3-Alignment, RSEM

TOPICS	Alignment MultiQC RSEM STAR docker building
DATE	@March 23, 2022 2:00 PM
LAST EDITED BY	
LAST EDITED TIME	@April 11, 2022 9:04 AM
MADE BY	
PROF	Calogero
Recording	Calogero Lesson 3.m4a
STATUS	☑

Alignment

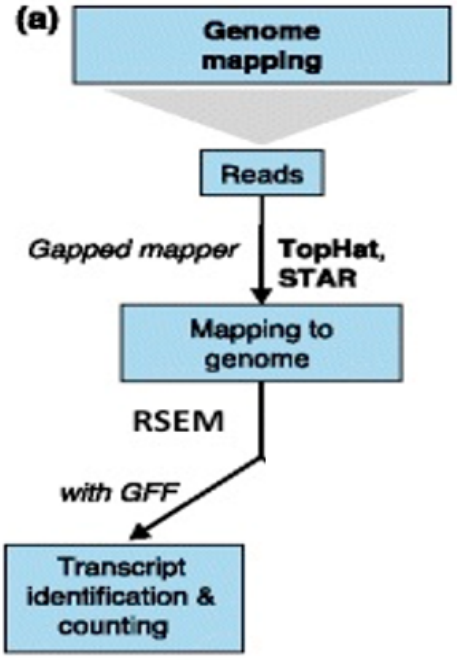
so next step is the alignment of the reads to the genome, map the data to the genome

We can do quality control of data that came out from **bulk RNA sequencing** (of a pool of cells). In order to perform alignment, the material that is needed is the following:

- fastqc file to be aligned (reads)
- reference genome
- alignment algorithm
- annotation information

Therefore, reads should be first of all aligned on a reference genome (also possible to use a transcriptome, but in this course, the focus will be on genomes in order to retrieve more precise information), to carry out the so-called **genome mapping**. In order to do genome mapping, reads (thus all the related information) are mapped to the reference genome using generally the **STAR** aligner (gold standard for transcriptomic analyses).

At this point, another software is used to evaluate the overall composition of isoforms present in the different gene loci, and then this information is converted into *counts* related to the isoforms, and later on into counts related to the genes (as the counts related to the gene are nothing but the summary of all the counts related to each single isoform). To be able to perform this kind of analysis, a previous annotation is required (which is given by GFF or GTF files).

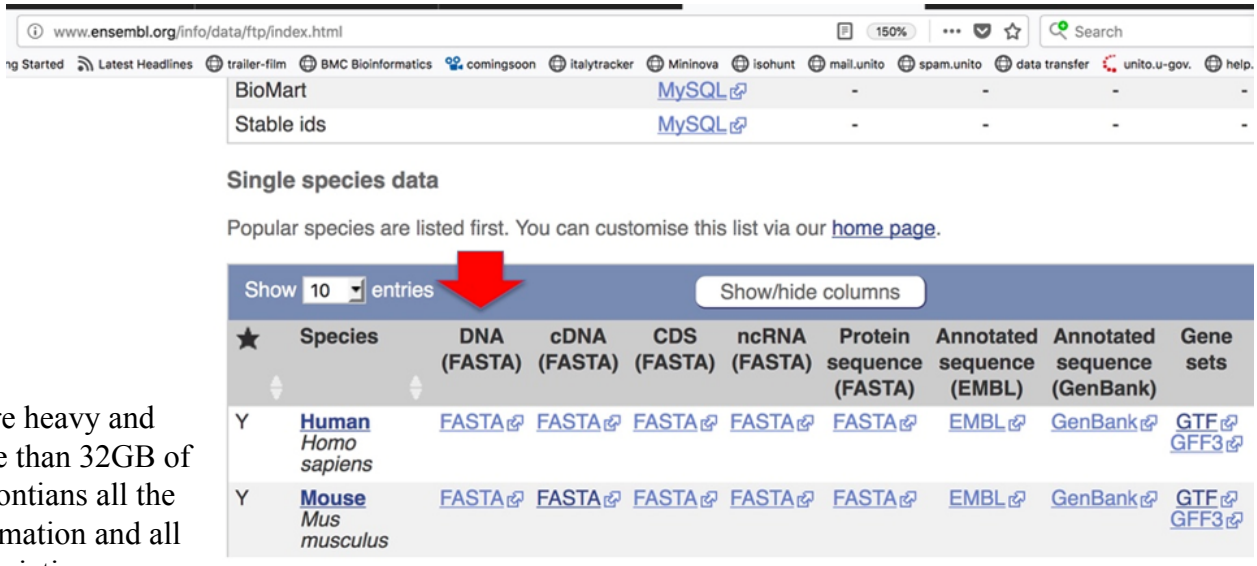


Reference genomes are already available, and the most used one of course is the human genome (the most recent version is the hg38). Every 5 years there generally is new assembly and release of more recent versions. What does “new assembly” mean? At the beginning, when the first version of the human genome was released, people thought that it was simply necessary to know differences between this reference genome and each different genome: this was an oversimplification though, as nowadays it is well acknowledged that the complexity of the human genome is remarkable. Variations are a lot and are spread all over the chromosomes. The majority of these variations are in regions without genes (structural variations) so they’re not so important; the majority of annotations are related to protein-coding genes and are not so critically affected by this kind of variability. The oversimplified view of a single alignment referring to one, unique human genome is no longer possible: other info referred to all of the alternative versions of the genome (that are now available) are necessary.

file containing the cromosome sequence

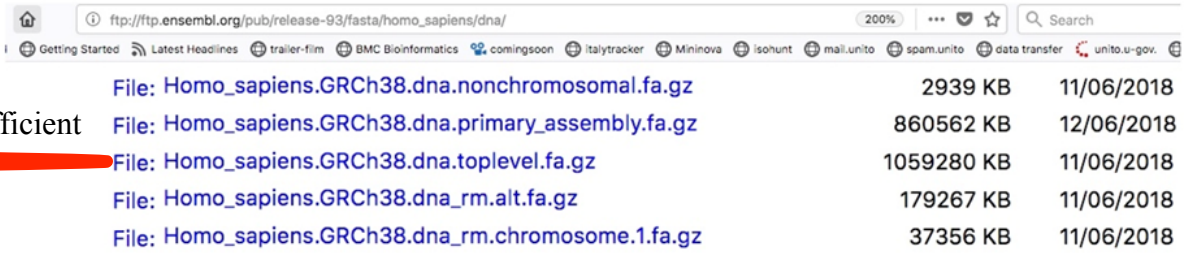
Other than the FASTA file of the reference genome (the mere sequence), the annotation file is necessary in order to perform the alignment. There is more than one repository where annotation files are stored: the main ones are the **Ensembl** (European, most used one. This database is quite gene-centric, everything focuses on the gene loci coordinates: there is the gene name that is the reference, and isoforms are made related to that specific gene name) and the **UCSC** repository (American one, transcript-centric: not the gene locus is considered, but the gene locus is a consequence of all the transcripts that are located to a specific chromosomal position. More tricky to use, as to get associations to genes another database should be used). There is a criticality of the UCSC database that is related to reproducibility: it is possible to download from UCSC a snapshot of the annotation (while the assembly is the same everywhere), but this snapshot refers to a specific moment, and when the database is updated everything changes: impossible to reproduce exactly that specific snapshot of annotation in another moment. On the contrary, the Ensembl database stores all the different annotations: it is always possible to reconstruct every type of annotation wished given the code associated to the specific annotations. Ensembl is the most used database because most people work on

genes, and it is much easier to have annotations referred to genes and then extrapolate the isoforms, than doing the opposite.



this is more heavy and requires more than 32GB of RAM and contians all the coding information and all the variation

this is sufficient



Repository of Ensembl. The data used to perform mapping is the **DNA (FASTA)**. Click on DNA (FASTA) → a list of files appears (bottom window). In particular, the third file from the top (Homo_spaiens.GRCh38.dna.toplevel.fa.gz) contains all the standard genomes plus all the alternative variants that have been released over the years by sequencing different genomes. Conventional expression data analysis does not require this file, it is sufficient to use the second file from the top (**primary_assembly**), because the primary assembly is the one that contains all the coding information, and also because to generate the reference that will be used by the aligner software it is necessary to have memory on the RAM (and the second file is smaller than the third one).

Genome mapping using STAR: an ultrafast universal RNA-seq aligner

Over the years, many aligners have been developed. A specific type of aligner is needed for human data and eukaryotes in general. These aligners should be able to handle the fact that genes are split in different parts (exons and introns), and thus that reads are spread all over the genome with gaps in between: **gapped aligners**. Gapped aligners handle gaps, which is not an issue when doing conventional mapping on bacteria, obviously.

STAR is a gapped aligner that became quite popular owing to its speed: it is possible to map with 12 threads something like 500 million reads per hour (faster than TopHat2 aligner).

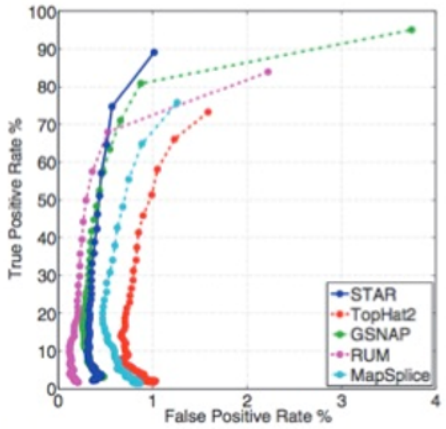


Fig. 2. True-positive rate versus false-positive rate (ROC-curve) for simulated RNA-seq data for STAR, TopHat2, GSNAP, RUM and MapSplice

Table 1. Mapping speed and RAM benchmarks on the experimental RNA-seq dataset

Aligner	Mapping speed: million read pairs/hour		Peak physical RAM, GB	
	6 threads	12 threads	6 threads	12 threads
STAR	309.2	549.9	27.0	28.4
STAR sparse	227.6	423.1	15.6	16.0
TopHat2	8.0	10.1	4.1	11.3
RUM	5.1	7.6	26.9	53.8
MapSplice	3.0	3.1	3.3	3.3
GSNAP	1.8	2.8	25.9	27.0

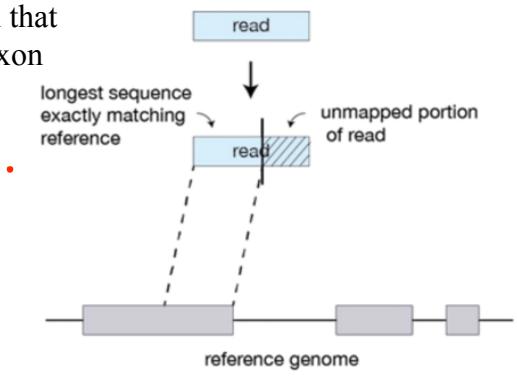
Testo

Actually, the real speed of the aligner is limited by the disk, as the time of writing on the disk is not able to cope with the speed of mapping. Anyway the speed of mapping is around hundred million reads/h.

Even though it is really fast, STAR is still precise and works pretty well on everything that is gapped (genes with introns). On the contrary, it will work poolry on genes without introns. The reason lies in the way the aligner was built.

