Illumina metagenomic data analysis

Luc van Zon

2025-02-07

Table of contents

Introduction

Welcome to the Illumina metagenomic data analysis manual. This manual contains a step by step guide for performing quality control, filtering host sequences, assembling reads into contigs, annotating the contigs, and then extracing viral contigs and their corresponding reads. In the final chapter of the manual we will show how to automate all of these steps into a single pipeline for speed and convenience.

1 Preparation

> **Important!**
>
> In the following sections whenever a **"parameter"** in brackets `{}` is shown, the intention is to fill in your own filename or value. Each parameter will be explained in the section in detail.

> **Tip**
>
> Notice the small *"Copy to Clipboard"* button on the right hand side of each code chunk, this can be used to copy the code.

1.1 Activating the correct conda software environment

Anaconda is a software management tool that can be used for creating specific environments where bioinformatics software can be installed in Linux, as it manages all the dependencies of different softwares for you. To run the steps outlined in this manual, be sure to first activate the proper conda environment.

Activate a conda environment can be done by copying and executing the following code:

```
conda activate {environment}
```

- `{environment}` is the name of your chosen conda environment.

> **Note**
>
> We are now ready to start executing the code to perform quality control of our raw Illumina sequencing data in the next chapter.

2 Quality control

> ⚠ Important!
>
> In the next steps we are going to copy-paste code, adjust it to our needs, and execute it on the command-line.
> **Please open a plain text editor to paste the code from the next steps, to keep track of your progress!**

For simplicity's sake, most steps will be catered towards an analysis of a single sample. It is recommended to follow a basic file structure like the following below:

```
my_project/
   raw_data/          # Contains the raw, gzipped FASTQ files
      sample1_R1_001.fastq.gz
      sample1_R2_001.fastq.gz
      sample2_R1_001.fastq.gz
      sample2_R2_001.fastq.gz
   results/           # This is where the output files will be stored
   log/               # This is where log files will be stored
```

2.1 Decompressing FASTQ

The first step is to decompress the raw FASTQ files. FASTQ files are often compressed using gzip to save disk space. We'll use the zcat command to decompress them.

Modify and run:

```
zcat {input.folder}/{sample}_R1_001.fastq.gz > {output.folder}/{sample}_R1.fastq
zcat {input.folder}/{sample}_R2_001.fastq.gz > {output.folder}/{sample}_R2.fastq
```

- `{input.folder}` is where your raw .fastq.gz data is stored.
- `{sample}` is the name of your sample.
- `{output.folder}` is where your decompressed .fastq files will be stored.

2.2 Deduplicate reads

This step removes duplicate reads from the uncompressed FASTQ files. Duplicate reads can arise during PCR amplification or sequencing and can skew downstream analyses. We'll use the cd-hit-dup program to identify and remove these duplicates.

After cd-hit-dup command is finished, we'll remove the .clstr file that cd-hit-dup creates. This file contains information about the clusters of duplicate reads, but it's not needed for downstream analysis, so we can safely remove it to save disk space.

```
cd-hit-dup -u 50 -i {input.R1} -i2 {input.R2} -o {output.R1} -o2 {output.R2} > {logs}/{sample

rm {output.dir}/*.clstr
```

- `{input.R1}` and `{input.R2}` are the decompressed R1 and R2 reads from **step 2.1**.
- `{output.R1}` and `{output.R2}` are your deduplicated .fastq reads. Think of where you want to store your results, something like `results/dedup/` will be sufficient, so output.R1 turns into `result/dedup/SampleName_R1.fastq`, etc.
- `{output.dir}` This should be pointed to wherever your deduplicated reads are stored.

2.3 Running fastp quality control software

The fastp software is a very fast multipurpose quality control software to perform quality and sequence adapter trimming for Illumina short-read and Nanopore long-read data.

Run and modify:

```
        fastp -i {input.R1} -I {input.R2} \
        -o {output.R1} -O {output.R2} \
        --unpaired1 {output.S} --unpaired2 {output.S} --failed_out {output.failed} \
        --length_required 50 \
        --low_complexity_filter \
        --cut_right \
        --cut_right_window_size 5 \
        --cut_right_mean_quality 25 \
        --thread {threads} \
        -j result/{sample}/dedup_qc/qc_report.json -h result/{sample}/dedup_qc/qc_report.html
```

- `{input.R1}` and `{input.R2}` are the reads belonging to the deduplicated sample **step 2.2**.
- `{output.R1}` and `{output.R2}` are the the quality controlled .fastq filenames.

