Neuro-genomics - class 3
Modeling variations in gene expression & Identifying differentially expressed genes

We previously introduced the concept of turning the RNA content of a tissue sample into a vector of length ~20,000. This vector represents the expression of each one of the genes in the tissue. Last week we outlined how this vector is constructed in general. Today we will give a concrete example of how to create such a vector.
Then, we will discuss how to compare this vector between different kinds of tissues, or between several conditions of the same tissue type -- for example, disease and healthy conditions. This is called differential expression analysis.

Concrete example - creating a vector of expression from RNAseq of hippocampal neurons

Download the dataset SRX9109810 from Short Reads Archive (SRA):

1. Paste the name of the dataset in SRA
   You will see details about the experiment, and about the reads generated (see picture below)
2. Click on the run name (in this case it will be SRR12627533)
3. Then click on 'Reads' tab
4. Then click 'Filtered Download'
5. Check the box 'FASTQ' (instead of 'FASTA')
6. Click 'Download'
   The data will be in a text file, termed 'sra_data.fastq'. In this case it will be 2.1GB.

SRA

SRA SRX9109810

Create alert Advanced

COVID-19 Information

Public health information (CDC) | Research information (NIH) | SARS-CoV-2 data (NCBI) | Prevention and treatment information (HHS) |

Full Send to:

SRX9109810: Two matching plates of hippocampal neurons culture were sequenced - this one with with RNAseq
1 ILLUMINA (Illumina MiSeq) run: 4.6M spots, 676.2M bases, 318.6Mb downloads

Design: See details in https://doi.org/10.1101/2020.05.13.094268. To perform a comparison of untargeted ExSeq to bulk RNAseq of a matching neuronal culture sample, a culture plate was fixed with 4% PFA in exactly the same way as the original sample, washed with 1X PBS and was kept in 70% ethanol at 4 until RNA extraction; throughout the experimental procedure of sequencing the RNA from a matching sample, we followed the steps of the ExSeq sequenced sample when possible. Accordingly, the DNA was removed using DNase I (Roche). RNA was extracted from the sample using an RNeasy FFPE Kit (Qiagen) following the Deparaffinization using Melting protocol indicated in the kit. The extracted RNA was then reverse transcribed using SSIV reverse transcriptase with random primers, following the manufacturer's protocol. The RNA was removed using RNase DNase-free (Roche) and RNase H. The single stranded cDNA was transformed into ds-cDNA with NEBNext second strand synthesis (New England Biolabs). After purification with the Genomic DNA Clean & Concentrator Kit (Zymo), the Nextera XT DNA Library Preparation Kit (Illumina) was used to simultaneously fragment and tag the dsDNA with the adapter sequences required for the Illumina in vitro sequencing. The Nextera XT Index Kit (Illumina) was used to add Illumina sequencing barcodes during the PCR amplification of the extracted DNA. The library was sequenced using a MiSeq instrument to generate paired-end sequencing reads with the MiSeq Reagent Kit v3.

Submitted by: MIT

Study: Validations of expansion sequencing
PRJNA663046 • SRP282177 • All experiments • All runs
show Abstract

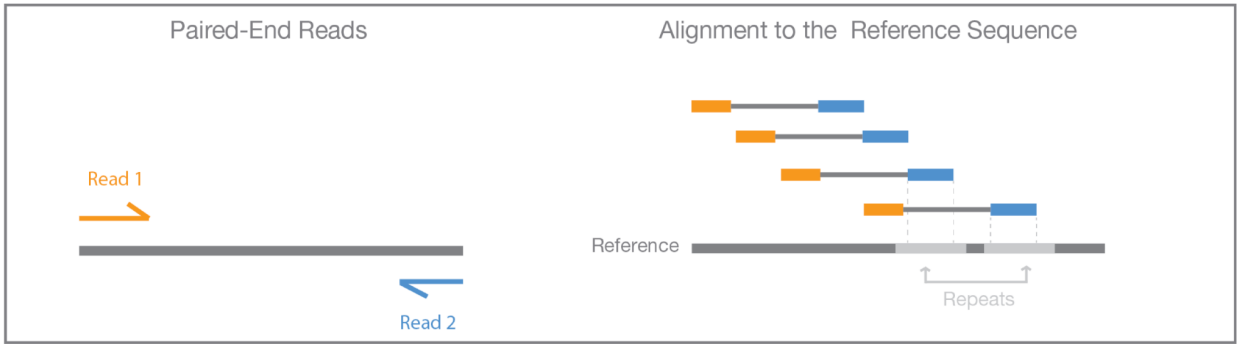
Sample: Two matching plates of hippocampal neurons culture were sequenced - this one with with RNAseq
SAMN16115965 • SRS7353179 • All experiments • All runs
Organism: Mus musculus