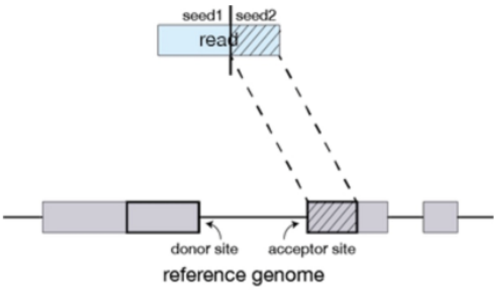
STAR works in a simple way. It is an optimization of the approaches used for alignment, making them faster. The general idea of alignment is that a read and a reference are taken and the alignment giving the higher score is considered (but there is no splitting). In this case, the Maximal Mappable Prefixes (MMPs) are considered: these are the longest matching sequences resulting from a normal alignment, and represent only a portion(s) of the read (which perfectly match one or more locations, which will be exons, on the reference genome). Of course annotation of the reference genome is needed, as to know where exons are located those information are essential. The different MMPs (parts of the read that are mapped separately) are called ‘seeds’, and this first step of the algorithm is called **seed searching**. The first MMP mapped to the genome will be *seed1*.

cuts the reads in a way that is abl to find positions all over the genome in which that piece of read match perfectly to the exon position,

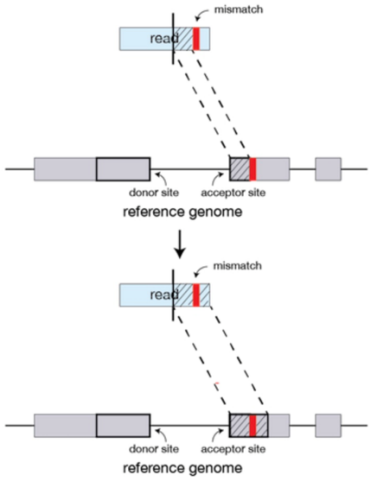


Seed searching. The longest sequence exactly matching the reference is the so called MMP.

When a MMP is found, the unmapped portion of the read (remaining part) should be collocated within hundreds kb from the MMP (because it will be aligned to adjacent exons of the gene): the analysis/search is shrinked a lot. The unmapped portion of the read is like ‘cut’, separated from the *seed1* portion (MMP for previous step) and mapped on the genome within the hundreds kb region: when a perfect match is found, this portion is called *seed2*. So basically the process is repeated every time, with each piece of read, in order to search for the next MMPs by jumping over the introns.



This sequential searching of only unmapped portions of reads underlies the efficiency of the STAR algorithm. The situation described above would be the ideal situation, but *what if there are mismatches at a cetrain point during the search?*



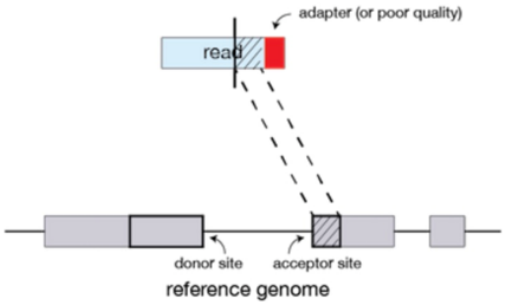
If STAR does not find an exact matching sequence for each part of the read, due to mismatches or indels, the previous MMPs will be extended.

The second part of the read that has to be mapped (where the mismatch is) is split in two: a part upstream to the mismatch and the part starting from the mismatch. The aligner is able to find a perfect match for the upstream portion, while for the mismatich-containg part the alinger ignores the mismatch and takes the rest of the sequence (after the mismatch) that can be perfectly matched on the reference (i.e., it ignores the mismatch and extends the upstream part). Overall the aligner maps after the position in which there is no alignment (and it cuts the read where the mismatch is), so it looks only at perfect matches to proceed.

It may occur that the extension does not give a good alignment. In this instance, the poor quality or adapter sequence (or other contaminating sequence) is soft clipped.

Let’s consider short fragments (reads). Adapters are synthetic sequences absent in the genome, so they won’t map on it. So if the red portion is related to the adpater, it won’t map anywhere and thus it is not considered: only the part of the read that is upstream to the adapter will map somewhere in the reference genome. The red part won’t be taken into account in the quality evaluation of the mapping score (as this part is

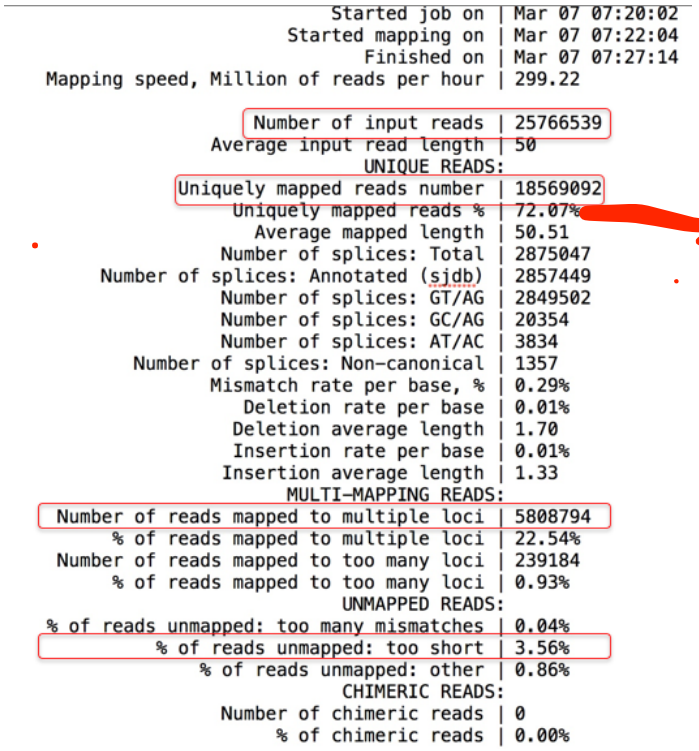
not part of the genome). If there are 150 nucleotides in the read, and 75 nucleotides do not map on the reference because they're not in the genome (as part of the adapter sequence), if these are considered in the quality evaluation they would cause a dramatic drop in the score (only 50% of the read will map), thus they are discarded, and the read will be of only 75 nucleotides (100% mapping, higher score).



Removing the adapters also gives extra information: it tells how big the insert of the unknown DNA is.

The approach so far described is done read by read (each read follows this approach: read split in pieces, and there can be either match or mismatch).

STAR gives extra information. In the folder 'dataset1' of the previous lecture, which contains 14 different samples, there is a logfinal.out file (see figure below) that is the output of the mapping information that come from STAR (generated sample-by-sample). This is one of the parts of the file read by MultiQC:



```
                         Started job on | Mar 07 07:20:02
                     Started mapping on | Mar 07 07:22:04
                            Finished on | Mar 07 07:27:14
Mapping speed, Million of reads per hour | 299.22

                       Number of input reads | 25766539
                 Average input read length | 50
                                UNIQUE READS:
              Uniquely mapped reads number | 18569092
                    Uniquely mapped reads % | 72.07%
                      Average mapped length | 50.51
                   Number of splices: Total | 2875047
        Number of splices: Annotated (sjdb) | 2857449
                   Number of splices: GT/AG | 2849502
                   Number of splices: GC/AG | 20354
                   Number of splices: AT/AC | 3834
           Number of splices: Non-canonical | 1357
                  Mismatch rate per base, % | 0.29%
                     Deletion rate per base | 0.01%
                    Deletion average length | 1.70
                    Insertion rate per base | 0.01%
                  Insertion average length | 1.33
                        MULTI-MAPPING READS:
    Number of reads mapped to multiple loci | 5808794
         % of reads mapped to multiple loci | 22.54%
    Number of reads mapped to too many loci | 239184
         % of reads mapped to too many loci | 0.93%
                             UNMAPPED READS:
% of reads unmapped: too many mismatches | 0.04%
          % of reads unmapped: too short | 3.56%
              % of reads unmapped: other | 0.86%
                             CHIMERIC READS:
                   Number of chimeric reads | 0
                        % of chimeric reads | 0.00%
```

these unique mapping reads are necessary coding gene because are unique and map in coding regions

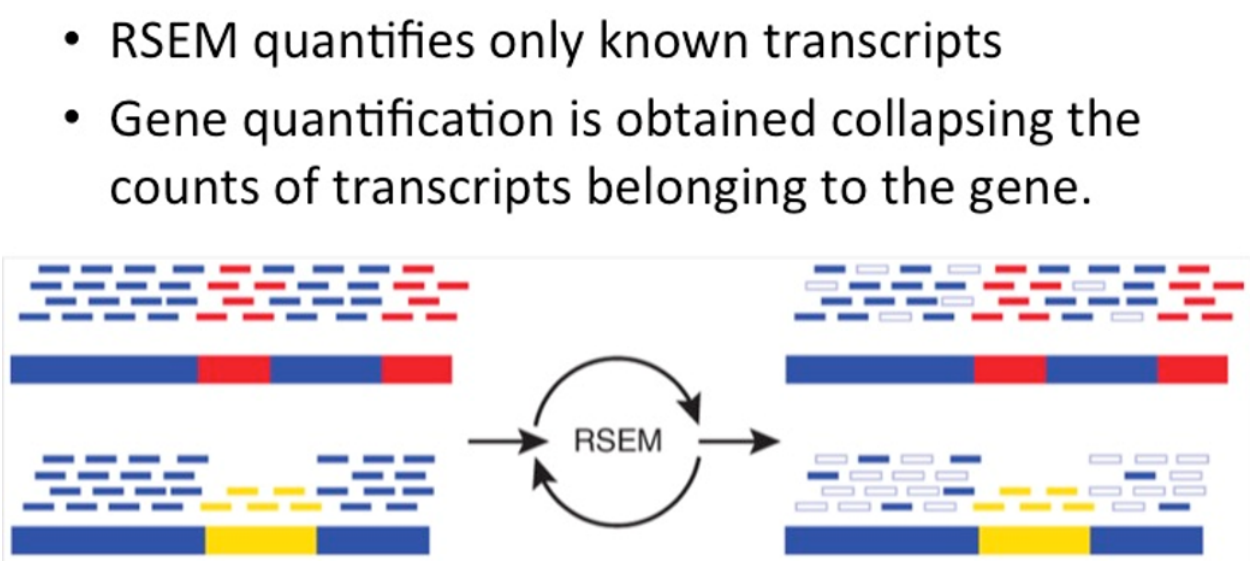
Number of input reads: in this specific case there are roughly 25 million reads. The *average length of each input read* is 50 nucleotides. The fraction of reads that map to the reference genome is 72.07% (*uniquely mapped reads*). In this case reads come from coding genes (transcriptomic experiment, only exonic regions), a very high number of reads that are mapping is expected: it follows that 72% is a little bit low, over 90% would be normally expected. Why there is such a loss? To answer this question, the first cause is given by the *number of reads mapped to multiple loci* (22.54%, not protein coding genes, which instead would map univocally): these are normally due to pieces of RNA mapping to multiple positions over the genome, since actually this is not RNA but contaminating DNA (this can happen when the polyA enrichment is done since some pieces of DNA can have A-rich fragments and thus be isolated together with mRNA). The sample should be cleaned up well from DNA to avoid this or at least reduce this score, thus this percentage suggests a poor cleaning step. The second cause that reduces the percentage of uniquely mapped reads is the *percentage of reads too short to me mapped* (3.56%): when reads are too short, multiple mapping to different positions in the genome occurs. Normally this situation refers to rRNA that is contaminating the mRNA sample → when not only polyA genes are studied, but instead every kind of transcript is considered (also non poly-adenylated ones), rRNA should be removed prior to retratranscription: if this step goes wrong, the number of reads too short to be mapped will finish there. It refers mainly to preparations in which it is necessary to remove rRNA, so this row gives info about rRNA contamination. The point is that when there is rRNA contmaination, since rRNA is the majority of RNA there will be mainly retrotranscription of rRNA, but since on the reference genome there are not ribosomal genes (in normal versions of the genome that are released there are no ribosomal genes, only non-repetitive sequences are present), there is mapping only for some pieces somewhere. Overall, 2 important information come out from this file: (1) info about contamination of DNA, (2) info about contmaination of rRNA when rRNA depletion is carried out. The scores you get for these two rows should warn you about contaminations. Let's suppose a mycoplasm contamination in a human cell line: its genome won't align on the human genome of course, and it will cause a drop in the *uniquely mapped reads %*. Another critical point, different from bacterial/yeast contamination, is cross-contamination with another cell line (that may occur working under hoods).

