024 0.8015281
#> DIVminus8.Rep3      0.9611926      0.9609036      0.9613304 0.7943499 0.7994011
#> DIVminus8.Rep4      0.9386769      0.9377638      0.9371555 0.7821715 0.7871162
```

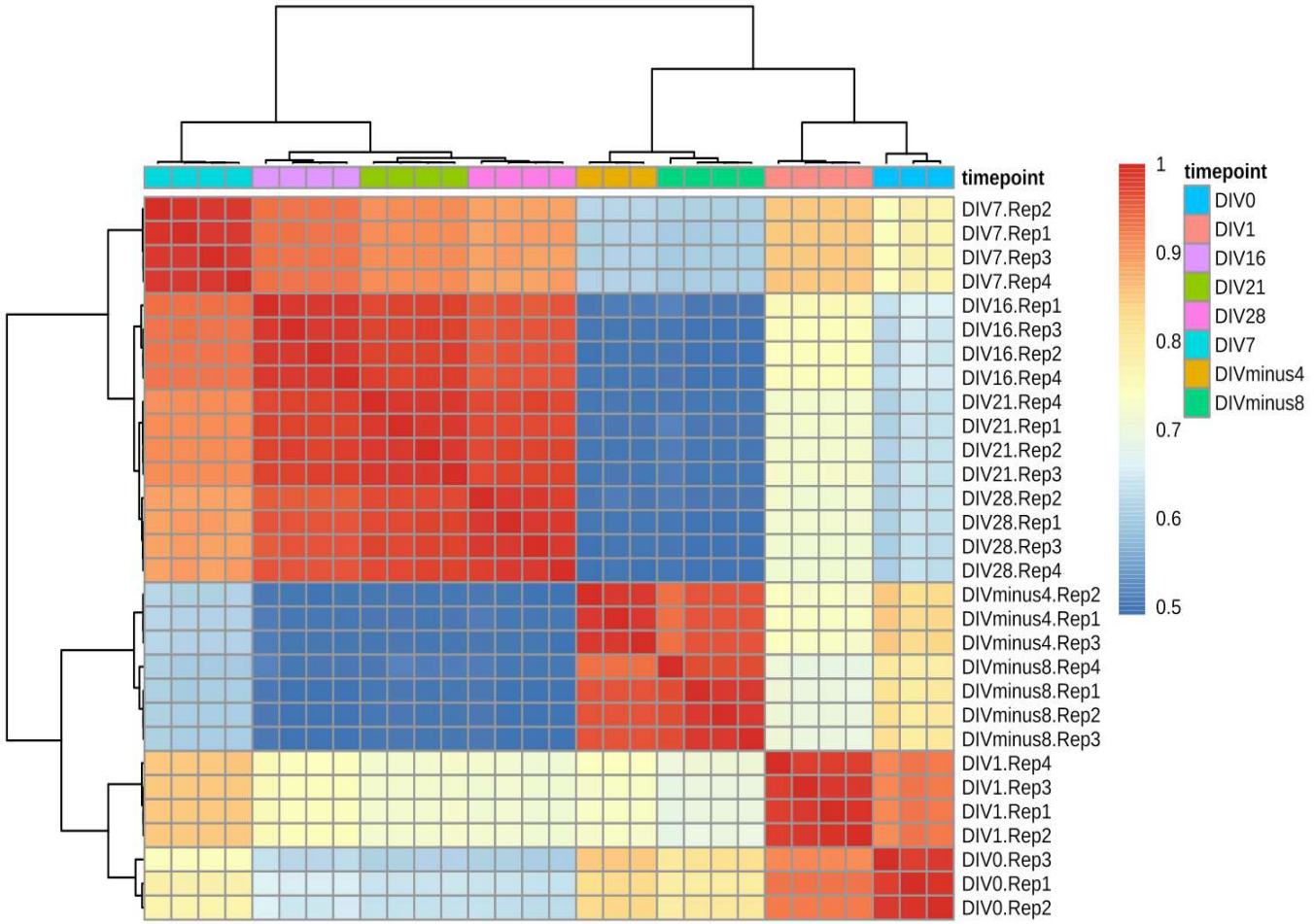
Hierarchical clustering

Now we need to plot these and have similar samples (i.e. those that are highly correlated with each other) be clustered near to each other. We will use `pheatmap` to do this.

```
library(pheatmap)

#Make a dataframe of annotations
annot <- data.frame(row.names = colnames(tpms.cor), timepoint = c(rep('DIVminus8', 4), rep('DIVminu
                                                                  rep('DIV0', 3), rep('DIV1', 4), r
                                                                  rep('DIV16', 4), rep('DIV21', 4),
pheatmap(tpms.cor, annotation_col = annot, fontsize = 7, show_colnames = F)
```

This looks pretty good! There are two main points to takeaway here. First, all replicates for a given timepoint are clustering with each other. Second, you can kind of derive the order of the timepoints from the clustering. The biggest separation is between early (DIVminus8 to DIV1) and late (DIV7 to DIV28). After that you can then see finer-grained structure.



PCA analysis

Another way to visualize relationships is using a dimensionality reduction technique called Principal Component Analysis (PCA). Let's watch this short video. It focuses more on how to interpret them rather than the math behind their creation.

PCA analysis

PCA works best when values are approximately normally distributed, so we will first take the log of our expression values.

With our cutoff as it is now (genes have to have expression of at least 1 TPM in half the samples), it is possible that we will have some 0 values. Taking the log of 0 might cause a problem, so we will add a pseudocount.

```
tpms.cutoff.matrix <- dplyr::select(tpms.cutof
  as.matrix(.)

