RNA-seq data analysis practical - Agricultural Omics, EBI

Mar Gonzàlez-Porta

2014/02/19

Contents

Introduction

This tutorial will illustrate how to use standalone tools, together with R and Bioconductor for the analysis of RNA-seq data. Keep in mind that this is a rapidly evolving field and that this document is not intended as a review of the many tools available to perform each step; instead, we will cover one of the many existing workflows to analyse this type of data.

We will be working with a subset of a publicly available dataset from *Drosophila melanogaster*, which is available both in the Short Read archive (SRP001537[1] - raw data) and in Bioconductor (pasilla package[2] - processed data). For more information about this dataset please refer to the original publication (Brooks et al. 2010[3]).

The tools and R packages that we will be using during the practical are listed below (see Software requirements[4]) and the necessary data files can be found here[5]. After dowloading and uncompressing the `tar.gz` file, you should have the following directory structure in yuor computer:

```
RNAseq
|-- reference              # reference info (e.g. genome sequence and annotation)
`-- data
    |-- raw                # raw data: fastq files
    |-- mapped             # mapped data: BAM files
    `-- demultiplexing     # extra fastq files for the demultiplexing section
```

[1] http://www.ebi.ac.uk/ena/data/view/SRP001537
[2] http://www.bioconductor.org/packages/release/data/experiment/html/pasilla.html
[3] http://genome.cshlp.org/content/early/2010/10/04/gr.108662.110
[4] https://github.com/mgonzalezporta/TeachingMaterial#software-requirements
[5] http://www.ebi.ac.uk/~mar/courses/RNAseq.tar.gz

Dealing with raw data

The FASTQ format

The nucleotide sequences and qualities of the short reads produced in a sequencing experiment are commonly stored in a plain text file using the FASTQ format. In the `data/raw` directory, you will find two fastq files, which contain information about the short reads obtained from one of the samples in the *Drosophila melanogaster* experiment.

Exercise: Why do we have two fastq files for this given sample? Solution[6]

To confirm that we are working with a fastq file and to get an idea of how this format looks like we can print the first lines of our files by typing this into the terminal:

```
zcat SRR031714_1.fastq.gz | head
zcat SRR031714_2.fastq.gz | head
```

Exercise: How many lines are used to represent a read in the fastq file? Which information do they contain? Solution[7]

Exercise: How many reads are there in each file? Do both files contain the same number of reads? Is that what we would expect? Solution[8]

Quality assessment (QA)

We will be using FastQC[9] to generate our first QA report. This software can be executed in two different modes: either using the graphical user interface (if we just type `fastqc` on the terminal) or as a command itself (if we add extra parameters). For example, we can print the help documentation by typing the following:

```
fastqc -h
```

To generate a report for our files we only have to provide the file names as an argument:

```
# might take a while
fastqc SRR031714_1.fastq.gz SRR031714_2.fastq.gz
```

As a result we will obtain the file `filename_fastqc.zip`, which will be automatically unzipped in the `filename_fastqc` directory. There we will find the QA report (`fastqc_report.html`), which provides summary statistics about the numbers of reads, base calls and qualities, as well as other information (you will find a detailed explanation of all the plots in the report in the project website[10]).

Exercise: The information provided by the QA report will be very useful when deciding on the options we want to use in the filtering step. After checking it, can you come up with some criteria for the filtering of our file (i.e. keeping/discarding reads based on a specific quality threshold)? Solution[11]

Exercise: As we have seen in the previous section, fastq files contain information on the quality of the read sequence. The reliability of each nucleotide in the read is measured using the Phred quality score, which represents the probability of an incorrect base call:

[6] https://github.com/mgonzalezporta/TeachingMaterial/blob/master/solutions/_fastq_ex1.md

[7] https://github.com/mgonzalezporta/TeachingMaterial/blob/master/solutions/_fastq_ex2.md

[8] https://github.com/mgonzalezporta/TeachingMaterial/blob/master/solutions/_fastq_ex3.md

[9] http://www.bioinformatics.babraham.ac.uk/projects/fastqc/

[10] http://www.bioinformatics.babraham.ac.uk/projects/fastqc/Help/3%20Analysis%20Modules/

[11] ../solutions/_qa_ex1.md

$$Q = -10 \cdot log_{10}P$$

Figure 1: Phred quality score formula

where `Q` is the Phred quality value and `P` the probability of error. For example, a Phred quality score of 20 would indicate a probability of error in the base call of 1 in 100 (i.e. 99% accuracy). If you inspect the fastq file again though, you will see that this information is not displayed in number format, but is encoded in a set of characters. During the filtering step, we will be using tools that read these characters and transform them into quality values, so we need to be sure first about the encoding format used in our data (either phred 33 or phred 64). Using the information provided in the QA report (under the *per base sequence quality* section) and in the Wikipedia entry for the FASTQ format[12], can you guess which encoding format was used? Solution[13]

Exercise: As we have seen, a visual interpretation of the QA report is a very useful practice when dealing with HTS data. However, it becomes a very tedious task if we are working with huge volumes of data (imagine we have 1000 fastq files to inspect!). Thankfully, the developers of FastQC have thought of that. Can you spot any alternative output of this software that we could use in this situation? Solution[14]

Filtering FASTQ files

After analysing the QA report, one might want to discard some of the reads based on several criteria, such as quality and nucleotide composition. We will use two different tools to perform these filtering steps: PRINSEQ[15] and fastq-mcf[16].