Once the mapping has been generated, the results are stored in a **BAM** file (binary version of a tab-delimited file SAM: the mapping info it contains are not associated to the name of genes/transcripts in each specific location). The BAM file is just an intermediate file with all mapping information of the peaks of reads in specific areas of the genome. These should be associated to isoforms and genes then. To do so, a commonly used software is **RSEM**.

dropping

RSEM

This software handles the fact that among isoforms there are regions that are in common. In the following picture, there are 2 isoforms from the same genomic locus: the blue parts are the common ones, while red and yellow portions allow to distinguish them. If the aim is to quantify the isoform, the problem is to handle common regions of the different isoforms. RSEM estimates the relative expression of the various isoforms on the basis of the specific areas that are isoform-specific. Practically, in the given example, the red region has approximately twice reads compared to the yellow one (2:1 ratio between the two isoforms). So RSEM takes the blue reads (common part), and assigns the double of the number to the read isoform and half of the number to the yellow isoform. This means that RSEM estimates on the specific sequences that are isoform-specific the ratio amount of various isoforms, and reads are assigned on the basis of this ratio. *Which is the limit?* The problem is related to the different level of coverage that there could be, that strongly affects the quantification → with 20 million reads it may be that the yellow isoform is not seen, that exon is too short to be catched with that coverage. Higher levels of coverage can be necessary to appreciate all isoforms. The reason why few people do work on isoforms using second generation sequencing is because of this problem. It follows that working on genes is different than working on isoforms → firt of all the existence of an isoform should be demonstrated, and then by removing only that isoform a biological effect should be isoform. Working on isoforms is much more complicated.

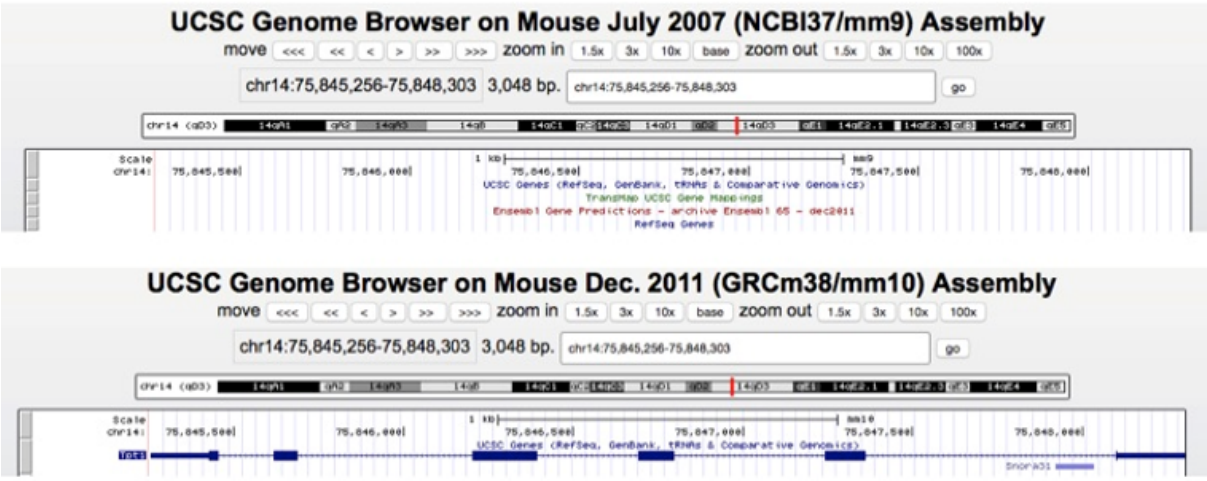


Shared (blue) sequences, unique (red, yellow) sequences. To estimate transcript abundances, RNA-seq reads (short bars) are first aligned to the transcript sequences (long bars, bottom). Unique regions of isoforms (red, yellow ones) will capture uniquely mapping RNA-seq reads, and shared sequences between isoforms (blue) will capture multiply-mapping reads. RSEM estimates the most likely relative abundancies of the transcripts and then fractionally assigns reads to the isoforms based on these abundancies.

When the isoforms are not known and should be discovered, the use of short reads is not even considered. Normally in these cases gene and isoform analysis given the available annotations is evaluated.

For the annotation, in order to use RSEM, another file is needed: this is a GTF file, and it comes associated to a speicfic release of the assembly. **Annotation and assembly go together, you can’t take an annotation and use it on a different assembly:**

- Any time a new genome assembly appears, e.g. mm10 for mouse, annotation information needs to be remapped on the new assembly.



Same physical location on the genome, on the chromosome 14. However, on mm9 for that location there are no genes, while on mm10 there are genes. The reason is that something shifted: maybe that gene was already there, but depending on the assembly some pieces can miss or not. Annotation should be associated to the right assembly, and it is important to know on which assembly the mapping is made.

This is an example of how files are organized: they are organized in a table. UCSC organization in the upper part, ENSEMBL below. A peculiarity of the UCSC organization is that everything is called ‘gene_id’, however the gene_id is the identifier of a specific isoform present in that specific locus. On the contrary, in the ENSEMBL organization there are various elements: exons, upper part, gene bodies, etc. All info are available and it’s much easier to identify the information of interest. Moreover the ENSEMBL annotation has biotypes, which mean coding genes (only genes associated to proteins), but then there are long non-coding genes, short non-coding genes: a lot of biotypes that allow to identify specific subpopulatons of RNA annotated all over the genome.

UCSC

chr1	mm10_knownGene	exon	3205904	3207317	0	-	.	gene_id "uc007aet.1"; transcript_id "uc007aet.1";
chr1	mm10_knownGene	exon	3213439	3215632	0	-	.	gene_id "uc007aet.1"; transcript_id "uc007aet.1";
chr1	mm10_knownGene	stop_codon	3216022	3216024	0	-	.	gene_id "uc007aeu.1"; transcript_id "uc007aeu.1";
chr1	mm10_knownGene	CDS	3216025	3216968	0	-	2	gene_id "uc007aeu.1"; transcript_id "uc007aeu.1";
chr1	mm10_knownGene	exon	3214482	3216968	0	-	.	gene_id "uc007aeu.1"; transcript_id "uc007aeu.1";
chr1	mm10_knownGene	CDS	3421702	3421901	0	-	1	gene_id "uc007aeu.1"; transcript_id "uc007aeu.1";
chr1	mm10_knownGene	exon	3421702	3421901	0	-	.	gene_id "uc007aeu.1"; transcript_id "uc007aeu.1";
chr1	mm10_knownGene	CDS	3670552	3671348	0	-	0	gene_id "uc007aeu.1"; transcript_id "uc007aeu.1";
chr1	mm10_knownGene	start_codon	3671346	3671348	0	-	.	gene_id "uc007aeu.1"; transcript_id "uc007aeu.1";
chr1	mm10_knownGene	exon	3670552	3671498	0	-	.	gene_id "uc007aeu.1"; transcript_id "uc007aeu.1";
chr1	mm10_knownGene	exon	3648311	3650509	0	-	.	gene_id "uc007aev.1"; transcript_id "uc007aev.1";
chr1	mm10_knownGene	exon	3658847	3658904	0	-	.	gene_id "uc007aev.1"; transcript_id "uc007aev.1";
chr1	mm10_knownGene	stop_codon	4292981	4292983	0	-	.	gene_id "uc007aew.1"; transcript_id "uc007aew.1";
chr1	mm10_knownGene	CDS	4292984	4293012	0	-	2	gene_id "uc007aew.1"; transcript_id "uc007aew.1";
chr1	mm10_knownGene	exon	4290846	4293012	0	-	.	gene_id "uc007aew.1"; transcript_id "uc007aew.1";
chr1	mm10_knownGene	CDS	4351910	4352081	0	-	0	gene_id "uc007aew.1"; transcript_id "uc007aew.1";
chr1	mm10_knownGene	exon	4351910	4352081	0	-	.	gene_id "uc007aew.1"; transcript_id "uc007aew.1";