#Add pseudocount
tpms.cutoff.matrix <- tpms.cutoff.matrix + 1e-
#Take log of values
tpms.cutoff.matrix <- log(tpms.cutoff.matrix)

tpms.cutoff.matrix
```

```
#>          DIVminus8.Rep1 DIVminus8.Rep2 DIVr
#>     [1,]   4.5304532346    4.538651295   4
#>     [2,]   4.0896630376    4.060137084   4
#>     [3,]  -1.7216552156   -1.529121396  -1
#>     [4,]   1.1596716395    0.919547423   0
#>     [5,]   2.3676627500    2.367244294   2
#>     [6,]   2.9791393187    2.969073824   3
#>     [7,]   2.6278618217    2.459623969   2
#>     [8,]   6.5735690616    6.583466635   6
#>     [9,]  -2.0820689904   -2.724408268  -2
#>    [10,]   1.0010023999    0.981178442   1
#>    [11,]  -1.7739234804   -1.313698389  -1
#>    [12,]   2.5586127853    2.519576076   2
#>    [13,]   4.2771333987    4.278469714   4
#>    [14,]   4.1385373492    4.180937677   4
```

PCA analysis

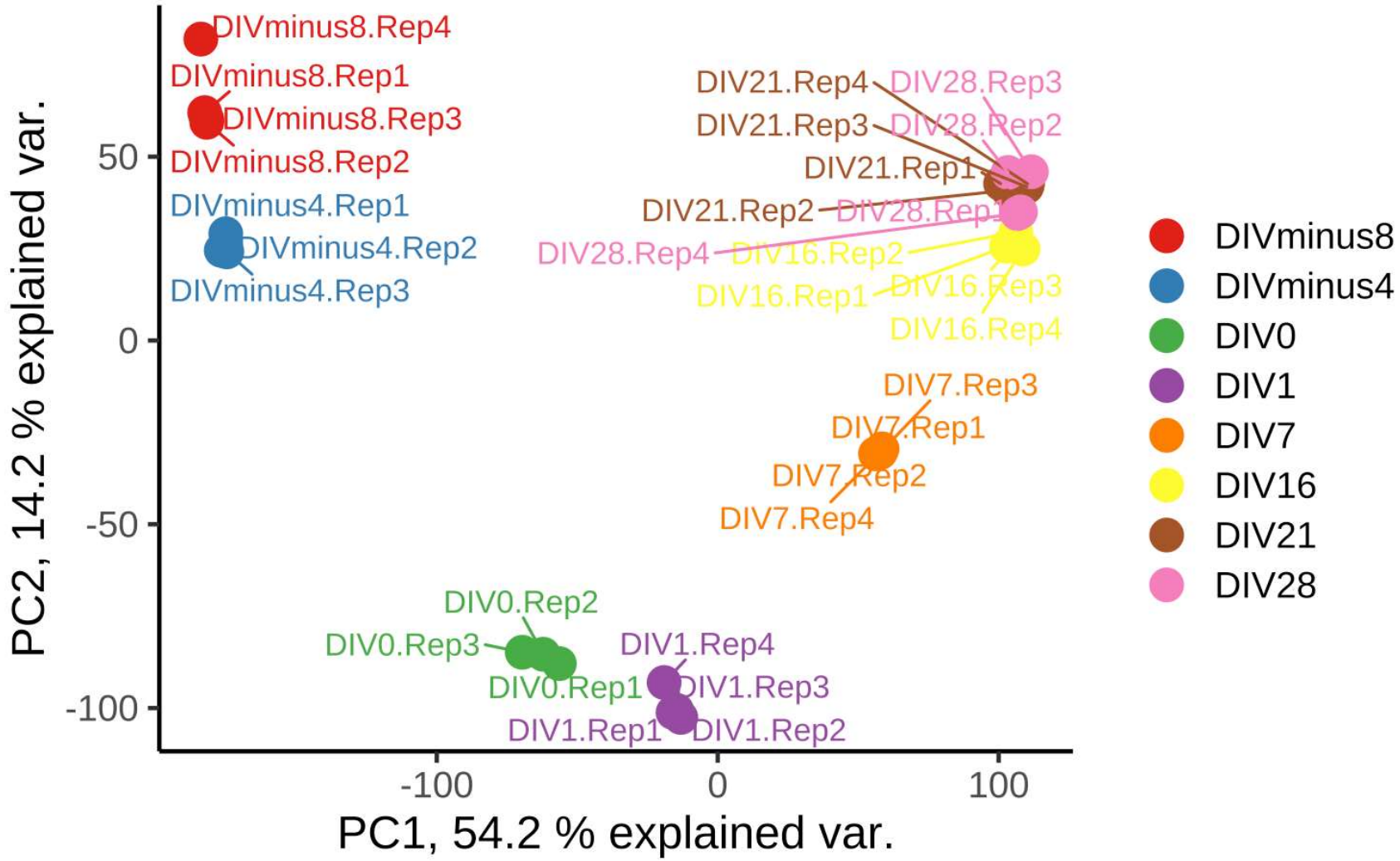
OK now we are ready to give this matrix to R's `prcomp` function to find principal components.

```
library(RColorBrewer)
library(ggrepel)
#prcomp expects samples to be rownames, right now they are columns
#so we need to transpose the matrix using `t`
tpms.pca <- prcomp(t(tpms.cutoff.matrix))

#The coordinates of samples on the principle components are stored in the $x slot
#These are what we are going to use to plot
#We can also also some data about the samples here so that our plot is a little more interesting
tpms.pca.pc <- data.frame(tpms.pca$x) %>%
  mutate(., sample = colnames(tpms.cutoff.matrix)) %>%
  mutate(., timepoint = c(rep('DIVminus8', 4), rep('DIVminus4', 3),
                          rep('DIV0', 3), rep('DIV1', 4), rep('DIV7', 4),
                          rep('DIV16', 4), rep('DIV21', 4), rep('DIV28', 4)))

#We can see how much of the total variance is explained by each PC using the summary function
tpms.pca.summary <- summary(tpms.pca)$importance