PRINSEQ offers a wide range of options for filtering and we can learn about them in the manual:

```
prinseq-lite -h
```

Based on what we have learned from the QA report, we could decide to apply the following filters:

```
zcat SRR031714_1.fastq.gz | prinseq-lite \
    -fastq stdin \
    -out_good SRR031714_1_filt1 \
    -out_bad null \
    -trim_qual_right 30 -min_len 32 \
    -ns_max_p 5 \
    -lc_method dust -lc_threshold 10

zcat SRR031714_2.fastq.gz | prinseq-lite \
    -fastq stdin \
    -out_good SRR031714_2_filt1 \
    -out_bad null \
    -trim_qual_right 30 -min_len 32 \
    -ns_max_p 5 \
    -lc_method dust -lc_threshold 10
```

[12] http://en.wikipedia.org/wiki/FASTQ_format
[13] ../solutions/_qa_ex2.md
[14] ../solutions/_qa_ex3.md
[15] http://prinseq.sourceforge.net/
[16] https://code.google.com/p/ea-utils/

Exercise: Which are the criteria that we are using to discard reads? Solution[17]

Exercise: Which extra option should we have had to use if our files had been in phred 64 format? Solution[18]

Exercise: After filtering the fastq file it is not a bad idea to obtain a QA report again to decide whether we are happy with the results. Do you think we got rid of the main issues spotted initially? Solution[19]

Exercise: Usually it is also useful to keep track of the number of reads available in the fastq file both before and after the filtering step. Can you gather this information from the FastQC reports? Given that we are working with paired-end data, do you see any limitation? Now imagine we were working with a bigger number of files; can you come up with a more automated way to check that? Solution[20]

The filtering step might become complicated if you are working with paired-end data, since you have to be sure that both fastq files (one for each read pair) contain the same number of reads in the same order. There are some tools available that simplify this task, for example fastq-mcf. We can print a list of the avaiable options just by typing in the name of the tool:

```
fastq-mcf
```

We observe that a main functionality of fastq-mcf is to remove adapters from the fastq file. For this reason we need to provide a fasta file with the adapter sequences. In our case, we have decided to check only for the standard Illumina paired-end adapter:

```
cat adapters.fa
```

Now that we have a good understanding of the required input we can proceed to execute the tool, trying to match the options that we used previously with PRINSEQ:

```
fastq-mcf adapters.fa SRR031714_1.fastq.gz SRR031714_2.fastq.gz \
    -o SRR031714_1_filt2.fastq -o SRR031714_2_filt2.fastq \
    -q 30 -P 33 -l 32 --max-ns 1
```

Exercise: How does the QA report look like this time? Do we have the same number of reads in each file? Solution[21]

Exercise: Something else that one might want to check is the read length. How long were our reads before we started with the filtering step? How long are they now? Solution[22]

Depending on the filtering options used, we might end up with a set of reads with different lengths. *A priori*, this should not be a limitation, but we might encounter some difficulties in the downstream analyses, depending on the tools we want to use. For this reason, if we have a clear idea of what we want to do with the data, it is always a good practice to check the requirements for the tools that we are planning to use before taking any steps. If that is not the case, in order to be on the safe side, we can use filtering options that affect all reads equally, eg:

[17] ../solutions/_filtering_fastq_ex1.md
[18] ../solutions/_filtering_fastq_ex2.md
[19] ../solutions/_filtering_fastq_ex3.md
[20] ../solutions/_filtering_fastq_ex4.md
[21] ../solutions/_filtering_fastq_ex5.md
[22] ../solutions/_filtering_fastq_ex6.md

```
zcat SRR031714_1.fastq.gz | prinseq-lite \
    -fastq stdin \
    -out_good SRR031714_1_filt3 \
    -out_bad null \
    -trim_right 5

zcat SRR031714_1.fastq.gz | prinseq-lite \
    -fastq stdin \
    -out_good SRR031714_1_filt3 \
    -out_bad null \
    -trim_right 5
```

Exercise: Let us generate the QA reports one last time. How do they compare to the previous ones? Solution[23]

In conclusion, there are many filtering combinations that you can apply, and the specific options will largely depend on the type of data and the posterior analyses. We recommend to check the PRINSEQ manual[24] for a nice overview on the topic.

De-multiplexing samples

Nowadays, NGS machines produce so many reads that the coverage obtained per lane for the transcriptome of organisms with small genomes is very high. Sometimes it is more valuable to sequence more samples with lower coverage than sequencing only one with very high coverage. With this end, multiplexing techniques have been optimised to sequence several samples in a single lane using 4-6 bp barcodes to uniquely identify the sample within the library (e.g. Lefrançois et al. 2009[25]). This approach is very advantageous for researchers, especially in terms of cost, but it adds an additional layer of pre-processing that is not as trivial as one would think, given the average 0.1-1% sequencing error rate that introduces a lot of multiplicity in the actual barcodes. Most commonly the data is de-multiplexed immediately after sequencing, and only FASTQ files that are ready for analyses are distributed. However, you might encounter the necessity to perform the de-multiplexing step yourself, or, given de-multiplexed FASTQ files, to remove the adaptors manually; thus, it is important to learn how to deal with such data.

The data which we were working on in the previous section was not mutiplexed, so we will now work with a different fastq file that can be found under the `data/demultiplexing` directory. In this directory you will also find information on the barcodes used:

```
cat barcodes.txt
```

Exercise: Imagine we do not know whether the barcode was introduced in the 5' or the 3' end of the reads. How can we figure that out? Solution[26]

In order to separate the reads in 4 different fastq files (one for each barcode/sample) we will use fastq-multx[27]. We can learn more about this tool by typing its name in the terminal:

```
fastq-multx
```

[23] ../solutions/__filtering__fastq__ex7.md

[24] http://prinseq.sourceforge.net/manual.html

[25] http://www.biomedcentral.com/1471-2164/10/37

[26] ../solutions/__demultiplexing__ex1.md

[27] https://code.google.com/p/ea-utils/

Although we already know where our barcodes are located within the read, from the documentation we observe that fastq-multx will attempt to guess the position for us. Let us try the automatic guessing with the following command:

```
fastq-multx barcodes.txt barcoded.fastq -o %.barcoded.fastq
```

After executing the command above you should have five new fastq files: one corresponding to each sample and one for the reads that did not match any of the barcodes.

Exercise: Try generating a QA report for one of the samples. How does it compare to the report for the initial multiplexed fastq file? What happened to the read length? Solution[28]

Aligning reads to the genome

So far we have been working with fastq files, which contain the reads that were generated during the sequencing experiment. *A priori* we do not know from which transcripts those reads were originated, and that is precisely what will be addressed in following steps, starting with the mapping. There are essentially two ways of approaching this: one is to align the reads to a known transcriptome or genome, and the other is to assemble these reads *de novo* into a transcriptome without the need for any reference.