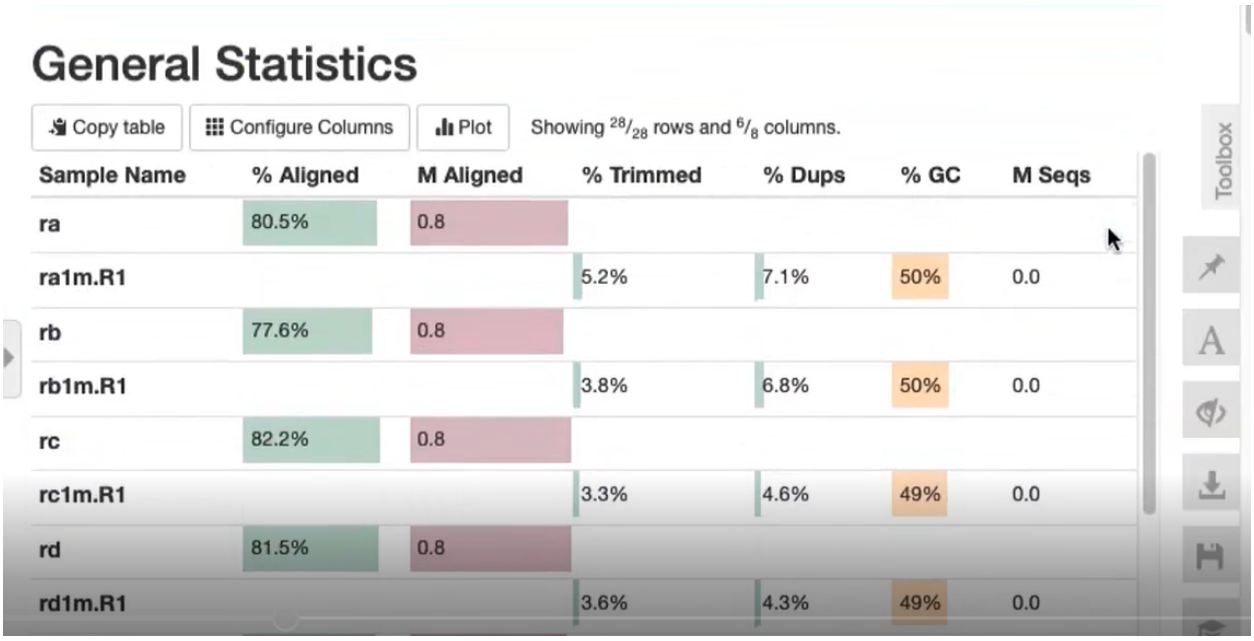
ENSEMBL

#genome-build GRCm38.p4								
#genome-version GRCm38								
#genome-date 2012-01								
#genome-build-accession NCBI:GCA_000001635.6								
#genebuild-last-updated 2015-12								
1	havana	gene	3073253	3074322	.	+	.	gene_id "ENSMUSG00000102693"; gene_version "1"; gene_name "4933401J01Rik"; gene_source "havana"; gene_bi
1	havana	transcript	3073253	3074322	.	+	.	gene_id "ENSMUSG00000102693"; gene_version "1"; transcript_id "ENSMUST00000193812"; transcript_version "1"
1	havana	exon	3073253	3074322	.	+	.	gene_id "ENSMUSG00000102693"; gene_version "1"; transcript_id "ENSMUST00000193812"; transcript_version "1"
1	ensembl	gene	3102016	3102125	.	+	.	gene_id "ENSMUSG00000064842"; gene_version "1"; gene_name "Gm26206"; gene_source "ensembl"; gene_bioty
1	ensembl	transcript	3102016	3102125	.	+	.	gene_id "ENSMUSG00000064842"; gene_version "1"; transcript_id "ENSMUST00000082908"; transcript_version "1"
1	ensembl	exon	3102016	3102125	.	+	.	gene_id "ENSMUSG00000064842"; gene_version "1"; transcript_id "ENSMUST00000082908"; transcript_version "1"
1	ensembl_ha	gene	3205901	3671498	.	-	.	gene_id "ENSMUSG00000051951"; gene_version "5"; gene_name "Xkr4"; gene_source "ensembl_havana"; gene_bio
1	havana	transcript	3205901	3216344	.	-	.	gene_id "ENSMUSG00000051951"; gene_version "5"; transcript_id "ENSMUST00000162897"; transcript_version "1"
1	havana	exon	3213609	3216344	.	-	.	gene_id "ENSMUSG00000051951"; gene_version "5"; transcript_id "ENSMUST00000162897"; transcript_version "1"
1	havana	exon	3205901	3207317	.	-	.	gene_id "ENSMUSG00000051951"; gene_version "5"; transcript_id "ENSMUST00000162897"; transcript_version "1"
1	havana	transcript	3206523	3215632	.	-	.	gene_id "ENSMUSG00000051951"; gene_version "5"; transcript_id "ENSMUST00000159265"; transcript_version "1"
1	havana	exon	3213439	3215632	.	-	.	gene_id "ENSMUSG00000051951"; gene_version "5"; transcript_id "ENSMUST00000159265"; transcript_version "1"

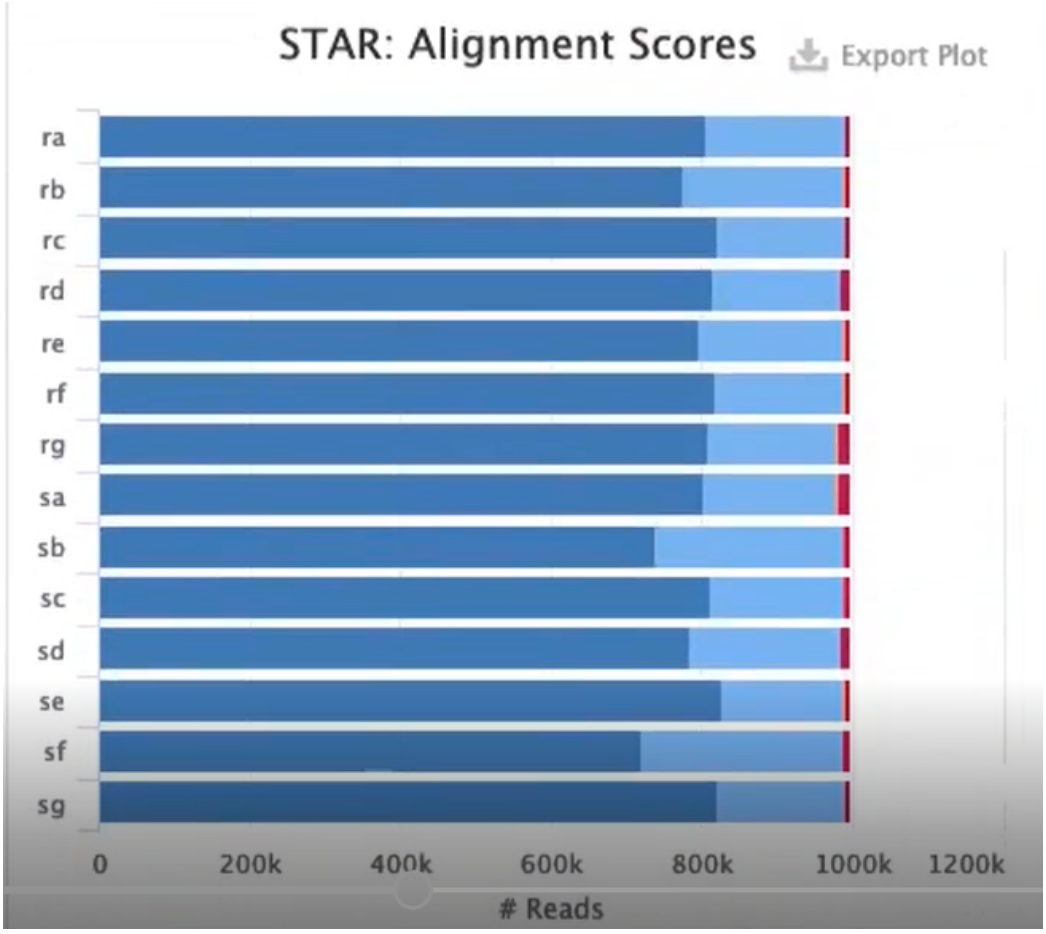
Annotation files

If we have the fastq file, and its quality is good, and then we have STAR and RSEM: we put everything together and we do mapping. **Trimming** should have been done on data previously, which is a step necessary to know the average size of the insert. All these information are summarized in a **multiqc file**.

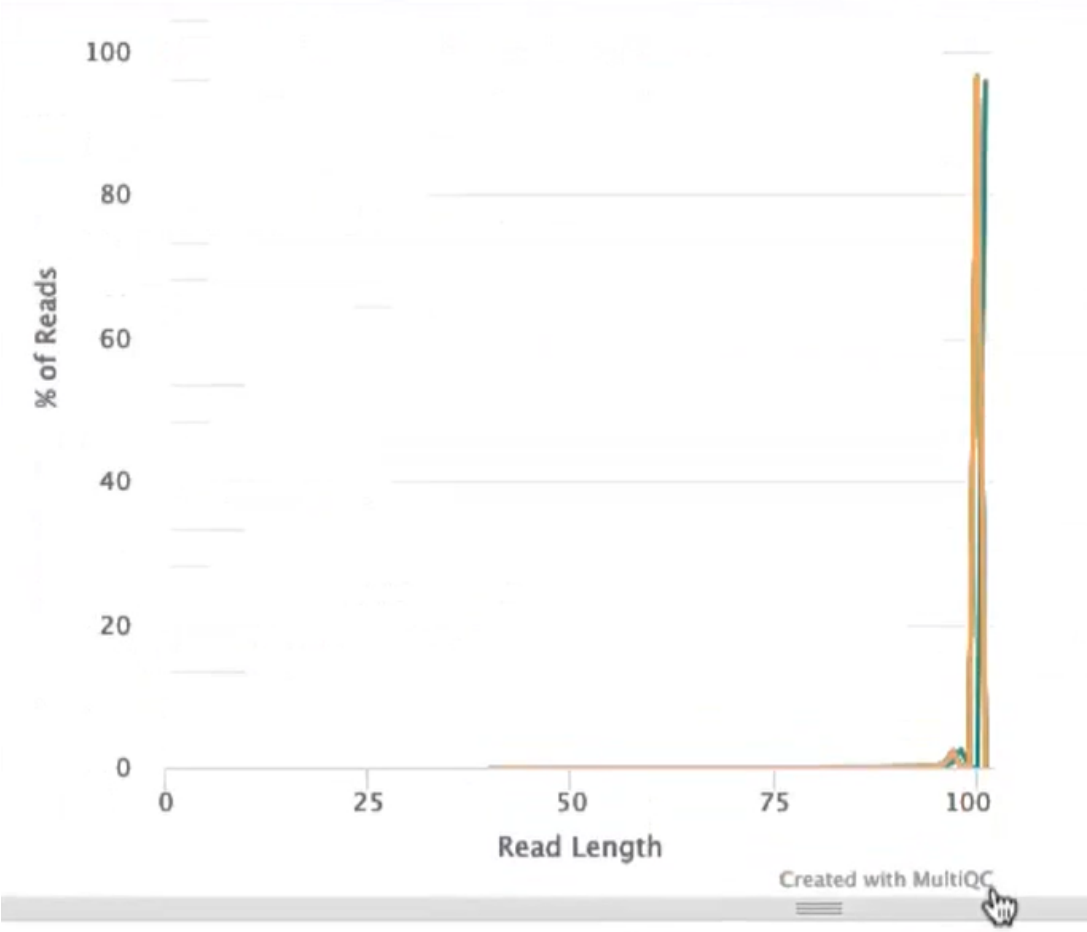
Until now we saw what we need to go from fastQ to count table (actually to go to counts referring to the gene of 1 single experiment, and we have multiple experiments referring to our experimental setting). Namely, we have the fastQ, we run FastQC and we get the results. At this point, the trimming is performed and with the obtained results the mapping with STAR can be done, and finally the counting is performed using RSEM. Once we have all these information, which refer to a single sample, the MultiQC gives summary of all the results, that refer to all the samples (so it gives an overview of the whole). Here there is the **MultiQC report**:



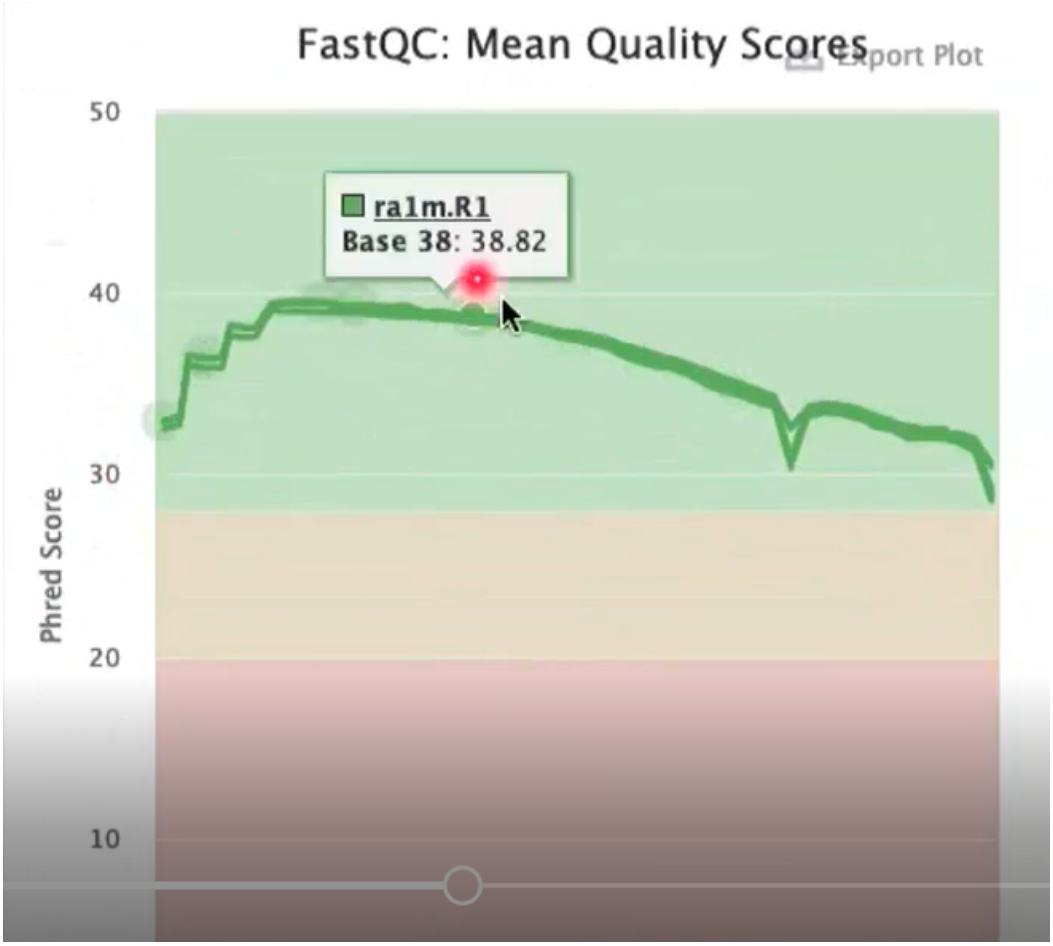
80.5% mapping alignment. 0.8 is the million reads aligned. The sample is called “ra1m” beacause it’s a subsample of the initial dataset (and we took only 1 million reads to make a quick mapping, so “1m”, just to have an idea of the analysis). The “.R1” part refers to the name of the fastq file sued for the mapping. We also have info about trimming, number of duplicates, GC content, etc.



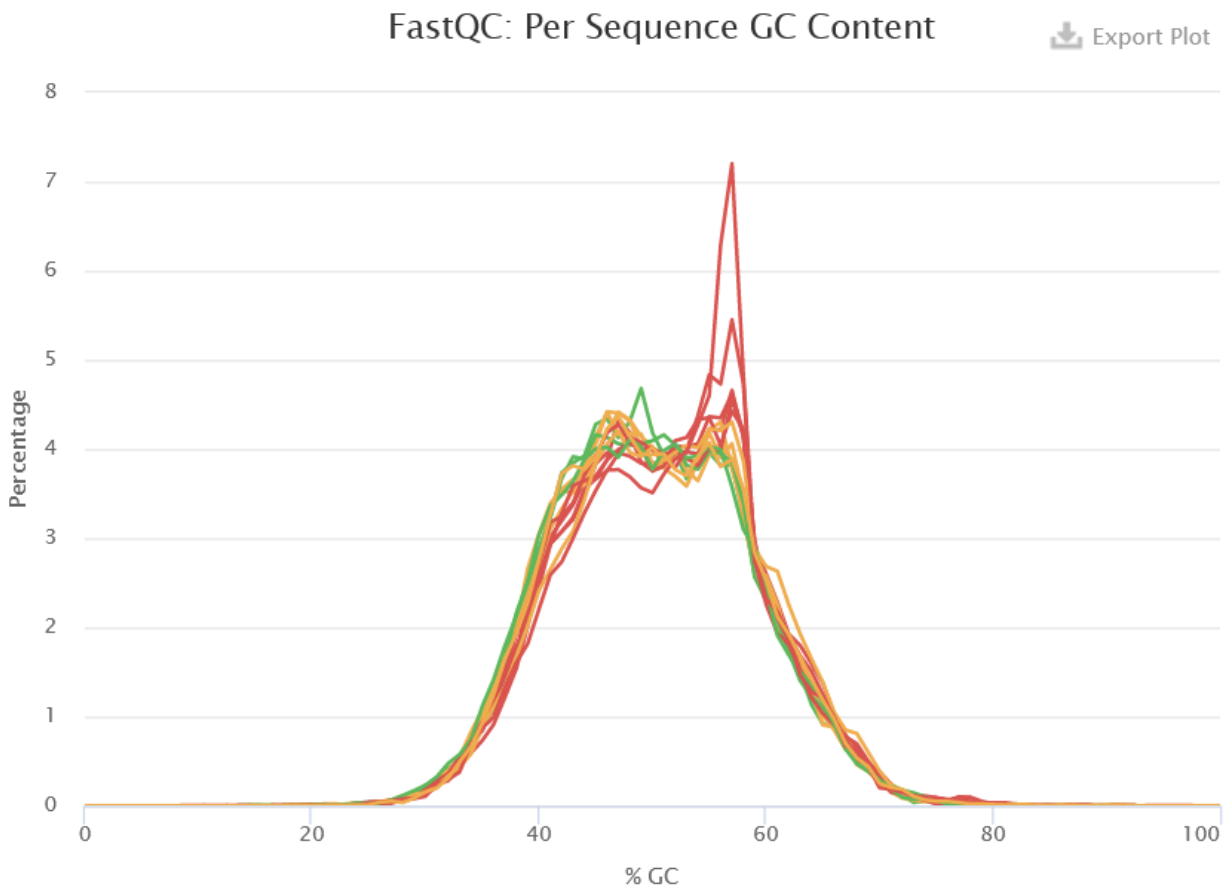
Going ahead, these are the info related to the reads, In this case the situation is easy as there are 1 millionreads for each sample, but it could be that samples have different reads number. If the quality of RNA is not homogeneous, also the yield of sequencing changes and thus we get a different number of reads. In this graph, blue refers to uniquely mapped reads, light blue are those reads mapped to multiple loci, while unmapped reads are the red ones. The overall behaviour of the samples, the quality of RNA, is pretty good: same amount of non-mapping reads and roughly also same amount of mapping reads.



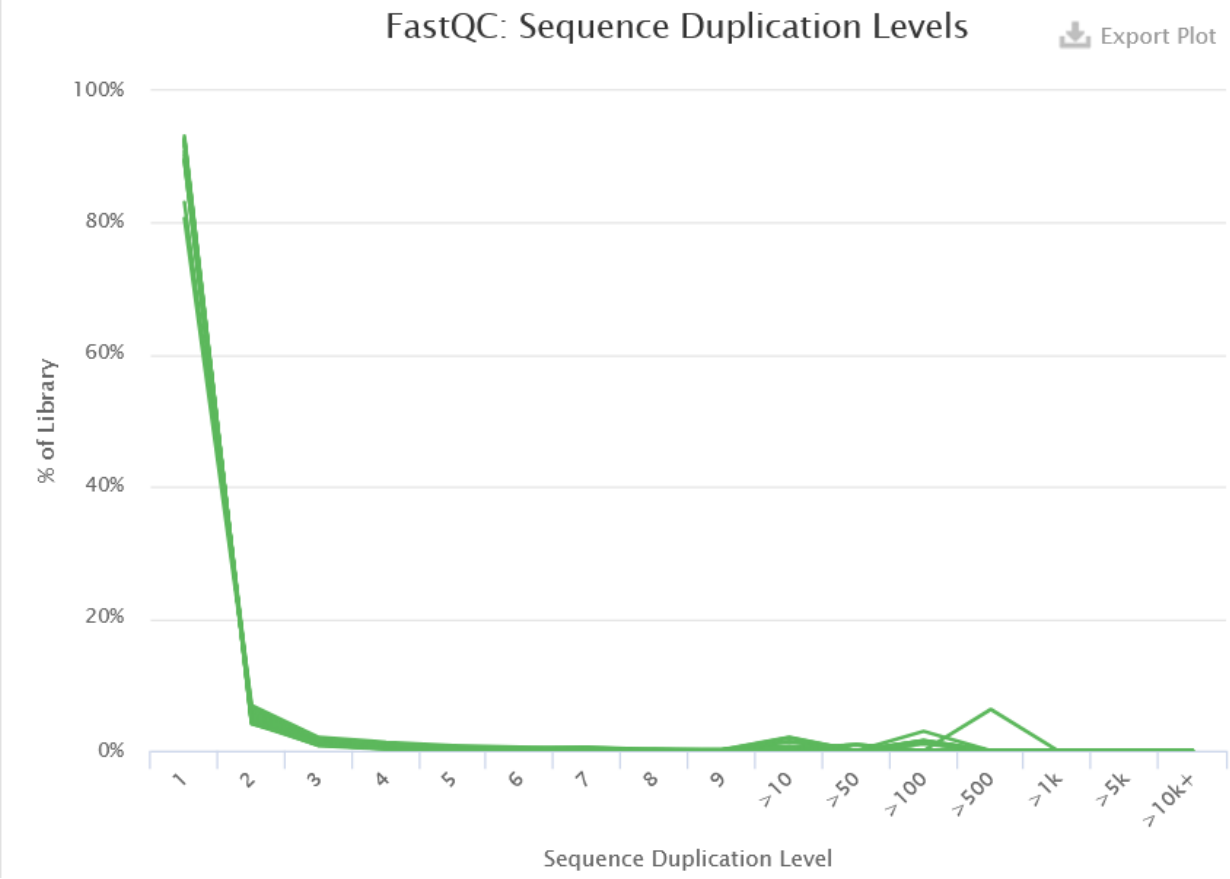
Skewer: read length distribution after trimming. This tells that we are trimming very little as only few nucleotides are trimmed: 100 nucleotides length, just the last few nucleotides are those trimmed out as they belong to the adapter. This means that the insert was roughly in the range of 95 nucleotides, which is a good length (considering that the optimal max length from a conventional polyA purified mRNA is about 150 nucleotides).