- `{output.S}` –unpaired1 and –unpaired2 tells fastp to write unpaired reads to a .fastq file. In our case, we write unpaired reads (whether they originated from the R1 or R2 file) to the same file, output.S.
- `{output.failed}` .fastq file that stores reads (either merged or unmerged) which failed the quality filters
- `{sample}` is the name of your sample.
- `{log}` is the directory for log files.

> **Note**
>
> `{threads}` is a recurring setting for the number of CPUs to use for the processing. On a laptop this will be less (e.g. 8), on an HPC you may be able to use 64 or more CPUs for processing. However, how much performance increase you get depends on the software.

2.4 Host filtering

This step removes reads that map to a host genome (e.g., human). This is important if you're studying metagenomes from a host-associated environment (e.g., gut microbiome, skin surface).

```
bwa mem -aY -t {threads} {params.reference} {input.R1} {input.R2} | \
samtools fastq -f 4 -s /dev/null -1 {output.R1} -2 {output.R2} -
bwa mem -aY -t {threads} {params.reference}  {input.S} | \
samtools fastq -f 4 - > {output.S}
```

- `{input.R1}` and `{input.R2}` are QC-filtered FASTQ files from **step 2.3**.
- `{output.R1}` and `{output.R2}` are FASTQ files (R1, R2) containing reads that did not map to the host genome.
- `{input.S}` are singleton reads from the previous step (output.S).
- `{params.reference}` is a reference genome sequence.

> **Note**
>
> We now have our quality controlled sequence reads which we can use to create an assembly in the next chapter.

3 De novo assembly

3.1 metaSPades

We will perform de novo assembly of the host-filtered reads to create contigs (longer, assembled sequences) with SPades.

```
COUNT=$(cat {input.S} | wc -l)
if [[ $COUNT -gt 0 ]]
then
    spades.py -t {threads} \
    --meta \
    -o {output} \
    -1 {input.R1} \
    -2 {input.R2} \
    -s {input.S} > {log} 2>&1
else
    spades.py -t {threads} \
    --meta \
    -o {output} \
    -1 {input.R1} \
    -2 {input.R2} > {log} 2>&1
fi
```

The `if [[ $COUNT -gt 0 ]]` condition checks if singleton reads are present. If so, we will include them with the -s option in the spades.py command

- `{input.R1}`, `{input.R2}` and `{input.S}` are host-filtered FASTQ files (R1, R2, and S) from **step 2.3**.
- `{output}` defines the directory where the assembly results will be stored.
- `{log}` specifies the log directory.

3.2 Renaming contigs:

We will add sample names to the beginning of each contig name. This will make sure that each sample's contig names are unique before we start aggregating contigs. If you are only dealing with a single sample, then this step can be seen as optional.

```
seqkit replace -p "^" -r "{sample}_" {input} > {output}
```

- `{sample}` is the sample name that will be placed in front of the contig name
- `{input}` is your contig file from **step 3.1**
- `{output}` is fasta file with the renamed contig

3.3 Aggregating contigs:

Next we will combine all renamed contigs into a single fasta file so we can perform taxonomic annotation across all samples in one go. Once again, this step can be seen as optional if you have a single sample.

```
cat {input} > {output}
```

- `{input}` are your renamed contig files from **step 3.2**
- `{output}` a single .fasta file

> **Note**
>
> We have created contigs that are ready for taxonomic annotation in the next chapter.

4 Taxonomic classification

4.1 Diamond

Now we will annotate the aggregated contigs by assigning taxonomic classifications to them based on sequence similarity to known proteins in a database using diamond blastx.

The `-f 6` parameter will ensure the output is in a tabular format. The `6` may be followed by a space-separated list of various keywords, each specifying a field of the output. For a full description of the output, please visit here.

```
diamond blastx \
--frameshift 15 \
-q {input} \
-d {db} \
-o {output} \
-f 6 qseqid sseqid pident length mismatch gapopen qstart qend sstart send evalue bits
--taxonmap {tx} \
--threads {threads} \
-b 10 -c 1 > {log} 2>&1
```

- `{input}` is the contig file created in either **step 3.3** or **step 3.1**, depending on your amount of samples.
- `{db}` is the protein database to be searched against.
- `{output}` is a .tsv file containing the annotation results.
- `{tx}` is a mapping file to convert protein accessions to taxonomic IDs.
- `{log}` is the log directory.