Exercise: Can you think of any advantages/disadvantages for the above mentioned approaches? Solution[29]

Here we decided to align our reads to a known reference genome. To achieve this task, you could use any aligner capable of exporting its results in the SAM/BAM format or in a format that can be easily converted to this one. In the case of RNA-seq data, we also want to be able to retain information about split reads (i.e. reads with a gap) and spliced reads (i.e. those that span multiple exons). There are many aligners available, some of them optimised for working with RNA-seq data (e.g. TopHat, GSNAP… - see Fonseca et al. 2012[30] for a review). In this practical we decided to use TopHat[31], and these are the commands we would use to map the fastq files that we have generated during the filtering step:

```
# do not run - very time consuming
# output already provided in the data/mapped directory

cd reference

# obtain the reference genome from Ensembl
wget ftp://ftp.ensembl.org/pub/release-62/fasta/drosophila_melanogaster/dna/\
    Drosophila_melanogaster.BDGP5.25.62.dna_rm.toplevel.fa.gz
gunzip Drosophila_melanogaster.BDGP5.25.62.dna_rm.toplevel.fa.gz

# index the reference genome
bowtie-build Drosophila_melanogaster.BDGP5.25.62.dna_rm.toplevel.fa \
    Drosophila_melanogaster.BDGP5.25.62.dna_rm.toplevel

# align the reads
```

[28] ../solutions/_demultiplexing_ex2.md

[29] https://github.com/mgonzalezporta/TeachingMaterial/blob/master/solutions/_aligning_ex1.md

[30] http://bioinformatics.oxfordjournals.org/content/28/24/3169

[31] http://tophat.cbcb.umd.edu/

```
tophat Drosophila_melanogaster.BDGP5.25.62.dna_rm.toplevel \
    ../data/raw/SRR031714_1_filt3.fastq \
    ../data/raw/SRR031714_2_filt3.fastq
```

The last command runs TopHat with the default options. A detailed description of those, as well as information on other commands (e.g. for single-end reads), can be found in the manual[32] or just by typing the name of the tool in the console (i.e. type `tophat` on its own).

Notice the `reference` directory, which contains, amongst other files, the genomic sequence for *Drosophila melanogaster* (`Drosophila_melanogaster.BDGP5.25.62.dna_rm.toplevel.fa`). We can inspect which chromosomes are present in this fasta file with the following command:

```
grep '^>' Drosophila_melanogaster.BDGP5.25.62.dna_rm.toplevel.fa | head
```

Exercise: Suppose we had decided to align to the transcriptome instead. Similarly to what we did with the genome sequence, the transcriptome sequence for *Drosophila melanogaster* can be obtained from Ensembl with the following command:

```
# do not run
wget ftp://ftp.ensembl.org/pub/release-62/fasta/drosophila_melanogaster/cdna/\
    Drosophila_melanogaster.BDGP5.25.62.cdna.all.fa.gz
gunzip Drosophila_melanogaster.BDGP5.25.62.cdna.all.fa.gz
```

This file has been provided on the `reference` directory. What do the "chromosome" names correspond to in this case? Solution[33]

We are now familiarised with the input required to align reads to a reference genome or transcriptome. In both cases, the output produced by the mapping tool is going to be stored in SAM/BAM format, which we will inspect in the next section.

Dealing with aligned data

The SAM/BAM format

The SAM/BAM format is the standard way of representing the results from the alignment step. It contains the same information as in the fastq file, plus some extra fields providing mapping information, for example, the coordinates where each of the reads was aligned. A SAM file is a plain text file with the information spread across different columns, and a BAM file is just its compressed version in binary format. In order to save disk space, we will typically work with BAM files; however, we can easily transform a BAM file into SAM format using samtools[34]:

```
# do not run
# output already provided in data/mapped
samtools view -h -o untreated3.sam untreated3.bam
```

We can now inspect the first lines of the file with standard Unix commands:

[32] http://tophat.cbcb.umd.edu/manual.html
[33] https://github.com/mgonzalezporta/TeachingMaterial/blob/master/solutions/_aligning_ex2.md
[34] http://samtools.sourceforge.net/samtools.shtml

```
head -n20 untreated3.sam
```

Alternatively, we can directly inspect the contents of a BAM file with the following samtools command:

```
samtools view untreated3.bam | head
```

Exercise: Why do we get a different output from the two previous commands? How can we obtain information about the header from the BAM file? Hint: try typing `samtools view` into the terminal. Solution[35]

Exercise: The first column in the BAM file contains the read name. Take a closer look at the first alignments: why do you think some of the names appear twice, while others seem to be present only once? Solution[36]

Exercise: A description of the SAM format can be found in the samtools website[37], under the section *SAM format*. With the combination of samtools and Unix commands, try to answer the following questions:

- How many reads are mapped in total?
- How many reads map to each chromosome?
- How many different mapping qualities are represented in the BAM file, and how many reads have each of them assigned?
- How many different alignment flags can you find in the BAM file? What do they represent? *Hint:* http://picard.sourceforge.net/explain-flags.html
- Try to print the unique CIGAR strings for the first 300 reads. What is their meaning? *Hint:* http://genome.sph.umich.edu/wiki/SAM

Solution[38]

Visualising aligned reads

Several genome browsers exist to visualise the files generated during the analysis of high-troughput sequencing data, including BAM files. Two of the most popular tools are IGV[39] and Tablet[40]. In this practical we will be using IGV to visualise our BAM file. We can launch this tool from the *Download* section in the project website[41].