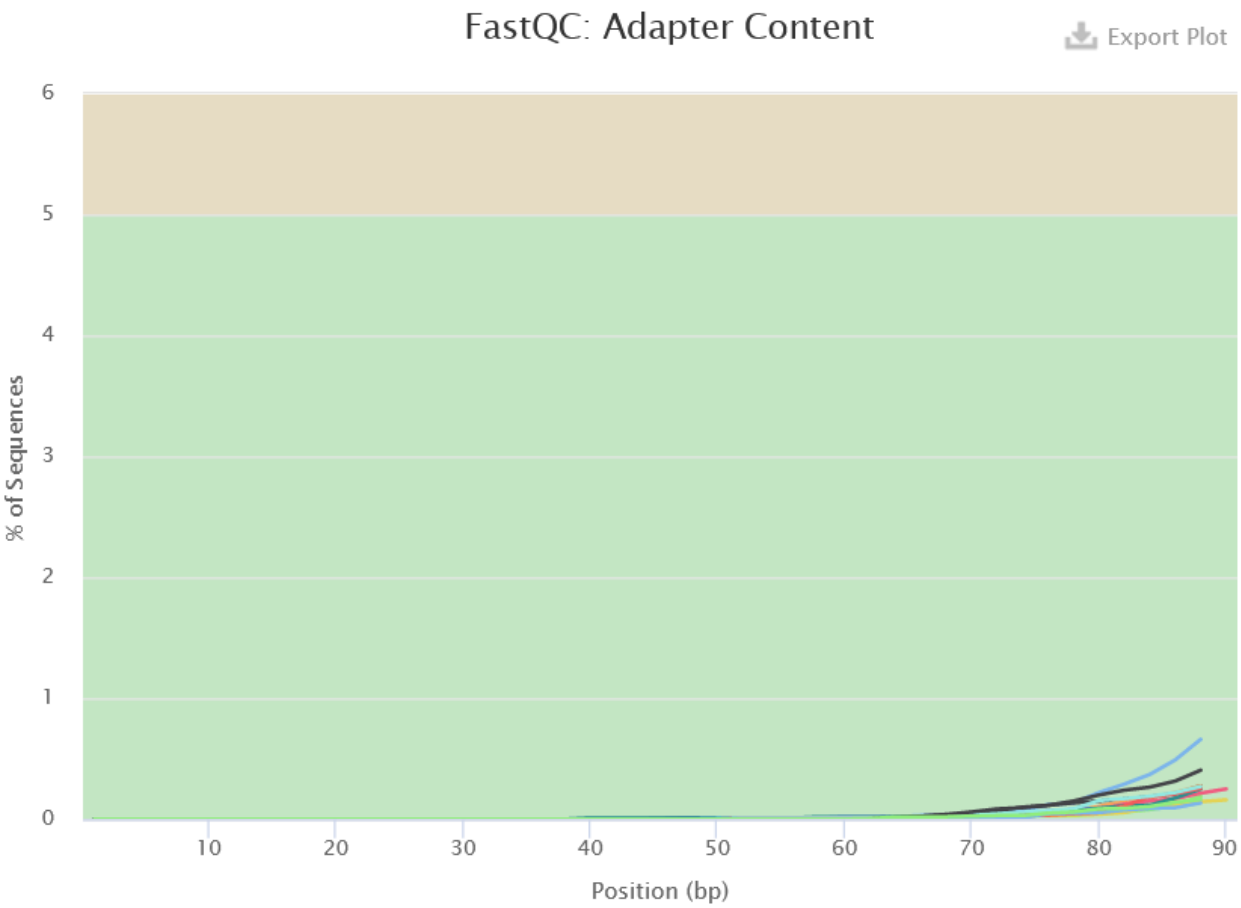
All the reads behave in the same way, there is homogeneity even when the quality is very poor. On the contrary, when just one or few samples are dishomogeneous compared to the others, analysis cannot be done.



GC content has a weird behaviour but homogenous all over the samples, so as before we don't care so much.



Since we did the chemical fragmentation, it is expected to find some duplications simply because of the characteristics of the fragmented RNA (as seen last time).



The adapters are present in a low percentage of sequence (below 1%) and are over 70 nucleotides. Thus they're a very negligible part of the reads (less of 1% needs to be trimmed to remove adapters). The library was relatively large in a way to be longer than the nucleotides sequenced.

Overall, MultiQC gives an overview of all the samples at the same time.

This analysis was done by means of a specific package called *docker4seq,* which embeds in Docker all tools needed to perform the RNAseq analysis (we won't use it because it expects to have Unix or Mac as default interface). We will do the same thing differently though.

Running docker4seq to do this analysis for each single sample is quite simple because it is handled by R. This is the command:

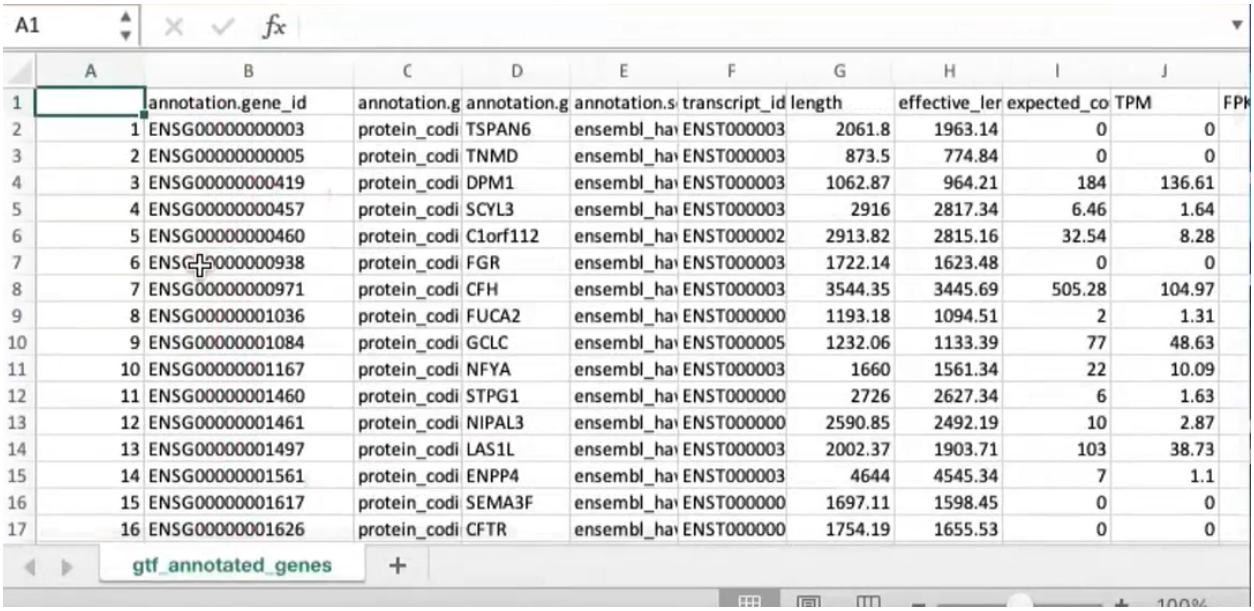
command.R — mapping_w_star

```
library(docker4seq)
rnaseqCounts( group = "docker",
    fastq.folder = getwd(),
    scratch.folder = "/home/rcaloger/scratch",
    threads = 8,
    adapter5= "AGATCGGAAGAGCACACGTCTGAACTCCAGTCA",
    adapter3="AGATCGGAAGAGCGTCGTGTAGGGAAAGAGTGT",
    seq.type = "se",
    min.length = 40,
    genome.folder = "/home/rcaloger/test/genomes/hg38",
    strandness = "none",
    save.bam = FALSE,
    org = "hg38",
    annotation.type = "gtfENSEMBL" )
```

You run a single function basically: each element is important. First the docker4seq library should be loaded. The function is **rnaseqCounts**, which is a wrapper (i.e. it embeds other functions, each one does a different thing: there's a function that retrieves a Docker with FastQC and runs it for each sample, than another one downloads the Docker with Skewer and does the trimming, and then a third one downloads the STAR and perform the alignment and counting, and finally there are some scripts that put everything together in the final format we want). SInce the rnaseqCounts runs over a Docker, we have to specify which kind of permission we are running. Normally we run Docker with the docker user permission, or alternatively we run it as superuser (too risky, don't do it). The function **getwd()** is the path where we are located at that moment (working directory), same as typing the whole path instead. Same concept as pwd of Linux, but here we're in R. Then, the **scratch.folder** is used: the concept of this analysis is that normally data are big and located on a low performing disk (let's say a conventional hard drive), but when trying to do the alignment on a conventional hard drive the duration is 10-fold higher than with a SSD. The idea is to set the scratch folder on the fast disk (data are first copied on the scratch, all the analysis is done there and then everything is put back). **Threads** indicates the number of computing units that we are using (?). **Adapter5** and **adapter3** can be found in literature (full lists of adapters depending on the library and the kit appear, the right ones allow to perform the trimming). **seq.type** has two options: "se" (single ends) or "pe" (paired ends). **min.length** means that if we have reads trimmed and are shorter than 40 nucleotides, these are thrown away. **genome.folder** is the location where there's the reference aligner file ready for mapping: it is not a fasta, but it is a version reorganized by STAR in such a way to have quick access to the genomic sequence (binary form). **Strandness** has 3 options: "none", "forward", "reverse"→ "none" means we don't care (as with poly-A selected genes, which are coding genes and do not overlap). Almost all kits are stranded, so we know which strand we are sequencing (not important for coding genes, which do not overlap, while it matters when analysing also noncoding genes since these are not poly-adenylated and lot of them are on the opposite strand of a coding gene). We need to know the type of strandness, the strand (in general, Illumina kits are reverse), and this reverse willo nly use the strand of the reference to perform the alignment. This allows to assing reads to the two strands independently even if the two reads are coming from the same position on the genome (you read the forward or the reverse depending on which gene is the coding one). **save.bam** is generally set as FALSE because we don't care about the bam file that much (we don't wnat to extract info related to the variants, we are simply counting sequences on specific positions of the genome) and we do ot want to occupy too much space with useless information. Lastly, the kind of annotation we are working with should be specified (**org** and **annotation.type**).

With this command (function) we do everything from the fastQ to the count table. We get an output of the FastQC, an output of the STAR mapping, then there is the output of the trimming, and then the file we are interested in is the *gtf_annotated_genes.results*. This is how it looks like on Excel:

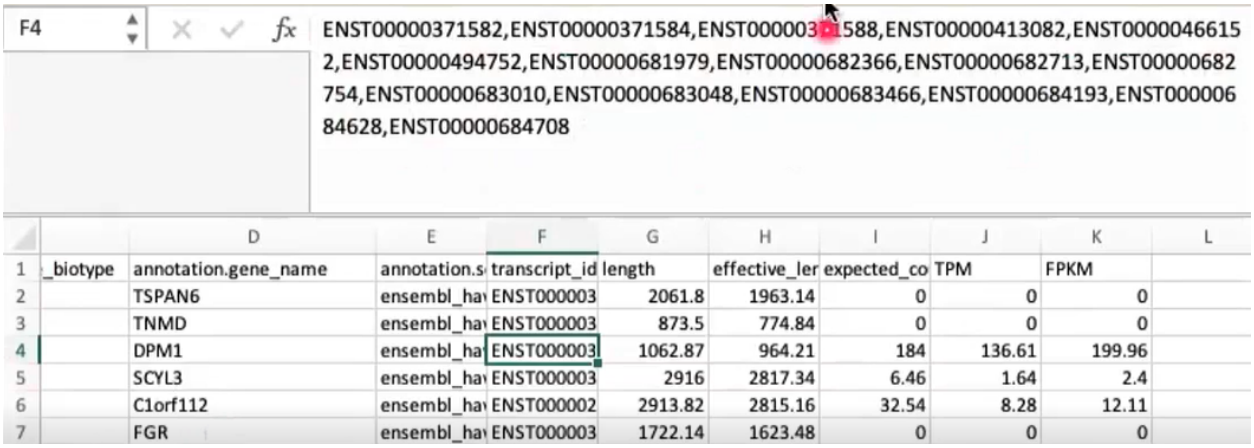
output of fastqc -> file html con nome sample
output Starmapping Log.final.out
output trimming on trimmed.log file
gtf_annotatedgene results in what we are interested in