Library:
Name: RNAseq_culture_matching
Instrument: Illumina MiSeq
Strategy: RNA-Seq
Source: TRANSCRIPTOMIC
Selection: other
Layout: PAIRED

Runs: 1 run, 4.6M spots, 676.2M bases, 318.6Mb

Run	# of Spots	# of Bases	Size	Published
SRR12627533	4,554,532	676.2M	318.6Mb	2020-09-13

As you can see, in this dataset 4,554,532 reads were generated, and they were paired-end.
Reminder - Paired-end sequencing (picture from here):
The idea is to sequence from both sides of the RNA fragment. This helps to accurately align the reads to the genome or transcriptome (i.e. the mRNA sequence, a.k.a the genes).



Last week we outlined two approaches to generate expression vectors: the first is to properly align the reads against the genome or the transcriptome and then to count the number of times each read is aligned to each gene. This approach will be better for quantifying modifications in RNA/DNA.
The second approach was to create pseudo-alignments and immediately count the number of copies each gene has. The second approach is faster, and will work great if we just need to quantify the expression of the genes. This is what we will practice now.

Run Kallisto to quantify the gene expression:

1. Download Kallisto
2. Download Kallisto indices for the transcriptome of the mouse. Several files will be downloaded, including the actual indices 'transcriptome.idx', and the file 'transcripts_to_genes.txt' which we will discuss later.
3. Run Kallisto:
   kallisto quant -i mus_musculus/transcriptome.idx -o kallisto_results ../sra_data.fastq

An error will appear…
Error: paired-end mode requires an even number of input files

From the manual of kallisto:
"kallisto can process either single-end or paired-end reads. The default running mode is paired-end and requires an even number of FASTQ files represented as pairs"

To understand what is going on, let's take a look at the fastq file. To look at the fastq text files, and to modify it, we will use python, and specifically Jupyter Notebook.

Note: make sure that python version 3 is installed on your computer. Then install Jupyter Notebook.
The jupyter notebook below can be found here.

In [3]:
```
file1 = open("../sra_data.fastq","r")
for x in range(16):
    print(file1.readline())
file1.close()
```

```
@SRR12627533.1.1 1 length=74

CTATACGCGGGAGCTATTTCCTGGAACCGCTTCGAAGCCCTTCCAATCCAATAAGGAAGAACAACATACGCGAT

+SRR12627533.1.1 1 length=74

-8,B<E<C+:7++8E<FGGFGG9,,8FF7FGGEC8<CFFFGGGGFGGGGF?EC<<@@E<8<F@8FFGCF@E=:F

@SRR12627533.1.2 1 length=75

CGCTAAAGCCATACACCGAAGCTGTGGATTTGCTTAGCAATAGGCAAGTGGTAGCGGAGCGTTCTGTAAGCCGTT

+SRR12627533.1.2 1 length=75

-8ACBCEFFGGGEFGGGGGGGGGFGFGFGGGCFGGGGGGGFFGGGGGFEC@FGFG@FGGGGGGGGGGFGDDGGGG

@SRR12627533.2.1 2 length=75

CACATACACCAAATGTCTGAACCTGCGGTTCCTCTCGTACTGAGCAGGATTACCATGGCAACAACAACATCATCA

+SRR12627533.2.1 2 length=75

--8ACAF9CF<,8E,FFGFCAFFG@CFEGEFGGGGGFGEGGFC@FFEGDGGGGGGGGGGFGGGGGGGGGGGDGGC

@SRR12627533.2.2 2 length=73

CTCTGTATTCGTACTGAAAATCAAGATCAAGCGAGCTTTTGCCCTTCTGCTCCACGGGAGGTTTCTGTCCTCC

+SRR12627533.2.2 2 length=73

-8CCCGGGGGGGGGGGFGGGGGGFGGGGGGGGDGCFGGGGGGGGGGGGGGGGGGGGGGGGGGGGFGGGGGG
```