Multiple different databases and taxomaps can be found in: `\\cifs.research.erasmusmc.nl\viro0002\workgr`

4.2 Split annotation files

We will split the combined annotation file back into individual annotation files for each sample. This step can be seen as optional if you are dealing with a single sample.

```
mkdir -p tmp_split

sed 's/_NODE/|NODE/' {input} | awk -F'|' '{{
    identifier = $1;  # Construct the identifier using the first two fields

    output_file = "tmp_split/" identifier;  # Construct the output filename

    if (!seen[identifier]++) {{
        close(output_file);  # Close the previous file (if any)
        output = output_file;  # Update the current output file
    }}

    print $2 > output;  # Append the line to the appropriate output file
}}'

for file in tmp_split/*; do
    mkdir -p result/$(basename ${{file}})/annotation/;
    mv $file result/$(basename ${{file}})/annotation/diamond_output.tsv;
done

rmdir tmp_split
```

- `{input}` is the combined annotation file from **step 4.1**.

4.3 Parsing diamond output

Now we will process the DIAMOND output files with a custom Python script called **post_process_diamond.py**. This script will further enrich taxonomic information for each contig based on the DIAMOND alignment results. If a contig has multiple matches in the database, it will select the best hit based on a combined score of bitscore and length. Lastly, it separates the contigs into two lists: those that were successfully annotated and unannotated.

```
python /mnt/viro0002/workgroups_projects/Bioinformatics/scripts/post_process_diamond
-i {input.annotation} \
-c {input.contigs} \
-o {output.annotated} \
-u {output.unannotated} \
-log {log}
```

- `{input.annotation}` is the annotation file **step 4.2**.

- `{input.contigs}` are the contigs from the SPAdes **step 3.1**.
- `{output.annotated}` is a set of annotated contigs.
- `{output.unannotated}` is a set of unannotated contig IDs.
- `{log}` is the log directory.

> **Note**
>
> We can now move on to the final steps where we will create various files needed for downstream analysis.

5 Extracting Viral Sequences and Analyzing Mapped Reads

We will conclude the pipeline with various steps which will create valuable data files for further downstream analysis.

5.1 Extract viral annotations

A basic grep command can be used to extract contigs that have been annotated as viral from the annotation file in **step 4.3**.

```
grep "Viruses$" {input.annotated} > {output.viral} || touch {output.viral}
```

- `{input.annotated}` is the annotation file from **step 4.3**.
- `{output.viral}` contains all of your viral contigs.

5.2 Mapping reads to contigs

We can map the quality-filtered and host-filtered reads back to the assembled contigs to quantify the abundance of different contigs in each sample. We will create a mapping file for paired reads (R1 and R2) and singletons (S) and then merge these two files together.

```
bwa index {input.contigs}
bwa mem -Y -t {threads} {input.contigs} {input.R1} {input.R2} | samtools sort - > {output.pai
bwa mem -Y -t {threads} {input.contigs} {input.S} | samtools sort - > {output.S}/tmp_singlets
samtools merge {output.merged}/contigs.bam {output.paired}/tmp_paired.bam {output.S}/tmp_sing
rm {output.paired}/tmp_paired.bam {output.S}/tmp_singlets.bam
```

- `{input.contigs}` contains the assembled contigs from **step 3.1**.
- `{input.R1}`, `{input.R2}` and `{input.S}` are the quality controlled and host-filtered reads from **step 2.4**.
- `{output.paired}` is the directory for the .bam mapping file based on paired reads.
- `{output.S}` is the directory for the .bam mapping file based on singleton reads.
- `{output.merged}` is the directory for the merged .bam file.