#The amount of variance explained by PC1 is the second row, first column of this table
#It's given as a fraction of 1, so we multiply it by 100 to get a percentage
pc1var = round(tpms.pca.summary[2,1] * 100, 1)
```


DIVminus8.Rep4
DIVminus8.Rep1
DIVminus8.Rep3
DIVminus8.Rep2
DIVminus4.Rep1
DIVminus4.Rep2
DIVminus4.Rep3
DIV21.Rep4
DIV28.Rep3
DIV21.Rep3
DIV28.Rep2
DIV21.Rep1
DIV21.Rep2
DIV28.Rep1
DIV28.Rep4
DIV16.Rep2
DIV16.Rep1
DIV16.Rep3
DIV16.Rep4
DIV7.Rep3
DIV7.Rep1
DIV7.Rep2
DIV7.Rep4
DIV0.Rep2
DIV0.Rep3
DIV1.Rep4
DIV0.Rep1
DIV1.Rep3
DIV1.Rep1
DIV1.Rep2
50
0
-50
-100
-100
0
100
PC2, 14.2 % explained var.
PC1, 54.2 % explained var.
DIVminus8
DIVminus4
DIV0
DIV1
DIV7
DIV16
DIV21
DIV28

RNAseq: DE

Matthew Taliaferro

Contact Info

Greetings experimentalist humans 👋

matthew.taliaferro@cuanschutz.edu

Learning Objectives

By the end of the class, you should be able to:

- Identify genes that are differentially expressed between two samples using `DESeq2`
- Explore differential expression results qualitatively
- Plot the expression of single genes across conditions
- Investigate changes in expression for groups of genes based on their membership within gene ontology categories

Rigor & Reproducibility

As with all computational **experiments** (yes, they are experiments, don't let your pipette-toting friends tell you otherwise), keeping track of what you did is key. In the old days, I kept a written notebook of commands that I ran. Sounds silly, but there were many times that I went back to that notebook to see exactly what the parameters were for a given run using a piece of software.

Today, there are better options. You are using one of the better ones right now. Notebooks, including RMarkdown (mainly for R) and Jupyter (mainly for Python), are a great way to keep track of what you did as well as give justification or explanation for your analyses using plain 'ol English.

Trust me, one day you will be glad you used them. The Methods section of your paper is never fun to write without them.

Problem Set and Grading Rubric

Today's problem set is composed of 3 problems. The first one is worth 30% of the total points while the next two are worth 35% each. For all of these problems, you will be presented with an RNAseq dataset. Transcript quantification will have already been performed using `salmon`, and differential expression analysis will have been performed with `DESeq2`.

- In the first, you must take those results and plot the expression of a single gene across two different conditions.
- In the second, you will ask for differences in expression in a *group* of genes across two conditions.
- In the third, you will plot differences in expression for a group of genes to identify those that are differentially expressed.

Further reading

If you are interested, here is a little more information about today's topic that you can read later:

- A vignette describing the use of `DESeq2`
- The paper describing `DESeq2`'s method

Overview

In this class, we will examine RNAseq data collected over a timecourse of differentiation from mouse embryonic stem cells to cortical glutamatergic neurons (Hubbard et al, F1000 Research (2013)). In this publication, the authors differentiated mESCs to neurons using a series of *in vitro* culture steps over a period of 37 days. During this timecourse, samples were extracted at selected intervals for transcriptome analysis. Importantly, for each timepoint, either 3 or 4 samples were taken for RNA extraction, library preparation and sequencing. This allows us to efficiently use the statistical frameworks provided by the DESeq2 package to identify genes whose RNA expression changes across the timecourse.

DATA ARTICLE

Longitudinal RNA sequencing of the deep transcriptome during neurogenesis of cortical glutamatergic neurons from murine ESCs [version 1; peer review: 2 approved]

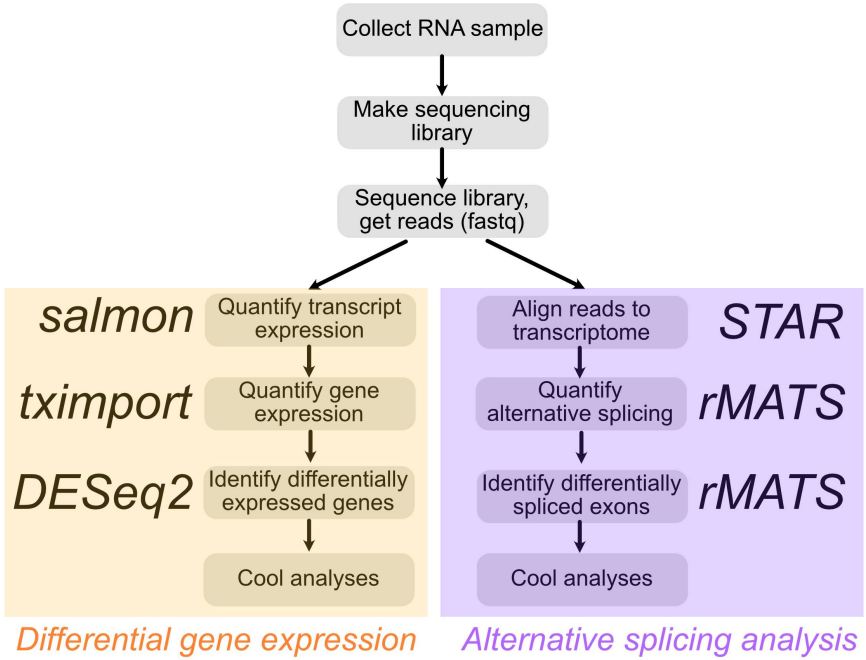
Kyle S Hubbard, Ian M Gut, Megan E Lyman, Patrick M McNutt

United States Army, Medical Research Institute of Chemical Defense, MD, 21010, USA

Overview

Cells were grown in generic differentiation-promoting media (LIF-) for 8 days until aggreates were dissociated and replated in neuronal differentiation media. This day of replating was designated as *in vitro* day 0 (DIV0). The timepoints taken before this replating therefore happened at "negative" times (DIV-8 and DIV-4). Because naming files with dashes or minus signs can cause problems, these samples are referred to as DIVminus8 and DIVminus4. Following the replating, samples were taken at days 1, 7, 16, 21, and 28 (DIV1, DIV7, DIV16, DIV21, and DIV28).

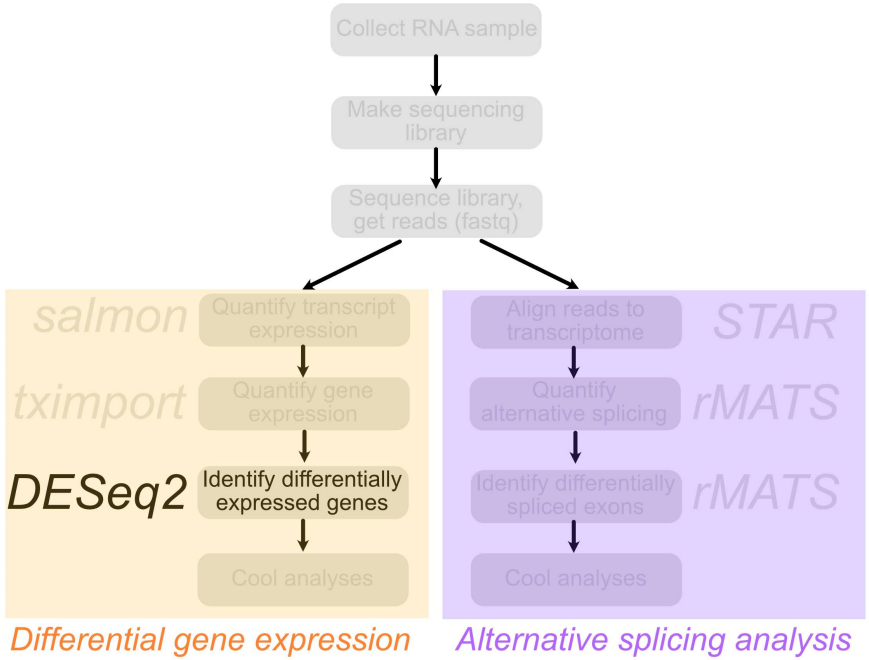
Today we will focus on identifying differentially expressed genes between two of these timepoints.



Overview

Cells were grown in generic differentiation-promoting media (LIF-) for 8 days until aggreates were dissociated and replated in neuronal differentiation media. This day of replating was designated as *in vitro* day 0 (DIV0). The timepoints taken before this replating therefore happened at "negative" times (DIV-8 and DIV-4). Because naming files with dashes or minus signs can cause problems, these samples are referred to as DIVminus8 and DIVminus4. Following the replating, samples were taken at days 1, 7, 16, 21, and 28 (DIV1, DIV7, DIV16, DIV21, and DIV28).

Today we will focus on identifying differentially expressed genes between two of these timepoints.



Overview

Last time we took RNAseq data from an *in vitro* differentiation timecouse from mouse ESCs to glutaminergic neurons (Hubbard et al, F1000 Research (2013)). We took in transcript-level quantifications produced by `salmon` and collapsed them to gene-level quantifications using `tximport`. We then inspected the quality of the data by relating distances between samples using two methods: hierarchical clustering and principal components analysis.

We found that the data was of high quality, as evidenced by the fact that replicates from a given timepoint were highly similar to other replicates for the same timepoint, and the distances between samples made sense with what we know about how the experiment was conducted.

Today, we are going to pretend that this isn't a timecourse. We are going to imagine that we have only two conditions: DIV0 and DIV7. We will use `DESeq2` to identify genes that are differentially expressed between these two timepoints.

We will then plot changes in expression for both individual genes and groups of genes that we already know going in might be interesting to look at. Finally, we will look at some features of transcripts and genes that are differentially expressed between these two timepoints. We are not doing this two-timepoint comparsion because it is necessarily a good thing to do with timepoint data. Instead, we are doing this because often in an RNAseq experiment, you only have two conditions to compare.

Identifying differentially expressed genes with DESeq2

The first thing we need to do is read in the data again and move from transcript-level expression values to gene-level expression values with `tximport`. Let's use `biomaRt` to get a table that relates gene and transcript IDs.