	A	annotation.gene_id	annotation.g	annotation.g	annotation.s	transcript_id	length	effective_ler	expected_co	TPM	FPK
2	1	ENSG00000000003	protein_codi	TSPAN6	ensembl_hav	ENST000003	2061.8	1963.14	0	0	
3	2	ENSG00000000005	protein_codi	TNMD	ensembl_hav	ENST000003	873.5	774.84	0	0	
4	3	ENSG00000000419	protein_codi	DPM1	ensembl_hav	ENST000003	1062.87	964.21	184	136.61	
5	4	ENSG00000000457	protein_codi	SCYL3	ensembl_hav	ENST000003	2916	2817.34	6.46	1.64	
6	5	ENSG00000000460	protein_codi	C1orf112	ensembl_hav	ENST000002	2913.82	2815.16	32.54	8.28	
7	6	ENSG00000000938	protein_codi	FGR	ensembl_hav	ENST000003	1722.14	1623.48	0	0	
8	7	ENSG00000000971	protein_codi	CFH	ensembl_hav	ENST000003	3544.35	3445.69	505.28	104.97	
9	8	ENSG00000001036	protein_codi	FUCA2	ensembl_hav	ENST000000	1193.18	1094.51	2	1.31	
10	9	ENSG00000001084	protein_codi	GCLC	ensembl_hav	ENST000005	1232.06	1133.39	77	48.63	
11	10	ENSG00000001167	protein_codi	NFYA	ensembl_hav	ENST000003	1660	1561.34	22	10.09	
12	11	ENSG00000001460	protein_codi	STPG1	ensembl_hav	ENST000000	2726	2627.34	6	1.63	
13	12	ENSG00000001461	protein_codi	NIPAL3	ensembl_hav	ENST000000	2590.85	2492.19	10	2.87	
14	13	ENSG00000001497	protein_codi	LAS1L	ensembl_hav	ENST000003	2002.37	1903.71	103	38.73	
15	14	ENSG00000001561	protein_codi	ENPP4	ensembl_hav	ENST000003	4644	4545.34	7	1.1	
16	15	ENSG00000001617	protein_codi	SEMA3F	ensembl_hav	ENST000000	1697.11	1598.45	0	0	
17	16	ENSG00000001626	protein_codi	CFTR	ensembl_hav	ENST000000	1754.19	1655.53	0	0	

gtf_annotated_genes

The second column is the **gene identifier** in ENSEMBL, and tells which is the code associated to that specific gene on ENSEMBL. Then there is the **biotype** (protein_coding in this case). Then there is column with gene symbols (**annotation_gene_name**). **TPM** and **FPKM** are sort of normalization that will be discussed later on in the course. The **expected_counts** is the output of RSEM (which gets for each isoform the number of counts, sums them together and gives the expected counts for the gene given all the annotated transcripts) → looking at **transcript_id,** in this column all transcripts associated to that specific genomic locus (indicated by the gene name column) are annotated: by putting together all those transcripts we get the counts associated to that specfiic gene:



F4: ENST00000371582,ENST00000371584,ENST00000371588,ENST00000413082,ENST00000466152,ENST00000494752,ENST00000681979,ENST00000682366,ENST00000682713,ENST00000682754,ENST00000683010,ENST00000683048,ENST00000683466,ENST00000684193,ENST00000684628,ENST00000684708

	_biotype	annotation.gene_name	annotation.s	transcript_id	length	effective_ler	expected_co	TPM	FPKM
2		TSPAN6	ensembl_hav	ENST000003	2061.8	1963.14	0	0	0
3		TNMD	ensembl_hav	ENST000003	873.5	774.84	0	0	0
4		DPM1	ensembl_hav	ENST000003	1062.87	964.21	184	136.61	199.96
5		SCYL3	ensembl_hav	ENST000003	2916	2817.34	6.46	1.64	2.4
6		C1orf112	ensembl_hav	ENST000002	2913.82	2815.16	32.54	8.28	12.11
7		FGR	ensembl_hav	ENST000003	1722.14	1623.48	0	0	0

So the output we are interested in (since we're going to work with it) is this *gtf_annotated_genes.results* (it contains counts of the gene, and we'll work mainly with gene counts).

Docker

It is possible to download a docker from the web. However it is also possible to build our own docker and put whatever we want in it. *How can we build a docker container?* There are relatively few commands needed.

Docker container

There are already some dockers in the docker repository that you can download from the internet (don't need to start from scratch). The command for downloading a docker container is:

```
docker pull (name of the container)

In our specific example:
docker pull ubuntu:21.04 (21.04 is the tag that indicates the specific version that we want to download)
```

⚠️ pull the docker ubuntu version 21.04 (is not the latest one)

To see the images that are present in the docker, you have to write the command:

```
docker images

This command gives a list of all the images that are present in the local repository
```

Building a docker is done to render life easier. In principle, each of you has to build your own docker and then each time that you have to run the analysis you can use the docker that contains all the software and the scripts.

To perform data analysis we will use R as coding language and R library that work only in a Linux environment. So, the idea is to build a docker container with **Ubuntu** (a virtual machine that creates a Linux environment on Windows). R is changing every 6 months and is not necessary always to have the latest version if the software works also in the previous versions. Moreover, the latest version of Ubuntu is 21.10, but is not always a good idea to use the latest one since it is the one in which not all the options have been tested yet.

💡 Google is our best friend and it is strongly recommended to copy.

When do we have to change version? We have to change version when we have to run some software that can be run only in the more updated version. Normally you don't have to change and you change to a new version only if somebody tells you to do it for the quality of the results obtained.

If you want to run a docker container you have to use this command:

```
docker run -i -t ubuntu:21.04

-i means interactively
-t means that you are going to have a terminal that allow interacting with the docker container
```

Running a docker container means that the docker is not an image anymore, but becomes a docker that is running in our computer (in the RAM). Until the docker is an image, it is a piece of software, once you run it becomes a container that is running into the RAM and you are actually going to use it.

If you execute `docker run  -i -t ubuntu:21.04`, you get an # simbol and you are inside the docker. This is visible because you are in a user that is root@.....

Now you are inside the docker and this means that you are actually in a Linux environment and all the Linux commands will work.

> 💡 If you type `docker ps` you know which are the docker that is running at the moment.

When you are inside the docker (it is visible for the # simbol) you can type `ls` and see what is inside. Then you can go inside the home by typing `cd /home`. So now you are using your computer but actually, you are inside a docker container. If you type `exit` you are going out of the docker container and then if you type `docker ps` you will see that there aren't anymore images running.

```
C:\Users\Utente\Desktop>docker run -i -t ubuntu:21.04
root@9ee5dbe8f98c:/# ls
bin  boot  dev  etc  home  lib  lib32  lib64  libx32  media  mnt  opt  proc  root  run  sbin  srv  sys  tmp  usr  var
root@9ee5dbe8f98c:/# ls/home
bash: ls/home: No such file or directory
root@9ee5dbe8f98c:/# ls /home
root@9ee5dbe8f98c:/# cd /home
root@9ee5dbe8f98c:/home# exit
exit

C:\Users\Utente\Desktop>docker ps
CONTAINER ID   IMAGE     COMMAND   CREATED   STATUS    PORTS     NAMES

C:\Users\Utente\Desktop>
```

When you are inside the docker, you actually want ot perform some action.

You can do:

```
apt-get update
```

This command is used to update the Ubuntu version that you downloaded before. For example, you have downloaded the Ubuntu version 21.04, but this version was made up 2 years ago and of course some library and some issues have been updated over the years. So, to have all the library updated you have to type the command above (`apt-get update`) which will search on internet the ubuntu version 21.04, search if there are some update to do and if there are install the update.

Then you do:

```
apt-get upgrade
```

If the version was already updated you don't obtain an upgrade, while if some changes occurred over the months you upgrade your version to the latest library that it has.

```
C:\Users\Utente\Desktop>docker run -i -t ubuntu:21.04
root@996acfbe7d42:/# apt-get update
Hit:1 http://security.ubuntu.com/ubuntu hirsute-security InRelease
Hit:2 http://archive.ubuntu.com/ubuntu hirsute InRelease
Hit:3 http://archive.ubuntu.com/ubuntu hirsute-updates InRelease
Hit:4 http://archive.ubuntu.com/ubuntu hirsute-backports InRelease
Reading package lists... Done
root@996acfbe7d42:/# apt-get upgrade
Reading package lists... Done
Building dependency tree... Done
Reading state information... Done
Calculating upgrade... Done
0 upgraded, 0 newly installed, 0 to remove and 0 not upgraded.
root@996acfbe7d42:/#
```

⚠️ Don't write any space between apt-get

These two command have to be run (always). Of course, the first time that you start your environment.

There is a code associated to the docker that is the one after root@. This code is an alphanumeric code and it's called **instance**.

We want to run the command as sudo. **Sudo** means the possibility to run a command applied as superuser. Normally, when you install software (especially docker) you prefer to run this as sudo (superuser). Running with sudo means that you are running with the highest priority, but you also can delete everything. The advantage is that if you delete everything and then you exit from the docker run, and you haven't saved anything, you can recover all.

To do that you have to install sudo and so you have to type the command below on the terminal inside the docker:

```
apt-get install sudo
```

```
root@996acfbe7d42:/# apt-get install sudo
Reading package lists... Done
Building dependency tree... Done
Reading state information... Done
sudo is already the newest version (1.9.5p2-2ubuntu3).
0 upgraded, 0 newly installed, 0 to remove and 0 not upgraded.
root@996acfbe7d42:/# sudo ls
bin  boot  dev  etc  home  lib  lib32  lib64  libx32  media  mnt  opt  proc  root  run  sbin  srv  sys  tmp  usr  var
root@996acfbe7d42:/#
```

Now you get sudo and if you type some commands with sudo such as `sudo ls` you see that now sudo is present.

If you now `exit` and then you go inside the docker again, and you type `sudo ls`, it will appear that the sudo doesn't exist, because you have not saved anything.

What happened, basically, is that initially you have runned the docker and you have created a specific instance, then you make the update, upgrade, install sudo and then you get out. Then, when you run again the docker, it will create a new instance that doesn't have anything of what we have done before.

To save the changes we have to remember the instance in which we have made the changes (in this case, install sudo).

If you type `docker ps` you don't see any docker running; but if you type `docker ps -a` it will tell all the docker that were previously closed (means `exit` from the docker).

> ⚠️ The instance is volatile, when you close you don't save the changes.

Now we want to save the changes of the images that is called ubuntu:21.04 with the modification that we made (for example installing sudo).

To do that you have to do :

```
docker commit (instance code) ubuntu:21.04
```

Then you go again inside the docker with `run docker -i -t ubuntu:21.04`, then type `sudo ls` and you see that now you have saved the changes.

Before doing that, you also have to install nano with:

```
sudo apt-get install nano
```

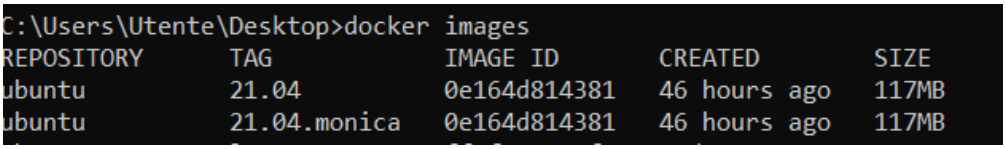
The real name of an image is an alphanumeric code that is called **image ID**.