The open command opens a file, in this case for reading only (“r”), and creates a handle for the file - ‘file1’ in this case.
Then, file1.realine() reads one line from this file. In this case we are printing the first 16 lines from the file.
Finally, file1.close() closes files according to the file handle.
How to understand the fastq file?
Reminder - each read has 4 lines: (1) @ and then the read name; (2) the actual read sequence; (3) + and then the read name again; (4) the read quality score.
The quality scores are:

```
LLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLL..................................................
PPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPP
!"#$%&'()*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmnopqrstuvwxyz{|}~
```

```
0.2.....................26...31........41
0..................20........30........40........50.....................................93
```

As you can see, the paired-end reads appear in the same file (for example, read 1.1 is the pair of read 1.2), whereas two separate files are required by Kallisto. Therefore we need to divide the fastq file into two files, according to the .1 and .2 character.
Let's use regular expressions to create these two fastq files. The jupyter notebook below can be found here.

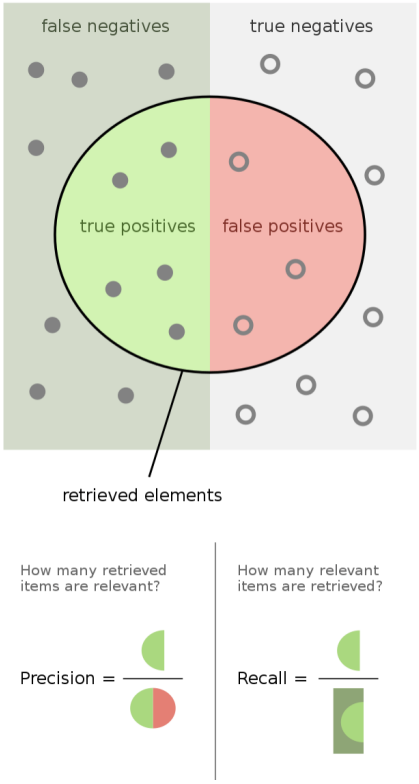
```
In [33]: import re
file0 = open("../sra_data.fastq","r")
file1 = open("Pair1.fastq","w")
file2 = open("Pair2.fastq","w")
count = 0
x=file0.readline()
while True:
    match1 = re.search('.+\.1\s',x)
    match2 = re.search('.+\.2\s',x)
    if match1:
        file1.writelines(x)
        for x in range(3):
            x=file0.readline()
            file1.writelines(x)
    elif match2:
        file2.writelines(x)
        for x in range(3):
            x=file0.readline()
            file2.writelines(x)
    else:
        print(x)
    x=file0.readline()
    if not x:
        break
file0.close()
file1.close()
file2.close()
```

We created three files: one that we are reading ("r") which will get the handle file0, and two files that we are writing ("w") which will get handles file1 and file2.
The regex pattern is '.+\.1\s'
which means any symbol '.' more than one time '+'. Then we escape the character '.' by using backslash '\'. This means that we are looking for the actual character '.' (and not any symbol). In this case, we are looking for the specific pattern '.1'. However, this pattern alone can appear in any read that ends with the number 1 (for example SRR12627533.1.1). Therefore we add the special character \s at the end, which means that a space must be present after the '.1' pattern. match1 and match2 will be true if the pattern is found in the string.
In the command x=file0.readline(), x will be true for every line, except when we reach the end of the file.

And now we can properly run Kallisto:
kallisto quant -i mus_musculus/transcriptome.idx -o kallisto_results Pair1.fastq Pair2.fastq

Let's take a look at two lines from the displayed output:
[quant] processed 4,554,532 reads, 1,255,634 reads pseudoaligned

[quant] estimated average fragment length: 265.046
The first line tells us that out of 4,554,532 reads, only 1,255,634 reads were pseudo-aligned. This means that most of the reads will not be used at all! We will quantify only 1,255,634 reads. This is an approach which gives high Precision and low Recall - meaning that we discard many reads that might be useful, but the reads that we do align are probably correct (image from wikipedia).