```
mart <- biomaRt::useMart("ENSEMBL_MART_ENSEMBL", dataset = "mmusculus_gene_ensembl", host='www.ense
t2g <- biomaRt::getBM(attributes = c('ensembl_transcript_id', 'ensembl_gene_id', 'external_gene_nam
```

Quantify gene expression with `tximport`

Now we can read in the transcript-level data and collapse to gene-level data with `tximport`

```
#The directory where all of the sample-specific salmon subdirectories live
base_dir <- 'data/salmonouts/'

#The names of all the sample-specific salmon subdirectories
sample_ids <- c('DIV0.Rep1', 'DIV0.Rep2', 'DIV0.Rep3',
                'DIV7.Rep1', 'DIV7.Rep2', 'DIV7.Rep3', 'DIV7.Rep4')

#So what we want to do now is create paths to each quant.sf file that is in each sample_id.
#This can be done by combining the base_dir, each sample_id directory, and 'quant.sf'
#For example, the path to the first file will be
#data/salmonouts/DIVminus8.Rep1/quant.sf
salm_dirs <- sapply(sample_ids, function(id) file.path(base_dir, id, 'quant.sf'))

#Run tximport
txi <- tximport(salm_dirs, type = 'salmon', tx2gene = t2g, dropInfReps = TRUE, countsFromAbundance
```

Quantify gene expression with `tximport`

Let's take a look at what `tximport` did just to make sure everything is 💯

```
kable(txi$abundance[1:20,])
```

	DIV0.Rep1	DIV0.Rep2	DIV0.Rep3	DIV7.Rep1	DIV7.Rep2	DIV7.Rep3	DIV7.Rep4
ENSMUSG00000000001	148.004511	145.539529	164.680183	76.088940	76.669016	79.505738	76.038479
ENSMUSG00000000003	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000
ENSMUSG00000000028	34.180035	36.215924	39.486489	6.582942	6.752942	5.547107	5.510259
ENSMUSG00000000031	7.213808	6.431925	13.253322	151.244212	137.227925	169.762968	150.599359
ENSMUSG00000000037	5.877237	5.001733	7.579149	2.750903	1.827190	2.446732	2.894751
ENSMUSG00000000049	0.000000	0.046208	0.000000	0.188254	0.050654	0.277837	0.189814
ENSMUSG00000000056	31.553583	28.610456	19.741875	41.828967	46.812515	44.172289	46.834670
ENSMUSG00000000058	0.420304	0.507728	0.602757	0.194399	0.102993	0.063154	0.130644

Identifying differentially expressed genes with DESeq2

OK we are going to need to give `DESeq2` information about the samples like which sample belongs to which timepoint.

```
samples <- data.frame(row.names = c('DIV0.Rep1', 'DIV0.Rep2', 'DIV0.Rep3',
                                    'DIV7.Rep1', 'DIV7.Rep2', 'DIV7.Rep3', 'DIV7.Rep4'),
                      timepoint = c(rep('DIV0', 3), rep('DIV7', 4)))

samples
```

```
#>           timepoint
#> DIV0.Rep1      DIV0
#> DIV0.Rep2      DIV0
#> DIV0.Rep3      DIV0
#> DIV7.Rep1      DIV7
#> DIV7.Rep2      DIV7
#> DIV7.Rep3      DIV7
#> DIV7.Rep4      DIV7
```

Design formulae

There are essentially two steps to using `DESeq2`. The first involves creating a `DESeqDataSet` from your data. Luckily, if you have a `tximport` object, which we do in the form of `txi`, then this becomes easy.