One problem comes out if you want to redownload the same Ubuntu version: the system will completely reset the ubuntu image that you have modified. To solve this problem, it is possible to save the modified version of ubuntu with your own tag using the `docker tag` command.

```
docker tag (image ID) (new name that you want)

Example
docker tag 0e164d814381 ubuntu:21.04.monica
```

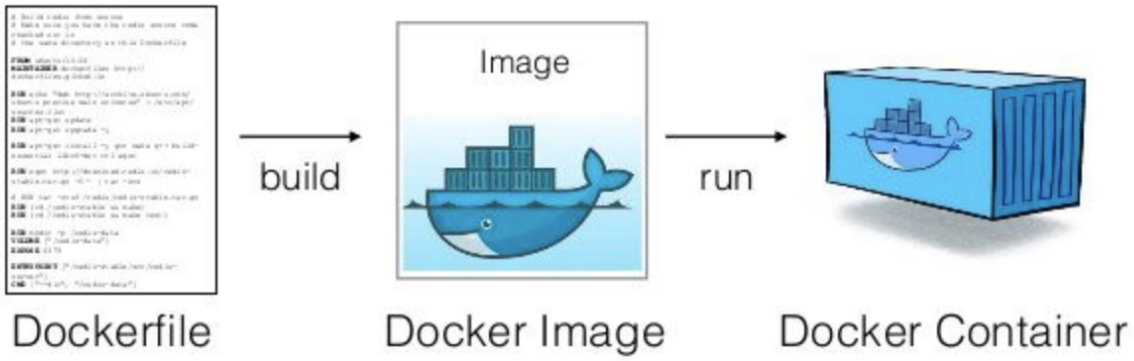
At this point, if you type `docker images` you will have 2 ubuntu images with the same images ID, but different tag. Now, if you redownload ubuntu:21-04 with `docker pull`, it will go in ubuntu:21.04 images and will not modify the one with the new tag. The ubuntu version with the new tag that we have created will be your own ubuntu (with all the modifications that we have done before).



The **tag** is useful because you can have ubuntu with your own tag and this can be the version that you modify (e.g. install sudo, nano..). It is also possible to change completely the name, not only changing the tag, and indeed you can have the same images with multiple names.

The problem of doing all the steps we did before is that when you close the terminal you'll lose all the information on what you have done. There is a way that allows to put in a text file all the steps that we have done so far, in order to remember them and make an easier way to build a docker. This is done using a file called **Dockerfile**. Dockerfile is a file in which you can store the information to build a docker from scratch in a text file.

Dockerfile



To generate a Dockerfile you first have to generate a new folder (better to put the folder in such a way to have a shorter path to call, usually in the desktop). Inside the folder, you have to put the Dockerfile. Dockerfile is a summary of all the infomation written above.

```
FROM ubuntu:20.04
RUN apt-get update
RUN apt-get upgrade -y
RUN apt-get install -y sudo
RUN sudo apt-get install -y nano
COPY ./command.sh /home/
RUN chmod +x /home/command.sh
```

What does the Dockerfile do? First, download the image, then FROM the ubuntu image ubuntu:20.04 that is someway in the repository, RUN (inside the ubuntu) the apt-get update, apt-get upgrade (-y → means that is there any software that has to be installed it will be installed without asking; this is done because the dockerfile is used to do things without the user intervention), then install sudo, and then install nano. Next, there is also the command COPY that copies the file command.sh in the folder home. Finally, there is RUN chmod that is used to change the properties of the file (in this case, chmod +x is done to render the file executable).

This is important because you can create scripts (with R for example), and place the scripts somewhere in the docker, which then will be runned. Dockerfile is a way to put together more than one command to build a docker container.

in this way you will put the docker file in the docker container you are creating and you will note loose

After you have created the folder with the Dockerfile, using the terminal you have to go in the parent folder of the folder in which the docker file is located. For example, if the folder with the dockerfile is on the desktop, you open the terminal and go with the `cd` command in the desktop.

There is a command that is used to create a new image from scratch.

```
docker build (name of the folder in which is the dockerfile) -t (name of the docker images that I want)
```

So basically this command uses the dockerfile that is inside the folder that was written to build a docker, and with `-t` (tagging) gives a name to the docker that is creating.

After the running is finished, you type `docker images` and you find an image with the name that you give. Then, if you go inside the docker image with `docker run` and then type `ls -l` you'll see all the elements with the long representation. If then you type command.sh the terminal gives "Hello word". This is simply because inside the command.sh file there is written echo "Hello word". The shell command `echo` is printing what is written after echo, and so it will print "Hello word". This is done using the nano terminal, that is really an old terminal, but the advantage of the nano terminal is that wherever nano is installed, it helps to manipulate in an easy way a test file. Nano is a simple editor that can be used even if you don't have a graphical interface.

Basically there are two way to build a docker:

- make all the steps seen before directly on the terminal
- create a dockerfile and then use the `docker build` command to execute the dockerfile

There is a difference between these two modalities: the first one is used to modify a docker while the second one is used to build a new docker with all the characteristics that you want. The second method has also the advanges that in the folder where there is the Dockerfile it is possible to add all the data and software that you want to put in the docker.

Exercise 3 (Homework)

Create a Dockerfile to build a docker image named r4:v.0.01. To build this docker start from rocker/r-ubuntu:20.04 which is an available docker container with a version of R. Rocker is the name of the user in the docker hub and there is a version that is called r-ubuntu:20.04 (this is an ubuntu version that contains already R20.04 inside).

What we have to do?

Build a docker filer starting from this version, then do update, upgrade, then sudo, nano. Finally, you have to try yourself to find the command to install the upam package which is a R package inside the new docker that you have built using the Dockerfile.

The latest point is that when you have the docker, the docker is isolated respect to the computer and with `-v` you will make a connection between the docker and the folder with the data. The idea is that everything is much more controlled because you do the installation ones and then every time that you need you have only to recover the image.

Solution exercise3

First, you have to create a new folder with the Docker file inside.

The Dockerfile can be created with Notes or with a text file editor.

Since the exercise require also the installation of the umap R package, you have to open R studio, create a script with the R command to install upam and save the script in the folder in which the Dockerfile is present with the name umap_command.R.

R command to install umap:

```
install.packages('umap', repos='http://cran.us.r-project.org')
```

Inside the Dockerfile:

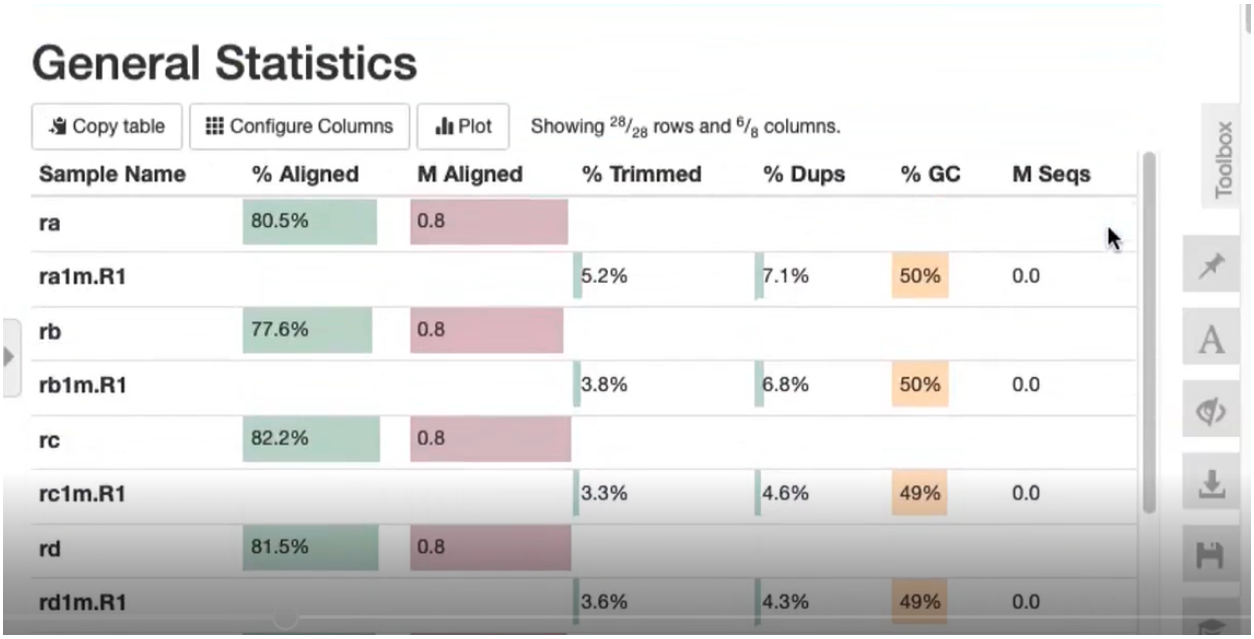
FROM rocker/r-ubuntu:20.04 #getting base image from rocker
RUN apt-get -y update #execute the update and the upgarde
RUN apt-get -y upgrade
RUN apt-get -y install sudo #install sudo
RUN apt-get -y install nano #install nano
COPY ./umap_command.R /home/ #copy the Rfile with the command to install umap packeges
RUN chmod +x /home/umap_command.R #enamble to run the umap_command in the R file redering the file executable

RUN sudo apt-get update && sudo apt-get -y install libssl-dev

After creating the Dockerfile goes on the terminal and type the command below.

```
docker build Dockerfile -t r4:v.0.01 #build the docker
docker run -i -t r4:v.0.01  #run the docker
sudo ls   #check that sudo is present and work
Rscript ./umap_command.R #run the R file to install umap
exit
docker ps -a    #find the instance name ID  of the instance previously closed
docker commit instaceID r4:v.0.01  #commit the changes to the docker
docker run -i -t r4:v.0.01   #run again the docker
sudo ls    #check that the changes were saved
exit
```

⚠️ When you perform docker build on the terminal is possible that will appear an error due to the Dockerfile. It ir really important that the Dockerfile don’t have the extension .txt but has to be consider a simple file. To change the extension you have to click on the file View or Visualizza (in the upper part of the window)—> then go in the section Show/Hide or Mostra/Nascondi—> then put the tik on file name extension and on hidden items or Estensioni nomi file and Elementi Nascosti. At this point the name of the file appear with the the .txt extension and you have to rename the Dockerfile eliminating the .txt.


General Statistics
Copy table
Configure Columns
Plot
Showing 28/28 rows and 6/8 columns.
Sample Name
% Aligned
M Aligned
% Trimmed
% Dups
% GC
M Seqs
ra
80.5%
0.8
ra1m.R1
5.2%
7.1%
50%
0.0
rb
77.6%
0.8
rb1m.R1
3.8%
6.8%
50%
0.0
rc
82.2%
0.8
rc1m.R1
3.3%
4.6%
49%
0.0
rd
81.5%
0.8
rd1m.R1
3.6%
4.3%
49%
0.0
Toolbox