Once the interface is loaded, we can proceed to load the necessary files. In our case, we will load the following information:

- the reference genome:

  ```
  Genomes > Load genome from file >
      reference/Drosophila_melanogaster.BDGP5.25.62.dna_rm.toplevel.fa
  ```

- the BAM file:

[35] https://github.com/mgonzalezporta/TeachingMaterial/blob/master/solutions/_bam_ex1.md
[36] https://github.com/mgonzalezporta/TeachingMaterial/blob/master/solutions/_bam_ex2.md
[37] http://samtools.sourceforge.net/samtools.shtml
[38] https://github.com/mgonzalezporta/TeachingMaterial/blob/master/solutions/_bam_ex3.md
[39] http://www.broadinstitute.org/igv/
[40] http://bioinf.scri.ac.uk/tablet/
[41] http://www.broadinstitute.org/igv/download

```
File > Load from file > data/mapped/untreated3.bam

# IGV requires the BAM file to be indexed, which can be achieved with samtools
#   (i.e. `samtools index bam_file)
# For this practical the index is already provided, so there is no need to run this command
```

- the annotation:

```
File > Load from file > reference/Drosophila_melanogaster.BDGP5.25.62.gtf
```

Exercise: Spend some time exploring the loaded BAM file and how the reads overlap with the known annotation. Can you find examples of split and spliced reads? For a subset of the reads, some nucleotides are highlighted in a different color. What do you think the explanation for this is? *Hint:* http://www.broadinstitute.org/igv/AlignmentData

Solution[42]

Filtering BAM files

Samtools can also be used to further modify and/or subset BAM files. For example, some tools will require that the reads are sorted by coordinate or by name. In addition, and similarly to what we did with the fastq files, one might consider to discard reads with a low alignment quality (including reads that align to several locations in the genome), or in the case of paired-end data, discard reads that are not properly paired.

Exercise: Try to answer to the following questions using samtools and the information provided in the documentation[43]:

- How many reads are properly paired?
- By default, TopHat creates BAM files where the reads are sorted by coordinate. How would you sort the properly paired reads by name instead? Save the output in a new BAM file called `untreated3_paired.bam`, which we will use later on during the practical.
- Which percentage of those properly paired reads map uniquely? *Hint:* Have a look at the options for `samtools sort` and `samtools view`.

Solution[44]

Counting reads overlapping annotated genes

Following the read mapping step, we can proceed working with BAM files with standalone tools or load them directly in R. These two worfkflows are not exclusive and we will cover both of them for illustrative purposes.

[42] https://github.com/mgonzalezporta/TeachingMaterial/blob/master/solutions/_visualising_ex1.md
[43] http://samtools.sourceforge.net/samtools.shtml
[44] https://github.com/mgonzalezporta/TeachingMaterial/blob/master/solutions/_filtering_bam.md

With htseq-count

htseq-count[45] is a simple but yet powerful tool to overlap a BAM file with the genome annotation and thus obtain the number of reads that overlap with our features of interest. As usual, we can obtain information on the tool by typing `htseq-count -h` and by referring to its website.

Exercise: 0ne of the input files required by htseq-count is a GTF file. For this practical, you will find this file under the directory `reference`. Which information does it contain? Solution[46]

Exercise: As we have already mentioned, the other required input file is a BAM file. Can you spot any specific requirement regarding this file? *Hint*[47] - Solution[48]

In addition to the input file requirements, special care must be taken in dealing with reads that overlap more than one feature (e.g. overlapping genes), and thus might be counted several times in different features. To deal with this, htseq-count offers three different counting modes: union, intersection-strict and intersection-nonempty.

Exercise: What are the differences between these three counting modes? *Hint*[49] - Solution[50]

Now that we have a good understanding of the input files and options, we can proceed to execute htseq-count:

```
# the following command takes a while to execute
# the output file is already provided in the data/mapped directory
samtools view untreated3_paired.bam | htseq-count \
    --mode=intersection-nonempty \
    --stranded=no \
    --type=exon \
    --idattr=gene_id - \
    ../../reference/Drosophila_melanogaster.BDGP5.25.62.chr.gtf > htseq_count.out
```

Exercise: In addition to the counts that overlap known genes, the output file also contains some extra information on reads that could not be assigned to any of those; can you find it? Solution[51]

With R

Computing gene counts in R is very similar to what we have done so far with htseq-count. However, it requires some extra steps, since we first need to load the necessary files (i.e. BAM files and annotation).

Note: All the commands provided in this section have to be executed in R. Make sure to specify the working directory properly before starting (e.g. `setwd("./data")`*).*

Importing BAM files There are three main functions to load BAM files into R:

- *scanBam*: this function is part of the *Rsamtools* package and is the low level function used by the other two. It potentially reads *all* fields (including CIGAR strings and user defined tags) of a BAM file into a list structure, but allows you to select specific fields and records to import.

[45] http://www-huber.embl.de/users/anders/HTSeq/doc/count.html
[46] https://github.com/mgonzalezporta/TeachingMaterial/blob/master/solutions/_counting_ex1.md
[47] http://www-huber.embl.de/users/anders/HTSeq/doc/count.html
[48] https://github.com/mgonzalezporta/TeachingMaterial/blob/master/solutions/_counting_ex2.md
[49] http://www-huber.embl.de/users/anders/HTSeq/doc/count.html
[50] https://github.com/mgonzalezporta/TeachingMaterial/blob/master/solutions/_counting_ex3.md
[51] https://github.com/mgonzalezporta/TeachingMaterial/blob/master/solutions/_counting_ex4.md

- *readAligned*: a higher-level function defined in the *ShortRead* package which imports some of the data (query names, sequences, quality, strand, reference name, position, mapping quality and flag) into an *AlignedRead* object. *ShortRead* was the first package developed to read in NGS data and is able to read almost sequencer every manufacturer proprietary formats, so you could for example also use it to read an Illumina export file produced by a GenomeAnalyzer GAIIx.
- *readGappedAlignments* and *readGappedAlignmentPairs*: two functions from the *GenomicRanges* package that create an object intended for operations such as searching for overlaps or coverage. Each alignment is described by its position and strand on the reference and read ids, sequences and base qualities are discarded for the sake of memory usage and speed.

In this section we will import the data using the *readGappedAlignmentPairs* function, intended for paired-end data. In order to speed up the process of importing the data, we will use the function *ScanBamParam* to load only the reads that map to chromosome 4:

```
library(GenomicRanges)
library(Rsamtools)

# define a filter
which=RangesList(IRanges(1, 1351857))
names(which)="chr4"
which
param=ScanBamParam(which=which)

# import the data
aln_chr4=readGappedAlignmentPairs("untreated3.bam", use.names=T, param=param)
aln_chr4
```

We have now stored our data in an object of the class *GappedAlignmentPairs*, which has been defined in the *GenomicRanges* package and does not correspond to the standard R classes. For this reason, it is useful to check the documentation for this package[52] in order to learn how to access our data.