```
ddsTxi <- DESeqDataSetFromTximport(txi, colData = samples, design = ~ timepoint)
```

You can see that `DESeqDataSetFromTximport` wants three things. The **first** is our `tximport` object. The **second** is the dataframe we made that relates samples and conditions (or in this case timepoints). The **last** is something called a *design formula*. A design formula contains all of the variables that will go into `DESeq2`'s model. The formula starts with a tilde and then has variables separated by a plus sign. It is common practice, and in fact basically required with `DESeq2`, to put the variable of interest last. In our case, that's trivial because we only have one: timepoint. So our design formula is very simple:

```
design = ~ timepoint
```

Design formulae

Your design formula should ideally include **all of the sources of variation in your data**. For example, let's say that here we thought there was a batch effect with the replicates. Maybe all of the Rep1 samples were prepped and sequenced on a different day than the Rep2 samples and so on. We could potentially account for this in `DESeq2`'s model with the following formula:

```
design = ~ replicate + timepoint
```

Here, timepoint is still the variable of interest, but we are controlling for differences that arise due to differences in replicates. Of course, this formula would require us to go back and make a new `samples` table that included 'replicate' as a factor.

Making a DESeq dataset

OK so let's go ahead and make our dataset with `DESeq2`.

```
ddsTxi <- DESeqDataSetFromTximport(txi, colData = samples, design = ~ timepoint)

ddsTxi
```

```
#> class: DESeqDataSet
#> dim: 52346 7
#> metadata(1): version
#> assays(1): counts
#> rownames(52346): ENSMUSG00000000001 ENSMUSG00000000003 ...
#>   ENSMUSG00000118655 ENSMUSG00000118659
#> rowData names(0):
#> colnames(7): DIV0.Rep1 DIV0.Rep2 ... DIV7.Rep3 DIV7.Rep4
#> colData names(1): timepoint
```

We can see here that `DESeq2` is taking the counts produced by `tximport` for gene quantifications. There are 52346 genes (rows) here and 8 samples (columns). We learned earlier that `DESeq2` uses counts, not TPM, not FPKM, not anything else, when identifying differentially expressed genes. This is very important for `DESeq2`'s statistical model. We won't say much more about this here or get into the guts of `DESeq2`, instead focusing on running it. Although we have mainly looked at their TPM outputs thus far,

Running DESeq2

Now using this ddsTxi object, we can run DESeq2. Let's do it.

```
dds <- DESeq(ddsTxi)
```

There are many useful things in this dds object. I encourage you to take a look at the vignette for DESeq2 for a full explanation about what is in there, as well as info on many more tests and analyses that can be done with DESeq2.

Accessing results

The results can be accessed using the aptly named `results()` function.

We are going to include the `contrast` argument here. `DESeq2` reports changes in RNA abundance between two samples as a log2FoldChange. That's great, but it's often not clear exactly what the numerator and denominator of that fold change ratio is. In this case, it could be either DIV7/DIV0 or DIV0/DIV7.

In actuality, it's of course not random. The alphabetically first condition will be the numerator. Still, I find it less confusing to explicitly specify what the numerator and denominator of this ratio are using the `contrast` argument.

```
#For contrast, we give three strings: the factor we are interested in,
#the numerator, and the denominator
#It makes the most sense (to me at least) to have DIV7 be the numerator
results(dds, contrast = c('timepoint', 'DIV7', 'DIV0'))
```

Accessing results

Another useful application of the `contrast` argument can be seen with more complicated design formulae. Remember our design formula that accounted for potential differences due to replicate batch effects:

```
design = ~ replicate + timepoint
```

As we said before, with this formula, `DESeq2` will account for differences between replicates here to find differences between timepoints. However, what if we wanted to look at differences between replicate batches? By supplying 'replicate' to `contrast` instead of 'timepoint', we could extract difference between these batches.

Accessing results

OK, back to what we were doing. Let's take a look at what `results()` will give us.

```
#For contrast, we give three strings: the factor we are interested in,
#the numerator, and the denominator
#It makes the most sense (to me at least) to have DIV7 be the numerator
results(dds, contrast = c('timepoint', 'DIV7', 'DIV0'))
```

```
#> log2 fold change (MLE): timepoint DIV7 vs DIV0
#> Wald test p-value: timepoint DIV7 vs DIV0
#> DataFrame with 52346 rows and 6 columns
#>                          baseMean     log2FoldChange             lfcSE
#>                         <numeric>          <numeric>         <numeric>
#> ENSMUSG00000000001 5579.77754585436 -0.953512002710987 0.0527959783401513
#> ENSMUSG00000000003                0                 NA                NA
#> ENSMUSG00000000028 718.696881667605  -2.55365044138704  0.107061390183316
#> ENSMUSG00000000031 3624.57797113539   4.12075207975084  0.289850192996735
#> ENSMUSG00000000037 242.620316811254  -1.27535503985687  0.199245035001358
#> ...                             ...                ...                ...
#> ENSMUSG00000118642 327.352114985249  -3.25323665588743   0.23412080090127
#> ENSMUSG00000118643 0.41966839171392  -1.01604388048036   2.91302113145088
#> ENSMUSG00000118645                0                 NA                NA
#> ENSMUSG00000118655 0.53983052909861    -0.203467630204   3.47258609246081
#> ENSMUSG00000118659                0                 NA                NA
#>                                stat              pvalue
#>                           <numeric>           <numeric>
#> 
#> 
```

Accessing results

We can see here that this is a dataframe where the rows are genes and the columns are interesting data. The columns we are most interested in are log2FoldChange and padj. log2FoldChange is self-explanatory. padj is the pvalue for a test asking if the expression of this gene is different between the two conditions. This pvalue has been corrected for multiple hypothesis testing using the Benjamini-Hochberg method.

Let's do a little work on this dataframe to make it slightly cleaner and more informative.