The second line tells us the RNA fragments were ~265 bases on average. From that fragment of ~265 bases, the sequencing was ~75 bases from each side.

The important output file is abundance.tsv (.tsv means tab-separated values). If you open it, you will see four columns:
target_id length eff_length est_counts tpm
target_id is the name of the transcript. Note that each gene can have several transcripts, due to alternative splicing. Therefore, different transcripts of the same gene are also called isoforms. Also note that not all transcripts belong to genes that code for proteins -- there are many non-coding genes. The link between transcripts and genes is given in the file 'transcripts_to_genes.txt' which was downloaded along with the indices. Using the UCSC genome browser we can view all the transcripts of a given gene.

transcripts_to_genes.txt

ENSMUST00000027035.9	ENSMUSG00000025902.13	Sox17
ENSMUST00000195555.1	ENSMUSG00000025902.13	Sox17
ENSMUST00000192650.5	ENSMUSG00000025902.13	Sox17
ENSMUST00000116652.7	ENSMUSG00000025902.13	Sox17
ENSMUST00000192505.1	ENSMUSG00000025902.13	Sox17
ENSMUST00000191647.1	ENSMUSG00000025902.13	Sox17
ENSMUST00000191939.1	ENSMUSG00000025902.13	Sox17
ENSMUST00000192913.1	ENSMUSG00000025902.13	Sox17
ENSMUST00000193450.1	ENSMUSG00000104238.1	Gm37587
ENSMUST00000194935.1	ENSMUSG00000104238.1	Gm37587
ENSMUST00000195361.1	ENSMUSG00000102269.1	Gm7357
ENSMUST00000180019.1	ENSMUSG00000096126.1	Gm22307
ENSMUST00000193889.1	ENSMUSG00000103003.1	Gm38076
ENSMUST00000195771.1	ENSMUSG00000104328.1	Gm37323
ENSMUST00000192738.1	ENSMUSG00000102735.1	Gm7369
ENSMUST00000182774.1	ENSMUSG00000098104.1	Gm6085
ENSMUST00000193443.1	ENSMUSG00000102175.1	Gm6119
ENSMUST00000157375.1	ENSMUSG00000088000.1	Gm25493
ENSMUST00000195671.1	ENSMUSG00000103265.1	Gm2053
ENSMUST00000193658.1	ENSMUSG00000103922.1	Gm6123
ENSMUST00000130201.7	ENSMUSG00000033845.13	Mrpl15
ENSMUST00000156816.6	ENSMUSG00000033845.13	Mrpl15
ENSMUST00000045689.13	ENSMUSG00000033845.13	Mrpl15
ENSMUST00000115538.4	ENSMUSG00000033845.13	Mrpl15
ENSMUST00000192286.1	ENSMUSG00000033845.13	Mrpl15
ENSMUST00000146665.2	ENSMUSG00000033845.13	Mrpl15
ENSMUST00000132625.1	ENSMUSG00000033845.13	Mrpl15

length is the length of the transcript, in bases
eff_length is the effective length of the transcript. The meaning of effective length of a transcript is related to the concept of 'counting' to quantify gene expression (explanation adapted from here):
"Counts" usually refers to the number of reads that align to a particular transcript. These numbers are heavily dependent on two things: (1) the amount of fragments you sequenced and (2) the length of the transcript, or more appropriately, the effective length. Effective length refers to the number of possible start sites a feature could have generated a fragment of that particular length.
Imagine, for simplicity, that the length of a gene is 10 bases and the length of the RNA fragments is 7 bases. Then the effective length is 4, because there are 4 possible ways to align 7 bases long RNA in a 10 bases long transcript.
In general, the effective length is:

$$\widetilde{l}_i = l_i - \mu_{FLD} + 1$$

Where l is the length of the transcript, and μ is the length of the fragment.
Indeed, in our example, for most transcripts the effective length is just l-264 (for transcripts shorter or close in length to F the calculation is a bit different).
est_counts means estimated counts, marked by Xi.This is the number of reads aligned to each transcript. **In most cases, counts are used by differential expression methods since they can be modeled (we will discuss this shortly).**
Why is it called estimated counts, and not just counts? The problem is that we are aligning against transcripts, and in most cases different transcripts share the same exon(s). As a result, we will have cases in which the read actually aligns to more than one transcript. Therefore, the

number of counts per transcripts is usually estimated using an expectation-maximization procedure (figure from this paper):