Exercise: After having a look at the *GenomicRanges* documentation, try to answer the following questions:

- How many reads have been loaded?
- How can we access the read names? What about the strand information?
- How can we access the information for the first reads in the pair? Try to print a vector with their start coordinates.
- How many reads are properly paired?
- What is the percentage of reads that map to multiple locations?
- What information does the command `seqlevels(aln_chr4)` provide? *Hint:* look for the *GappedAlignmentPairs* class

Solution[53]

Since we have loaded only the reads that map to chromosome 4, we can proceed to modify the `aln_chr4` object accordingly:

```
seqlevels(aln_chr4)="chr4"
aln_chr4
```

[52] http://www.bioconductor.org/packages/release/bioc/manuals/GenomicRanges/man/GenomicRanges.pdf

[53] https://github.com/mgonzalezporta/TeachingMaterial/blob/master/solutions/_counting_ex5.md

Importing the annotation To link the alignments to their respective features, we need access to the genome annotation for the studied organism, in our case *Drosophila melanogaster*, which contains information on the coordinates of known exons, genes and transcripts. Similarly to what we encountered when loading BAM files, there is more than one way to load the annotation in R (see the Bioconductor resources[54] for further details). It is extremely important to pay attention to overlapping features (e.g. exons shared by multiple transcripts within the same gene), since they might end up complicating the downstream analysis (e.g. we need to make sure not to count the same read multiple times). In order to circumvent this limitation, in this practical we will use the *biomaRt* package to query Ensembl directly from R and retrieve only the necessary information:

```
library(biomaRt)

ensembl62=useMart(host="apr2011.archive.ensembl.org",
    biomart="ENSEMBL_MART_ENSEMBL",
    dataset="dmelanogaster_gene_ensembl")

fields=c("chromosome_name",
    "strand",
    "ensembl_gene_id",
    "ensembl_exon_id",
    "start_position",
    "end_position",
    "exon_chrom_start",
    "exon_chrom_end")
annot=getBM( fields, mart=ensembl62)
```

Exercise: Have a look at the newly created `annot` object. What type of object is it? *Hint:* use the function *class*. Solution[55]

Exercise: In the next subsection we will calculate the overlap between the loaded BAM file and the annotation with the function *summarizeOverlaps* from the *GenomicRanges* package. What is the input required? Do we have all the necessary objects ready? *Hint:* type `?summarizeOverlaps`. Solution[56]

Before we proceed to calculate the counts, we need to store the annotation information in an object of the proper class:

```
annot=GRanges(
    seqnames = Rle(paste("chr", annot$chromosome_name, sep="")),
    ranges = IRanges(start=annot$exon_chrom_start,
                  end=annot$exon_chrom_end),
    strand = Rle(annot$strand),
    exon=annot$ensembl_exon_id,
    gene=annot$ensembl_gene_id
  )
annot
class(annot)
```

Counting reads over known genes in R Now that we have the alignment locations (`aln_unique` object) and the genome annotation (`annot` object), we can quantify gene expression by counting reads

[54] http://www.bioconductor.org/help/course-materials/

[55] https://github.com/mgonzalezporta/TeachingMaterial/blob/master/solutions/_counting_ex6.md

[56] https://github.com/mgonzalezporta/TeachingMaterial/blob/master/solutions/_counting_ex7.md

over all exons of a gene and summing them together. Similarly to what we encountered with htseq-count, we need to pay attention to those reads that overlap with several features.

```
counts=summarizeOverlaps(
    annot, aln_chr4, ignore.strand=T, mode="IntersectionNotEmpty")
exon_counts=assays(counts)$counts[,1]
names(exon_counts)=elementMetadata(annot)$gene

head(exon_counts, n=15)
```

Exercise How many elements does the vector `exon_counts` contain? Why is that? *Hint:* use the function *length*. Solution[57]

Let us subset the counts for chromosome 4:

```
genes_chr4=unique(elementMetadata(annot[seqnames(annot)=="chr4"])$gene)
exon_counts_chr4=exon_counts[names(exon_counts) %in% genes_chr4]
```

Exercise: So far we have obtained the number of reads overlapping each exon. How can we combine this information to obtain gene counts? *Hint:* use the functions *split* and *sapply*. Solution[58]

Alternative approaches

In this section of the practical we have seen how to calculate the number of reads that overlap known gene models. In the two approaches evaluated here, those reads that mapped to multiple features were not considered. This is a simplification we may not want to pursue, and alternatively, there are several methods to probabilistically estimate the expression of overlapping features (e.g. Trapnell et al. 2010[59], Turró et al. 2011[60], Li et al. 2010[61]…).

Normalising counts

With RPKMs

While in the previous sections the data was derived from a single sample, in this exercise we will work with the precomputed counts for all the samples in our experiment[62]:

```
library("pasilla")

data("pasillaGenes")
counts=counts(pasillaGenes)
head(counts)
```

A common way to normalise reads is to convert them to RPKMs (or FPKMs in the case of paired-end data). This implies normalising the read counts depending on the feature size (exon, transcript, gene model…) and on the total number of reads sequenced for that library:

Exercise: Let us obtain RPKMs for the table `counts` following these steps:

[57] https://github.com/mgonzalezporta/TeachingMaterial/blob/master/solutions/_counting_ex8.md

[58] https://github.com/mgonzalezporta/TeachingMaterial/blob/master/solutions/_counting_ex9.md

[59] http://www.nature.com/nbt/journal/v28/n5/abs/nbt.1621.html

[60] http://genomebiology.com/content/12/2/R13

[61] http://bioinformatics.oxfordjournals.org/content/26/4/493.long

[62] http://bioconductor.org/packages/2.11/data/experiment/html/pasilla.html

$$RPKMs = \frac{gene\ counts}{gene\ length \cdot library\ size} \cdot 10^9$$

Figure 2: RPKM formula

- Calculate the length of the exons in the `annot` object and store the result in a vector, with the name of each element set to the corresponding gene. *Hint:* check the *width* and *elementMetadata* accessors.
- Obtain gene lengths by adding up the lengths of all the exons in each gene. *Hint:* check the functions *split* and *sapply*.
- Normalise the counts by the library size. *Hint:* check the function *colSums*.
- Continue normalising by gene length, but be aware that the object that contains the gene lengths and the one that contains the normalised counts might have a different number of genes.
- Finally, obtain RPKMs by multiplying by a factor of 10^9.