```
differentiation.results <- results(dds, contr
  #Change this into a dataframe
  as.data.frame(.) %>%
  #Move ensembl gene IDs into their own colum
  rownames_to_column(., var = 'ensembl_gene_i
  #Get rid of columns that are so useful to u
  dplyr::select(., -c(baseMean, lfcSE, stat,
  #Merge this with a table relating ensembl_g
  inner_join(unique(dplyr::select(t2g, -ensem
  #Rename external_gene_name column
  dplyr::rename(., Gene = external_gene_name)

kable(differentiation.results[1:20,])
```

ensembl_gene_id	Gene	log2FoldChange	p
ENSMUSG00000064372	mt-Tp	2.9854645	0.0000
ENSMUSG00000064371	mt-Tt	-2.9647630	0.0056
ENSMUSG00000064370	mt-Cytb	0.2521125	0.0025
ENSMUSG00000064369	mt-Te	-0.0730461	0.9203
ENSMUSG00000064368	mt-	0.2969130	0.0011

Identifying differentially expressed genes

OK now we have a table of gene expression results. How many genes are significantly up/down regulated between these two timepoints? We will use 0.01 as an FDR (p.adj) cutoff.

```
#number of upregulated genes
nrow(filter(differentiation.results, padj < 0.01 & log2FoldChange > 0))
#number of downregulated genes
nrow(filter(differentiation.results, padj < 0.01 & log2FoldChange < 0))
```

```
#> [1] 6671
```

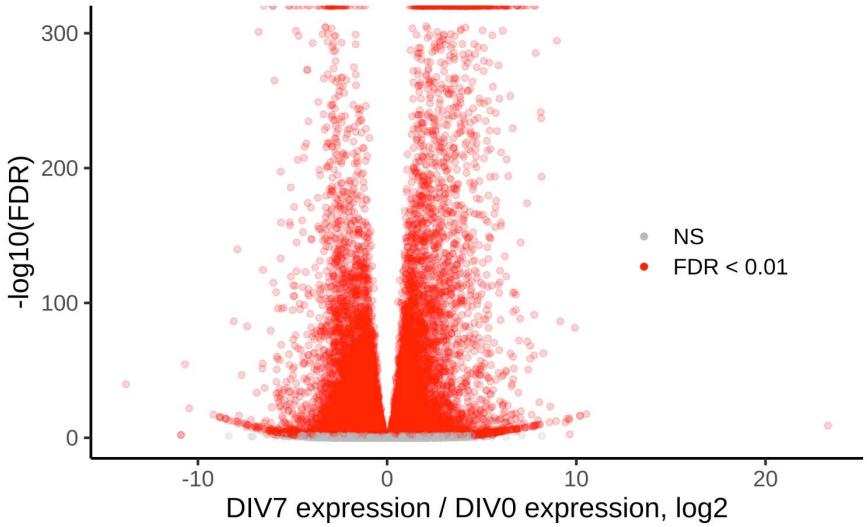
```
#> [1] 6813
```

Identifying differentially expressed genes

Let's make a volcano plot of these results.

```
#We are going to add an extra column to diffe
#hat tells us whether or not a gene
#meets the FDR cutoff
differentiation.results.sig <- mutate(differe
                                      sig = i
  #if a gene did not meet expression cutoffs
  #automatically does, it gets a pvalue of NA
  na.omit(.)

ggplot(differentiation.results.sig,
       aes(x = log2FoldChange, y = -log10(pad
  geom_point(alpha = 0.2) +
  xlab('DIV7 expression / DIV0 expression, lo
  ylab('-log10(FDR)') + theme_classic(16) +
  scale_color_manual(values = c('gray', 'red'
                     labels = c('NS', 'FDR <
  theme(legend.position = c(0.8, 0.5)) +
  guides(color = guide_legend(override.aes =
```



Adding a log2FC threshold

OK that's a lot of significant genes. What if, in addition to an FDR cutoff, we applied a log2FoldChange cutoff? We can do this by asking for p values that incorporate the probability that the log2FoldChange was greater than a threshold. This will of course be more conservative, but will probably give you a more confident set of genes.

```
#Is the expression of the gene at least 3-fold different?
differentiation.results.lfc <- results(dds, contrast = c('timepoint', 'DIV7', 'DIV0'),
                                       lfcThreshold = log(3, 2)) %>%
  #Change this into a dataframe
  as.data.frame(.) %>%
  #Move ensembl gene IDs into their own column
  rownames_to_column(., var = 'ensembl_gene_id') %>%
  #Get rid of columns that are so useful to us right now
  dplyr::select(., -c(baseMean, lfcSE, stat, pvalue)) %>%
  #Merge this with a table relating ensembl_gene_id with gene short names
  inner_join(unique(dplyr::select(t2g, -ensembl_transcript_id)), ., by = 'ensembl_gene_id') %>%
  #Rename external_gene_name column
  dplyr::rename(., Gene = external_gene_name) %>%
  mutate(., sig = ifelse(padj < 0.01, 'yes', 'no')) %>%
  na.omit(.)
```

Adding a log2FC threshold

As before, we can see the number of genes that are signficiantly up/down regulated between these two timepoints. This time, p values represent genes that we are confident are at least 3-fold up or down regulated.

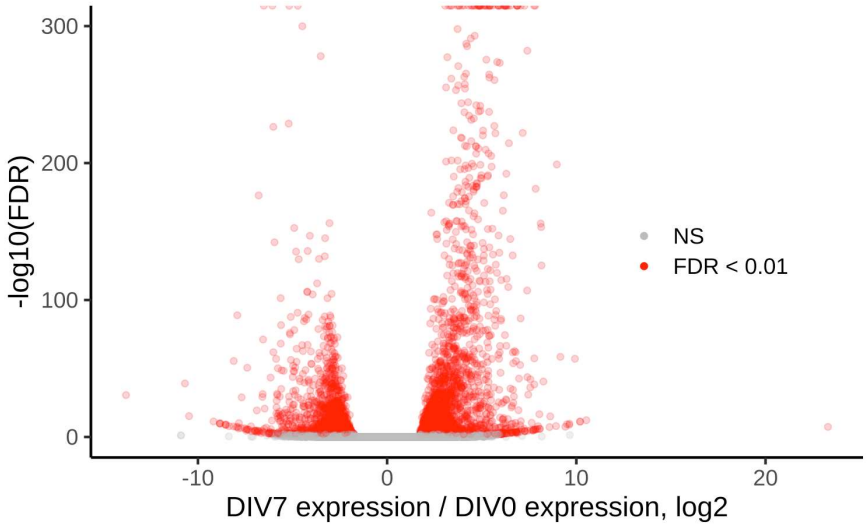
```
#number of upregulated genes
nrow(filter(differentiation.results.lfc, padj < 0.01 & log2FoldChange > 0))
#number of downregulated genes
nrow(filter(differentiation.results.lfc, padj < 0.01 & log2FoldChange < 0))
```