5.3 Extract mapped reads

We will extract the reads for each annotation file that we've created.

```
#Create temporary BED file to extract the annotated mappings from the BAM of all mappings
cut -f1 {input.annotated} | awk -F'_' '{{print $0 "\t" 0 "\t" $4}}' > result/{sample}/mapping
samtools view -bL result/{sample}/mapping/tmp.bed {input.mapped} > {output.annotated}

#Do the same for the unannotated contigs
cut -f1 {input.unannotated} | awk -F'_' '{{print $0 "\t" 0 "\t" $4}}' > result/{sample}/mappi
samtools view -bL result/{sample}/mapping/tmp.bed {input.mapped} > {output.unannotated}

#Do the same for the viral contigs
cut -f1 {input.viral} | awk -F'_' '{{print $0 "\t" 0 "\t" $4}}' > result/{sample}/mapping/tmp
samtools view -bL result/{sample}/mapping/tmp.bed {input.mapped} > {output.viral}

rm result/{sample}/mapping/tmp.bed
```

- `{input.annotated}`, `{input.unannotated}` and `{input.viral}` are the .tsv annotation files from **step 4.3 and 5.1**.
- `{input.mapped}` is the combined .bam file from **step 5.2**.
- `{output.annotated}`, `{output.unannotated}` and `{output.viral}` are .bam files for each annotation input.
- `{sample}` is the name of your sample.

5.4 Count mapped reads

Next we count the number of reads that mapped to each contig in the annotated, unannotated, and viral BAM files.

```
samtools view -bF2052 {input.annotated} | seqkit bam -Qc - | awk '$2 != 0 {{print}}' > {outpu
samtools view -bF2052 {input.unannotated} | seqkit bam -Qc - | awk '$2 != 0 {{print}}' > {ou
samtools view -bF2052 {input.viral} | seqkit bam -Qc - | awk '$2 != 0 {{print}}' > {output.vi
```

- `{input.annotated}`, `{input.unannotated}` and `{input.viral}` are the .bam output files from **step 5.3**.
- `{output.annotated}`, `{output.unannotated}` and `{output.viral}` are .tsv files containing the read counts for each .bam file

5.5 Extract contigs

Lastly, we will extract contigs for the annotated, unannotated and viral contigs.

```
seqkit grep -f <(cut -f1 {input.annotated}) {input.contigs} > {output.annotated}
seqkit grep -f <(cut -f1 {input.unannotated}) {input.contigs} > {output.unannotated}
seqkit grep -f <(cut -f1 {input.viral}) {input.contigs} > {output.viral}
```

- `{input.annotated}`, `{input.unannotated}` and `{input.viral}` are .tsv annotation files from **steps 4.3 and 5.1**.
- `{input.contigs}` is the .fasta file containing all contigs for a sample from **step 3.1**.
- `{output.annotated}`, `{output.unannotated}` and `{output.viral}` are .fasta files containing the contigs.

> **Note**
>
> You can now move to the final chapter to automate all of the steps we've previously discussed.

6 Automating data analysis

6.1 Preparing to run the workflow.

We can make use of a tool called Snakemake to automate the previous steps into a single pipeline. To set this up, you must create a tabular config file, which contains …

The tabular config file has the following structure:

[insert tabular example file]

Next, we need to create a place for your results to be stored. First check your current directory with the `pwd` command:

```
pwd
```

Change your current directory using `cd` if needed:

```
cd /{folder1}/{folder2}
```

Then create the new directory using the `mkdir` command:

```
mkdir results
```

This creates a new directory at the current location. Move the `sample_config.tsv` file to the results directory using the `mv` (move) command:

```
mv sample_config.tsv results
```

> **Tip**
>
> If you get a "file not found" error after changing your directory (`cd`) you may need to write the complete path of your sample_config.tsv file e.g `mv /home/username/Documents/sample_config.tsv results`

After preparing the results directory we have to make sure that we have activated the appropriate conda environment and check if the command line has `(your_environment_name)` in front.

Next, run the `ls` command to list the files in the current directory and check if the `InsertCorrectName.smk` file is present. This is the "recipe" for the workflow, describing all the steps we have done by hand. (you can open the `.smk` file with a text editor and have a look).

6.2 Running the workflow

After setting everything up, we can redo the analysis for all samples in a single step. First we will test out a dry run to see if any errors appear. A dry run will not execute any of the commands but will instead display what would be done. This will help identify any errors in the Snakemake or config file.

```
snakemake \
--snakefile \
InsertCorrectName.smk \
--directory {ourdir} \
--configfile sample_config={config} \
--cores {threads}
--dryrun
```

- `{ourdir}` is the directly we created.
- `{config}` is the tabular config file.

If no errors appear, then remove the `--dryrun` argument and run it again to fully execute the workflow.