Solution[63]

Such a count normalisation is suited for visualisation, but sub-optimal for further analyses. A better way of normalising the data is to use either the *edgeR* or *DESeq2* packages.

With DESeq2

RPKM normalisation is not the most adequate for certain types of downstream analysis (e.g. differential gene expression), given that it is susceptible to library composition biases. There are many other normalisation methods that should be considered with that goal in mind (see Dillies et al. 2012[64] for a comparison). In this section we are going to explore the one offered within the *DESeq2* package:

```
library("DESeq2")
library("pasilla")

# load the count data
# you can skip this if you have already loaded the data in the previous section
data("pasillaGenes")
countData=counts(pasillaGenes)

# load the experimental design
colData=data.frame(condition=pData(pasillaGenes)[,c("condition")])

# create an object of class DESeqDataSet, which is the data container used by DESeq2
dds=DESeqDataSetFromMatrix(
        countData = countData,
        colData = colData,
        design = ~ condition)

colData(dds)$condition=factor(colData(dds)$condition,
```

[63] https://github.com/mgonzalezporta/TeachingMaterial/blob/master/solutions/_normalising_ex1.md
[64] http://bib.oxfordjournals.org/content/early/2012/09/15/bib.bbs046.long

```
        levels=c("untreated","treated"))
dds

# the DESeqDataSet class is a container for the information we just provided
head(counts(dds))
colData(dds)
design(dds)
```

In order to normalise the raw counts we will start by determining the relative library sizes, or *size factors* for each library. For example, if the counts of the expressed genes in one sample are, on average, twice as high as in another, the size factor for the first sample should be twice as large as the one for the other sample. These size factors can be obtained with the function *estimateSizeFactors*:

```
dds=estimateSizeFactors(dds)
sizeFactors(dds)
```

Once we have this information, the normalised data is obtained by dividing each column of the count table by the corresponding size factor. We can perform this calculation by calling the function counts with a specific argument as follows:

```
deseq_ncounts=counts(dds, normalized=TRUE)
```

Exercise: We have now accumulated three different versions of the same dataset: the raw counts (`counts`), the RPKM quantifications (`rpkm`) and the DESeq normalised counts (`deseq_ncounts`). How would you visualise the performance of each normalisation method in getting rid of the variation that does not associate to the experimental conditions that are being evaluated? Solution[65]

Differential gene expression

NOTE: This section is based on the code provided in the DESeq2[66] vignette, which can be checked for extra information.

A basic task in the analysis of expression data is the detection of differentially expressed genes. The *DESeq2* package provides a method to test for differential expression by using a generalised linear model in which counts are modeled with a negative binomial distribution. It expects a matrix of count values where each column corresponds to a sample and each line to a feature (e.g. a gene). Typically, a *DESeq2* analysis is performed in three steps: count normalisation, dispersion estimation and differential expression test, although the authors also provide a wrapper function for those steps.

Count normalisation

Since we have already generated a matrix with the normalised counts in the previous section (see Normalising counts with DESeq2[67]), we will use it directly as input for the next step.

[65] https://github.com/mgonzalezporta/TeachingMaterial/blob/master/solutions/_normalising_ex2.md

[66] http://www.bioconductor.org/packages/2.13/bioc/html/DESeq2.html

[67] 25.normalising.md#with-deseq2

Dispersion estimation

An important step in differential expression analysis is to figure out how much variability we can expect in the expression measurements within the same condition. Unless this is known, we cannot make inferences about whether the change in expression observed for a given gene is big enough to be considered significant, or whether it corresponds to the variability that we would expect by chance. This is why it is so important to have replicates: they show how much variation occurs without a difference in the condition.

In *DESeq2*, in order to estimate the dispersion for each gene, we can use the function *estimateDispersions*:

```
dds=estimateDispersions(dds)
```

The result of the estimation can be visualised with the *plotDispEsts* function:

```
pdf(file="./de_dispersion.pdf")
plotDispEsts(dds)
dev.off()
```

Differential expression test

Finally, we will use the function *nbinomWaldTest* to contrast the two studied conditions:

```
dds=nbinomWaldTest(dds)
results=results(dds)
```

The *padj* column in the table `dds` contains the p-values adjusted for multiple testing with the Benjamini-Hochberg procedure (i.e. FDR). This is the information that we will use to decide whether the expression of a given gene differs significantly across conditions (e.g. we can arbitrarily decide that genes with an FDR<0.10 are differentially expressed).

Exercise: How would you select those genes that pass a given FDR threshold (e.g. FDR<0.10)? Which are the most significant? Solution[68]

Let us generate an MA plot to evaluate the results of the differential expression analysis:

```
pdf(file="./de_ma.pdf")
plotMA(dds,ylim=c(-2,2))
dev.off()
```

The *DESeq* wrapper function

The three steps detailed above can be performed with just one single function, which takes as input a DESeqDataSet object like the one we have generated in the previous section (see Normalising counts with DESeq2[69]).

```
# first create dds object
dds=DESeq(dds)
results=results(dds)
```

[68] ../solutions/_de_ex1.md

[69] 25.normalising.md#with-deseq2

Differential exon usage

NOTE: This section is based on the code provided in the DEXSeq[70] vignette, which can be checked for extra information.

So far we have been focusing on analysing the transcriptome from a gene-centric perspective. However, one of the advantages of RNA sequencing is that it allows us to address questions about alternative splicing in a much easier way than it used to be possible with microarrays. There are a large number of methods to detect significant differences in splicing across conditions (e.g. cuffdiff[71], mmdiff[72]), many of which rely on the non-trivial task of estimating transcript expression levels. Here we will focus on *DEXSeq*[73], a Bioconductor package for the identification of differential exon usage events from exon counts. In other words, *DEXSeq* reports cases where the expression of a given exon, relative to the expression of the corresponding gene, changes across conditions.

The structure of a *DEXSeq* analysis is analogous to the one we have seen so far for differential expression with *DESeq*: count normalisation, dispersion estimation and differential exon usage test. However, there are a couple of things to take into account before getting started with that workflow, as we'll see next.