```
#> [1] 1834
```

```
#> [1] 1295
```

Adding a log2FC threshold

```
ggplot(differentiation.results.lfc, aes(x = l
  geom_point(alpha = 0.2) +
  xlab('DIV7 expression / DIV0 expression, lo
  ylab('-log10(FDR)') + theme_classic(16) +
  scale_color_manual(values = c('gray', 'red'
                     labels = c('NS', 'FDR <
  theme(legend.position = c(0.8, 0.5)) +
  guides(color = guide_legend(override.aes =
```



Again, fewer genes, but we can probably be more confident in them.

Plotting the expression of single genes

Sometimes we will have particular marker genes that we might want to highlight to give confidence that the experiment worked as expected. We can plot the expression of these genes in each replicate. For this case, let's plot the expression of two pluripotency genes (which we expect to decrease) and two strongly neuronal genes (which we expect to increase).

So what is the value that we would plot? Probably the most correct value is the 'normalized counts' value provided by `DESeq2`. This value is the safest to compare across samples within a gene. These counts are raw counts that have been normalized for sequencing depth and sample composition. However, I find that it is difficult to quickly get a sense of the expression level of a gene from it's number of normalized counts.

Let's say a gene had 500 normalized counts. Is that a highly expressed gene? A lowly expressed gene? Well, I won't really know unless I knew the length of the gene, and I don't have the length of every gene memorized.

A more interpretable value to plot might be TPM, since TPM is length-normalized. Let's say a gene was expressed at 500 TPM. Right off the bat, I know generally what kind of expression that reflects (pretty high).

```
#If you are interested in getting normalized counts, here's how you do it
normcounts <- counts(dds, normalized = TRUE)
```

Plotting the expression of single genes

Let's plot the expression of Klf4, Sox2, Bdnf, and Dlg4 in our samples.

```
#Get a table of tpms...remember txi is our tximport object
tpms <- txi$abundance %>%
  as.data.frame(.) %>%
  rownames_to_column(., var = 'ensembl_gene_id') %>%
  inner_join(unique(dplyr::select(t2g, -ensembl_transcript_id)), ., by = 'ensembl_gene_id') %>%
  dplyr::rename(., Gene = external_gene_name) %>%
  #Filter for genes we are interested in
  filter(., Gene %in% c('Klf4', 'Sox2', 'Bdnf', 'Dlg4'))

kable(tpms[1:4,])
```

ensembl_gene_id	Gene	DIV0.Rep1	DIV0.Rep2	DIV0.Rep3	DIV7.Rep1	DIV7.Rep2	DIV7.Rep3	DIV7.Rep4
ENSMUSG00000003032	Klf4	40.022629	46.241797	57.374566	9.063035	9.208364	9.715119	8.817037
ENSMUSG00000048482	Bdnf	2.456452	2.401157	2.351342	7.799981	7.762255	7.638515	7.433529
ENSMUSG00000020886	Dlg4	18.752921	15.749749	12.212636	151.558403	156.043098	153.269475	153.538436
ENSMUSG00000074637	Sox2	131.032132	120.433905	110.664468	24.484942	26.080394	27.106054	26.995461

Plotting the expression of single genes

OK, this is close to what we want, but in order to plot it we need to turn it from a wide table into a long table.

```
#Key is the new 'naming' variable
#Value is the new 'value' variable
#At the end we give it the columns that contain the values
#we want to reshape
tpms <- gather(tpms, key = sample, value = tpm, DIV0.Rep1:DIV7.Rep4)
kable(tpms)
```

ensembl_gene_id	Gene	sample	tpm
ENSMUSG00000003032	Klf4	DIV0.Rep1	40.022629
ENSMUSG00000048482	Bdnf	DIV0.Rep1	2.456452
ENSMUSG00000020886	Dlg4	DIV0.Rep1	18.752921
ENSMUSG00000074637	Sox2	DIV0.Rep1	131.032132
ENSMUSG00000003032	Klf4	DIV0.Rep2	46.241797
ENSMUSG00000048482	Bdnf	DIV0.Rep2	2.401157
ENSMUSG00000020886	Dlg4	DIV0.Rep2	15.749749

Plotting the expression of single genes

Now add one more column that tells which condition each replicate belongs to.

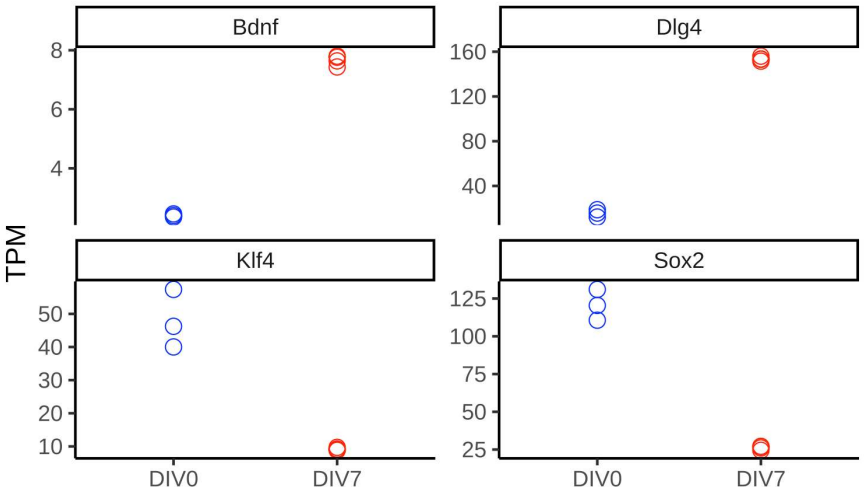
```
tpms <- mutate(tpms, condition = ifelse(grepl('DIV0', sample), 'DIV0', 'DIV7'))
kable(tpms)
```

ensembl_gene_id	Gene	sample	tpm	condition
ENSMUSG00000003032	Klf4	DIV0.Rep1	40.022629	DIV0
ENSMUSG00000048482	Bdnf	DIV0.Rep1	2.456452	DIV0
ENSMUSG00000020886	Dlg4	DIV0.Rep1	18.752921	DIV0
ENSMUSG00000074637	Sox2	DIV0.Rep1	131.032132	DIV0
ENSMUSG00000003032	Klf4	DIV0.Rep2	46.241797	DIV0
ENSMUSG00000048482	Bdnf	DIV0.Rep2	2.401157	DIV0
ENSMUSG00000020886	Dlg4	DIV0.Rep2	15.749749	DIV0

Plotting the expression of single genes

Now let's plot. Remember, we are expecting our pluripotency genes (Klf4 and Sox2) to go down while our neuron-specific genes (Bdnf and Dlg4) should go up.