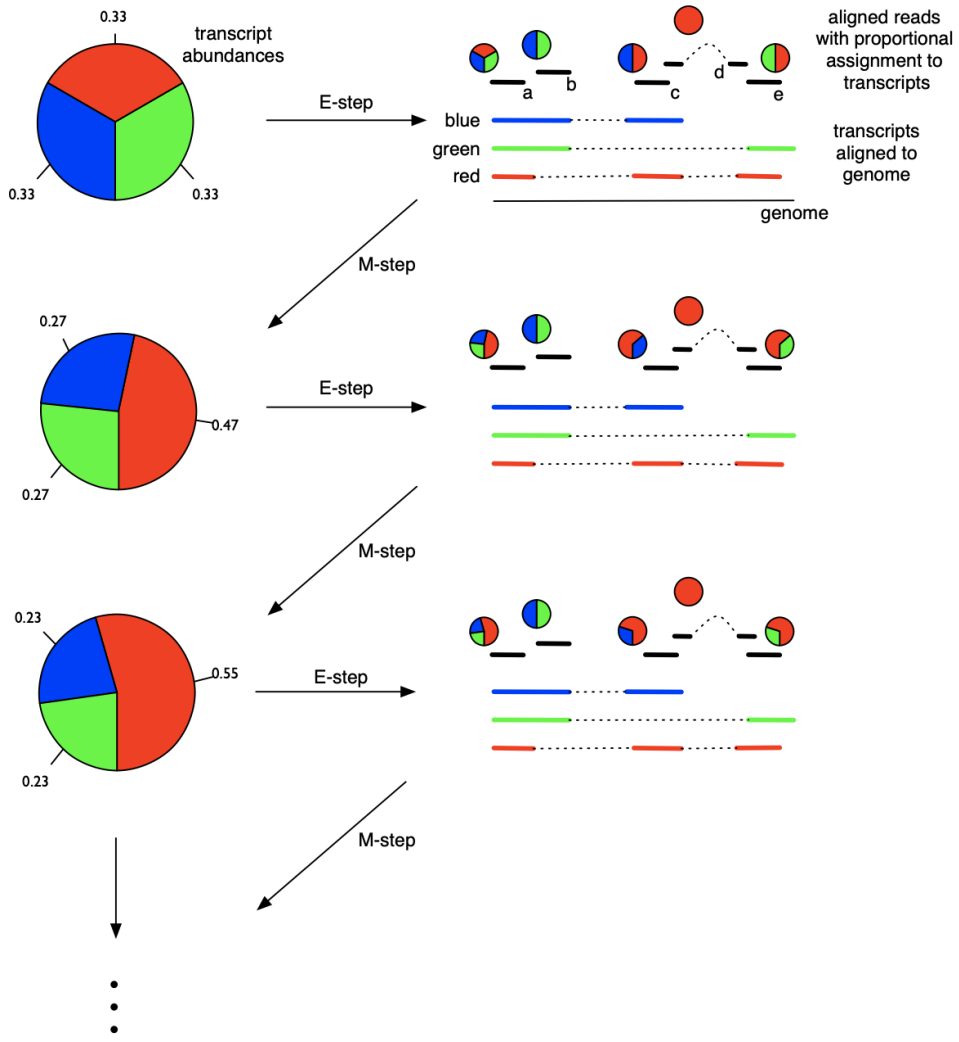


FIGURE 4. Illustration of the EM algorithm. The gene has three isoforms (`red, green, blue`) of the same length. There are five reads (a,b,c,d,e) mapping to the gene. One maps to all three isoforms, one only to red, and the other three to each of the three pairs of isoforms. Initially every isoform is assigned the same abundance $(\frac{1}{3}, \frac{1}{3}, \frac{1}{3})$. During the *expectation* (E) step reads are proportionately assigned to transcripts according to the isoform abundances. Next, during the *maximization* (M) step isoform abundances are recalculated from the proportionately assigned read counts. Thus, for example, the abundance of `red` after the first M step is estimated by $0.47 = (0.33 + 0.5 + 1 + 0.5)/(2.33 + 1.33 + 1.33)$.