Preparing the annotation

Exons might be represented multiple times in a standard GTF file if they are shared between multiple transcripts or genes. This overlap can include the entire exon, or just part of it, as illustrated in Figure 1[74] from the *DEXSeq* paper. Thus, in order to ensure that each exon is tested only once, *DEXSeq* requires a slightly modified annotation file, in which exons are flattened into counting bins. We only need to prepare this flattened annotation file once, and this can be achieved by executing the command below:

```
# do not run - it takes a while
# you'll find the output here: RNAseq_all/reference

cd RNAseq_all/reference
python /path/to/library/DEXSeq/python_scripts/dexseq_prepare_annotation.py \
    --aggregate=no Drosophila_melanogaster.BDGP5.25.62.chr.gtf \
    Drosophila_melanogaster.BDGP5.25.62.chr_dexseq_noaggregate.gff
```

Exercise: How does the newly generated annotation file differ from the previous one? Try this:

```
cd RNAseq_all/reference

original=Drosophila_melanogaster.BDGP5.25.62.chr.gtf
grep FBgn0031208 $original | awk '$3=="exon"'

flattened=Drosophila_melanogaster.BDGP5.25.62.chr_dexseq_noaggregate.gff
grep FBgn0031208 $flattened | awk '$3=="exonic_part"'
```

[70] http://www.bioconductor.org/packages/2.13/bioc/html/DEXSeq.html
[71] http://cufflinks.cbcb.umd.edu/manual.html#cuffdiff
[72] https://github.com/eturro/mmseq#flexible-model-comparison-using-mmdiff
[73] http://www.bioconductor.org/packages/2.13/bioc/html/DEXSeq.html
[74] http://genome.cshlp.org/content/22/10/2008/F1.expansion.html

What is the number of lines obtained in each case? What is the number of counting bins for this gene? Hint[75] - Solution[76]

Exercise: One of the options used in this example to generate the flattened annotation file is `--aggregate=no`. What does this refer to? Hint:

```
python /path/to/library/DEXSeq/python_scripts/dexseq_prepare_annotation.py -h
```

Solution[77]

Each of the exonic parts in the flattened annotation file are the potentially testable bins. However, before we can perform the testing, we first need to know the number of reads that overlap with each of them.

Counting reads overlapping exon bins

Provided that we have aligned our reads to the genome with a splice-aware aligner (e.g. TopHat) we can now proceed to count the number of reads that fall into exon bins in our sample:

```
# do not run - it takes a while
# you'll find the output here: RNAseq_all/data/mapped
gff_file=Drosophila_melanogaster.BDGP5.25.62.chr_dexseq_noaggregate.gff
bam_file=../data/mapped/untreated3.nsorted.bam
out=$bam_file.dexseq_noaggregate.txt

samtools view $bam_file | python /path/to/library/DEXSeq/inst/python_scripts/dexseq_count.py \
    --paired=yes --stranded=no $gff_file - $out
```

Exercise: By default, `dexseq_count` will only consider reads that mapped with a mapping quality of 10 or more. Even though we didn't explicitly set this option in the command above, we can learn about this on the help text:

```
python /path/to/library/DEXSeq/inst/python_scripts/dexseq_count.py -h
```

What does this threshold refer to? Solution[78]

Exercise: Previously we have been calculating the number of reads that overlap each gene using `htseq-count`. What's the difference between these two tools? Hint: think of the previous steps in this section and the input required by this tool. Also, would the gene counts generated with these two tools be the same? Solution[79]

[75] http://apr2011.archive.ensembl.org/Drosophila_melanogaster/Gene/Summary?db=core;g=FBgn0031208;r=2L:6687-10326

[76] ../solutions/_deu_ex1.md

[77] ../solutions/_deu_ex2.md

[78] ../solutions/_deu_ex3.md

[79] ../solutions/_deu_ex4.md

Loading the counts into R

Finally we just need to load the counts into R. Here we'll be working with an example dataset, and thus we'll be loading the counts directly from the *pasilla* library:

```
library(DEXSeq)
library("pasilla")

data("pasillaExons")
ecs=pasillaExons

head(counts(pasillaExons))
```

Alternative for your own data Alternatively, to load the count files for your experiment into R, you should first generate a table summarising your experimental design:

```
cat sampleTable.txt

#               countFile           condition
# untreated1    untreated1.counts   control
# untreated2    untreated2.counts   control
# untreated3    untreated3.counts   control
# untreated4    untreated4.counts   control
# treated1      treated1.counts     knockdown
# treated2      treated2.counts     knockdown
# treated3      treated3.counts     knockdown
```

And then load it in R and generate an expression set object:

```
library(DEXSeq)

sampleTable=read.table("sampleTable.txt")
annot="Drosophila_melanogaster.BDGP5.25.62.chr_dexseq_noaggregate.gff"

ecs = read.HTSeqCounts(
        countfiles = sampleTable$countFile,
        design = sampleTable,
        flattenedfile = annot )
```

Count normalisation

Independently on whether you're working with the example counts (available through the *pasilla* library) or the ones for your own samples, the next essential step of the workflow consists on estimating the size factors for each library, used to take into account variable sequencing depths. This step is common to the one previously followed with *DESeq* (see the section on Normalising counts[80]):

```
ecs=estimateSizeFactors(ecs)
sizeFactors(ecs)
```

[80] ./25.normalising.md

Dispersion estimation

Also analogous to the workflow followed with *DESeq* is the step on estimating the dispersion on the observed counts for each of the exons (see the section on Differential gene expression[81]). This information is used to quantify the variability that we can expect between biological replicates, and will help us in addressing which of the observed differences are big enough to be attributed to a change in the condition.

```
ecs=estimateDispersions( ecs )
ecs=fitDispersionFunction( ecs )
```

We can next plot the calculated dispersion estimates as a function of the mean normalised counts, just as a sanity test:

```
out="dexseq_dispersion.pdf"
pdf(file=out)
plotDispEsts( ecs )
dev.off()
```

Exercise: Exons with a low number of counts tend to have very high variability and will not end up as a significant result. In order to reduce computation time, *DEXSeq* skips such exons, as specified in the help for the `estimateDispersions` function:

```
?estimateDispersions
```