```
ggplot(tpms, aes(x = condition, y = tpm, color
  geom_point(pch = 21, size = 4) +
  ylab('TPM') + xlab('') + theme_classic(16) +
  scale_color_manual(values = c('blue', 'red')
  facet_wrap(~Gene, scales = 'free_y')
```



Plotting the expression of groups of genes

OK, that's cool and all, but let's do something a little bigger. Say that instead of plotting individual genes we wanted to ask whether a whole class of genes are going up or down. We can do that by retrieving all genes that belong to a particular gene ontology term.

There are three classes of genes we will look at here:

- Maintenance of pluripotency (GO:0019827)
- Positive regulation of the cell cycle (GO:0045787)
- Neuronal differentitaion (GO:0030182)

So the idea here is to ask if the genes that belong to these categories behave differently than genes not in the category.

Plotting the expression of groups of genes

We can use `biomaRt` to get all genes that belong to each of these categories. Think of it like doing a gene ontology enrichment analysis in reverse.

```
pluripotencygenes <- getBM(attributes = c('er
                           filters = c('go_pa
                           values = c('GO:001
                           mart = mart)

cellcyclegenes <- getBM(attributes = c('ensem
                        filters = c('go_parer
                        values = c('GO:00457
                        mart = mart)

neurongenes <- getBM(attributes = c('ensembl_
                     filters = c('go_parent_t
                     values = c('GO:0030182')
                     mart = mart)
```

```
#>       ensembl_gene_id
#> 1 ENSMUSG00000028640
#> 2 ENSMUSG00000027612
#> 3 ENSMUSG00000063382
#> 4 ENSMUSG00000008999
#> 5 ENSMUSG00000042275
#> 6 ENSMUSG00000050966
```

You can see that these items are one-column dataframes that have the column name 'ensembl_gene_id'. We can now go through our results dataframe and add an annotation column that marks whether the gene is in any of these categories.

Plotting the expression of groups of genes

```
differentiation.results.annot <- differentiation.results %>%
  mutate(., annot = case_when(ensembl_gene_id %in% pluripotencygenes$ensembl_gene_id ~ 'pluripoten
                      ensembl_gene_id %in% cellcyclegenes$ensembl_gene_id ~ 'cellcycle',
                      ensembl_gene_id %in% neurongenes$ensembl_gene_id ~ 'neurondiff',
                      TRUE ~ 'none'))

#Reorder these for plotting purposes
differentiation.results.annot$annot <- factor(differentiation.results.annot$annot,
                                              levels = c('none', 'cellcycle', 'pluripotency', 'neu
```

ensembl_gene_id	Gene	log2FoldChange	padj	annot
ENSMUSG00000064372	mt-Tp	2.9854645	0.0000000	none
ENSMUSG00000064371	mt-Tt	-2.9647630	0.0056641	none
ENSMUSG00000064370	mt-Cytb	0.2521125	0.0025608	none
ENSMUSG00000064369	mt-Te	-0.0730461	0.9203204	none
ENSMUSG00000064368	mt-Nd6	0.2969130	0.0011097	none

Plotting the expression of groups of genes

OK we've got our table, now we are going to ask if the log2FoldChange values for the genes in each of these classes are different that what we would expect. So what is the expected value? Well, we have a distribution of log2 fold changes for all the genes that are *not* in any of these categories. So we will ask if the distribution of log2 fold changes for each gene category is different than that null distribution.

You can tie yourself in knots worrying about what is the right test to use for a lot of these things. What are the assumptions of the test? Does my data fit these assumptions? Is it normally distributed? My advice is to pretty much always just use rank-based nonparametric tests. Yes, you will sacrifice some power, meaning that your *p* values will generally be higher than those produced by parametric tests that make assumptions. However, in genomics, we often have many observations of something or many members in our groups. So if your *p* value is borderline where the choice of test makes a big difference, I'm already a little skeptical.

Wilcoxon rank sum test

For each GO group, use a Wilcoxon rank sum test to ask if the log2FoldChange values for genes within the group are different from the log2FoldChange values for genes outside of the group.

```
#Compare log2FC values for pluripotency genes and other genes
p.pluripotency <- wilcox.test(filter(differentiation.results.annot, annot == 'pluripotency')$log2Fo
                              filter(differentiation.results.annot, annot == 'none')$log2FoldChange

#Only 2 significant digits for the p value, s'il vous plait
p.pluripotency <- signif(p.pluripotency, 2)

p.cellcycle <- wilcox.test(filter(differentiation.results.annot, annot == 'cellcycle')$log2FoldChan
                              filter(differentiation.results.annot, annot == 'none')$log2FoldChange
p.cellcycle <- signif(p.cellcycle, 2)

p.neurondiff <- wilcox.test(filter(differentiation.results.annot, annot == 'neurondiff')$log2FoldCh
                              filter(differentiation.results.annot, annot == 'none')$log2FoldChange
p.neurondiff <- signif(p.neurondiff, 2)
```

> Note: Astute viewers may notice that we are not comparing every gene within a category to all genes outside of it. We are comparing all genes within a category to genes that are not in *any* category. This is a reasonable approximation for all genes since most genes aren't in any of these three categories.

Plot results

```
ggplot(differentiation.results.annot, aes(x =
  xlab('Gene class') + ylab('DIV7 expression
  #Make gray dotted line at y=0
  geom_hline(yintercept = 0, color = 'gray',
  #Boxplot to compare distributions
  geom_boxplot(notch = TRUE, outlier.shape =
  #Set colors manually
  theme_classic(14) + scale_color_manual(valu
  #Label x values
  scale_x_discrete(labels = c('none', 'Cell c
  #Set plot yvalue limits
  coord_cartesian(ylim = c(-5, 8)) +
  #Draw lines over samples
  annotate('segment', x = 1, xend = 2, y = 4,
  annotate('segment', x = 1, xend = 3, y = 5,
  annotate('segment', x = 1, xend = 4, y = 6,
  #Add pvalue results
  annotate('text', x = 1.5, y = 4.4, label =
```