The last parameter in the output file of Kallisto is:
tpm which means Transcripts Per Million. Why do we need any other quantification beside the estimated counts? In other words, what is the limitation of the estimated counts?

There are two problems with the estimated counts: 1) it doesn't account for the effective length of the transcript. For example, if a transcript X has 1000 counts and a transcript Y has 500 counts, it can be just a result of the fact that transcript X is twice as long. This means that we can't compare counts between different transcripts within a sample! Another thing that we can't do with counts is to sum them together to measure the total counts of a gene. Again, this is because we need to account for the fact that each transcript can have a different length. 2) it doesn't account for the total number of reads sequenced. It is clear that if we generate many sequenced reads, then the number of counts will be higher for most transcripts.
A way to account for the effective length is to divide the estimated counts by the effective length for each transcript. A way to account for the total number of reads sequenced is to normalize the number of reads to 1 million. This is exactly the case for TPM (transcripts per million):

$$\mathrm{TPM}_i = \frac{X_i}{\widetilde{l}_i} \cdot \left(\frac{1}{\sum_j \frac{X_j}{\widetilde{l}_j}} \right) \cdot 10^6$$

We can compare TPM within the sample to learn which transcript was more expressed in the tissue. We can also sum all the TPM from different transcripts of the same gene, to get an estimate of the expression level of a gene.
Another common within-sample normalization, which accounts for the effective length of the transcripts, and normalizes for 1 million reads, is the FPKM. FPKM means Fragments Per Kilobase of exon per Million reads mapped:

$$\mathrm{FPKM}_i = \frac{X_i}{\left(\frac{\widetilde{l}_i}{10^3}\right)\left(\frac{N}{10^6}\right)} = \frac{X_i}{\widetilde{l}_i N} \cdot 10^9$$

Where N is the total number of reads sequenced.

Note that we only discussed within-sample normalization until now. We haven't discussed between-sample normalization. It is important to note that without between-sample normalization it is almost impossible to compare between the same gene in different samples, even if the values are TPM or FPKM. Normalization using sum of total counts is discussed in part2 of the exercise. More advanced normalizations exist, including for example TMM normalization. A review of normalization methods and their underlying assumptions can be found here.

Differential expression analysis

A typical sequencing experiment is designed to address the following question - which genes have different expression levels when comparing two (or more) experimental conditions. However, detecting differentially expressed genes is not trivial. Consider the following example: say that we have two normalized expression vectors, one of condition A and one of condition B, and gene X has expression level (i.e. counts) 150 in condition A and 200 in condition B.
How can we know if gene X is differentially expressed between the two conditions?

One can conclude from the increase in counts that gene X is influenced by the conditions tested. However, another likely explanation is that the increase in counts is due to either technical variations (noise introduced by the sequencing procedure) or biological variations (for example, difference in expression levels due to the inherent stochasticity of gene expression).

Therefore, technical repeats (samples generated from the same biologically material) and/or biological repeats (samples generated from biologically distinct material) are needed in each sequencing experiment. Repeats are essential, but unfortunately the number of repeats that can be performed is usually limited by experimental and financial considerations. Given that usually only a handful of repeats are performed for each condition, it's practically impossible to deduce the technical and biological variations in the expression level of each gene from the data alone.

Therefore we need to model the data (here 'data' is defined as the expression levels of each gene in one experimental condition but with several repeats) using well-known distributions. Knowing the distribution will provide the expected variability in the expression level of each gene, and allow comparison of different experimental conditions. However, if we use a well-known distribution that doesn't really model the data, then our estimation about the expected variability will be wrong.

In parts 3&4 of the exercise you will learn about two distributions, Poisson and Negative Binomial, and will learn how to decide if the expression vector actually fits these models.