What's the fraction of exons in our dataset that will be tested by *DEXSeq*? Hint:

```
counts_subset=head(counts(ecs))
testable_subset=head(fData(ecs)$testable)
cbind(counts_subset, testable_subset)
```

Solution[82]

Differential exon usage test

Finally, we can test for differential exon usage and calculate the fold-changes:

```
ecs=testForDEU(ecs)
ecs=estimatelog2FoldChanges( ecs )

result=DEUresultTable(ecs)
head( result )
```

Exercise: Why do we get `NA` values for `FBgn0000256:E006`? Solution[83]

Exercise: How many exons do we identify as differentially used (e.g. FDR < 0.1)? How many genes? Solution[84]

[81] ./26.de.md

[82] ../solutions/_deu_ex5.md

[83] ../solution/_deu_ex6.md

[84] ../solution/_deu_ex7.md

Visualisation

It is in general a good practise to visualise the results in the form of an MA-plot:

```
out="dexseq_ma.pdf"
pdf(file=out)
plotMA( ecs, FDR=0.1, ylim=c(-4,4), cex=0.8 )
dev.off()
```

In addition, *DEXSeq* offers a nice way to visualise differential exon usage events for a given gene:

```
out="dexseq_FBgn0085442.pdf"
pdf(file=out)

plotDEXSeq( ecs, "FBgn0085442", expression=FALSE,
    norCounts=TRUE, displayTranscripts=TRUE,
    legend=TRUE, cex.axis=1.2, cex=1.3, lwd=2 )

dev.off()
```

Identification, annotation and visualisation of splicing switch events

When working with RNA-seq data, several tools exist to quantify differences in splicing across conditions and to address the significance of those changes (e.g. DEXSeq). Quiet often though, these tools result in a long list of genes that is difficult to interpret. By relying on transcript level quantifications, LOREM provides a simple (yet powerful) approach to identify, annotate and visualise the most extreme changes in splicing across two different conditions, namely switch events. In brief, switch events are defined as those cases where, for a given gene, the identity of the most abundant transcript changes across conditions:

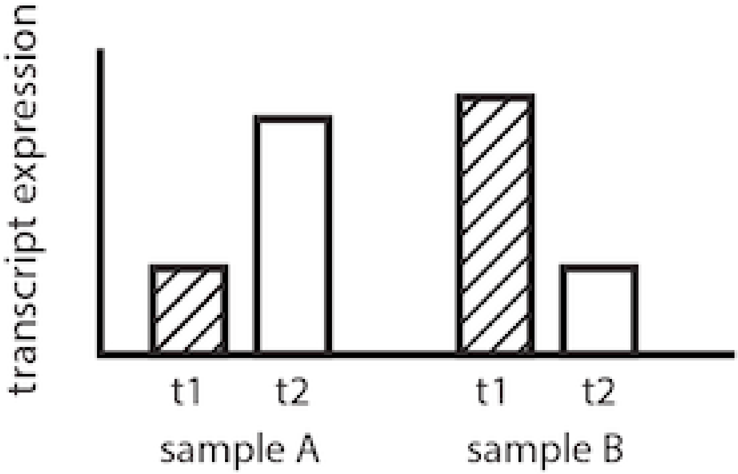

Figure 3: Switch event

Further information on LOREM can be found on the project wiki[85]. Here we'll briefly explore an example output report[86].

Exercise: How would you interpret the switch event identified for *SRSF6*? Hint[87]

Extra information

Software requirements

Note: depending on the topics covered in the course some of these tools might not be used.

- Standalone tools:
  - FastQC[88]
  - PRINSEQ[89]
  - eautils[90]
  - samtools[91]
  - IGV[92]
  - htseq-count[93]
- Bioconductor packages:
  - GenomicRanges[94]
  - Rsamtools[95]
  - biomaRt[96]
  - pasilla[97]
  - DESeq[98]

Other resources

Course data

- Complete course data, including command outputs and R sessions[99]

[85] https://github.com/mgonzalezporta/lorem/wiki
[86] http://www.ebi.ac.uk/~mar/tools/lorem/html_test1/
[87] https://github.com/mgonzalezporta/lorem/wiki/Tutorial#interpreting-lorem-output
[88] http://www.bioinformatics.babraham.ac.uk/projects/fastqc/
[89] http://prinseq.sourceforge.net/
[90] https://code.google.com/p/ea-utils/
[91] http://sourceforge.net/projects/samtools/
[92] http://www.broadinstitute.org/software/igv/download
[93] http://www-huber.embl.de/users/anders/HTSeq/doc/count.html
[94] http://www.bioconductor.org/packages/release/bioc/html/GenomicRanges.html
[95] http://www.bioconductor.org/packages/release/bioc/html/Rsamtools.html
[96] http://www.bioconductor.org/packages/release/bioc/html/biomaRt.html
[97] http://www.bioconductor.org/packages/release/data/experiment/html/pasilla.html
[98] http://www.bioconductor.org/packages/release/bioc/html/DESeq.html
[99] http://www.ebi.ac.uk/~mar/courses/RNAseq_all.tar.gz

Tutorials

- Course materials available at the Bioconductor website[100]
- Online training resources at the EBI website[101]
- R and Bioconductor tutorial by Thomas Girke[102]
- Do not forget to check the documentation for the packages used in the practical!

Cheat sheets

- R reference card[103]
- Unix comand line cheat sheet[104]

Aknowledgments

This tutorial has been inspired on material developed by Ângela Gonçalves, Nicolas Delhomme, Simon Anders and Martin Morgan, who I would like to thank and acknowledge. Special thanks must go to Ângela Gonçalves, with whom I started teaching, and Gabriella Rustici, for always finding a way to organise a new course.

[100] http://www.bioconductor.org/help/course-materials/
[101] http://www.ebi.ac.uk/training/online/course-list?topic%5B%5D=13&views_exposed_form_focused_field=
[102] http://manuals.bioinformatics.ucr.edu/home/R_BioCondManual
[103] http://cran.r-project.org/doc/contrib/Short-refcard.pdf
[104] http://sites.tufts.edu/cbi/files/2013/01/linux_cheat_sheet.pdf