Tutorial for metabarcode analysis

Benjamin Alric

2021-05-11

Contents

1 Aims

In this repository, you will find the tutotrial explaining how to process metabarcoding data with the `dada2` suite as implementated in R and generate tables of amplicon sequence variants (i.e., ASVs). This tutorial is adapted from the `dada2` tutorial.

2 Overview of `dada2` pipeline

The starting point of the `dada2` pipeline is a set of demultiplexed fastq files corresponding to the samples in our amplicon sequencing study. That is, `dada2` expects there to be an individual fastq file for each sample (or two fastq files, one forward and one reverse for each sample). Demultiplexing is often performed at the sequencing center, but if that has not been done there are a variety of tools to accomplish this.Once demultiplexed fastq files without non-biological nucleotides are in hand, the `dada2` pipeline proceeds in eight steps as follow:

- Step 1: Inspect read quality profiles
- Step 2: Filtering and trimming
- Step 3: Learn error rates
- Step 4: Dereplicate
- Step 5: Infer sample composition
- Step 6: Merge paired reads
- Step 7: Make sequence table
- Step 8: Remove chimeras

The output of pipeline is a table of ASVs with rows corresponding to samples and columns to amplicon.In this matrix, the value of each entry is the number of times ASV was observed in that sample. This table is

analogous to the traditional OTU table, except at higher resolution, i.e. exact amplicon sequence variants rather than (usually 97%) clusters of sequencing reads.

NOTE: Each step of our analysis pipeline is composed in two parts, a first part (in bash) to launch the second part (in R) which is the core part of the analysis. Each step should be made separately.

---

3 Step 1: Inspect read quality profiles

It is important to get a feel for the quality of the data that we are using. To do this, we will plot the quality of samples. These plots allow us to define where the quality of the sequences falls.

NOTE: Plots summarizing the quality of the reads are generated at sample scale.

```
#!/bin/bash
#SBATCH --job-name=dada2_01_quality_plot.qsub               # job name
#SBATCH --mail-type=BEGIN,END                               # send a mail at the begining/end of job
#SBATCH --mail-user=balric@sb-roscoff.fr                    # where to send mail
#SBATCH --cpus-per-task 1                                   # number of CPUs required per task
#SBATCH --mem 4GB                                           # memory per processor
#SBATCH --workdir=/shared/projects/indigene/PHYTOPORT/18SV4 # set working directory
#SBATCH -p fast                                             # partition
#SBATCH -o dada2_01_quality_plot.out                        # output file
#SBATCH -e dada2_01_quality_plot.err                        # error file

module load r/4.0.3

srun Rscript /shared/projects/indigene/PHYTOPORT/18SV4/script/dada2_01_quality_plot.R
```

```
# ------------------------------------------------------------------ #
# This section corresponds to the Rscript dada2_01_quality_plot.R #
# ------------------------------------------------------------------ #


# Some packages must be installed and loaded
library(dada2)
library(magrittr)
library(ggplot2)


# Set up pathway to forward and reverse fastq files
pathF <- "/shared/projects/indigene/PHYTOPORT/18SV4/FWD"
pathR <- "/shared/projects/indigene/PHYTOPORT/18SV4/REV"

# Create folders for quality profiles
dir.create(paste0(pathF, "/qualityplot"))
dir.create(paste0(pathR, "/qualityplot"))
```

```r
# Set file paths where quality plots will be stored
filtpathF <- file.path(pathF, "qualityplot")
filtpathR <- file.path(pathR, "qualityplot")

# Get a list of all fastq files in the work directory and separate FWD and REV
# Forward and reverse fastq files have format:
# SAMPLENAME_Cut1_trimmed.fastq.gz and SAMPLENAME_Cut2_trimmed.fastq.gz
fastqFs <- sort(list.files(pathF, pattern = "fastq.gz"))
fastqRs <- sort(list.files(pathR, pattern = "fastq.gz"))

# Select file with a size above to 1000 bytes
fastqFs <- fastqFs[file.size(file.path(pathF, fastqFs)) > 1000]
fastqRs <- fastqRs[file.size(file.path(pathR, fastqRs)) > 1000]
if (length(fastqFs) != length(fastqRs)) {
  stop("Forward and reverse files do not match.")
}

# Identify the name of the run (here one run: PHYTOPORT)
runs <- sub("^PHP_[B, F, H]\\d+_\\d+_\\d+_([^_]+).+$", "\\1", fastqFs) %>% unique

# Quality plots to the sample level
pdf(file.path(filtpathF, "indiv_F_Qplots.pdf"))
for (i in file.path(pathF, fastqFs)[1]) {
  print(plotQualityProfile(i))
}
dev.off()

pdf(file.path(filtpathR, "indiv_R_Qplots.pdf"))
for (i in file.path(pathR, fastqRs)) {
  print(plotQualityProfile(i))
}
dev.off()

# Version of packages used to build this document
sessionInfo()
```

Visualization example of the quality plot

Description of quality plots: In gray-scale is a heat map of the frequency of each quality score at each base position. The median quality score at each position id shown by the green line, and the quartiles of the quality score distribution by the orange lines. Th ered line shown the scaled proportion of reads that extend to at least that position (this is more useful for other sequecing technologies, as Illumina reads are typically all the same length, hence the flat red line).

---

4 Step 2: Filtering and trimming

In sequence data, low-quality sequences can contain unexpected and misleading errors, and Illumina sequencing quality tends to drop off at the end of reads. Therefore, before chosing sequence variants, we trim reads where their quality scores begin to drop (the `truncLen` and `truncQ` values) and remove any low-quality reads that are left over after we have finished trimming (the `maxEE` value).

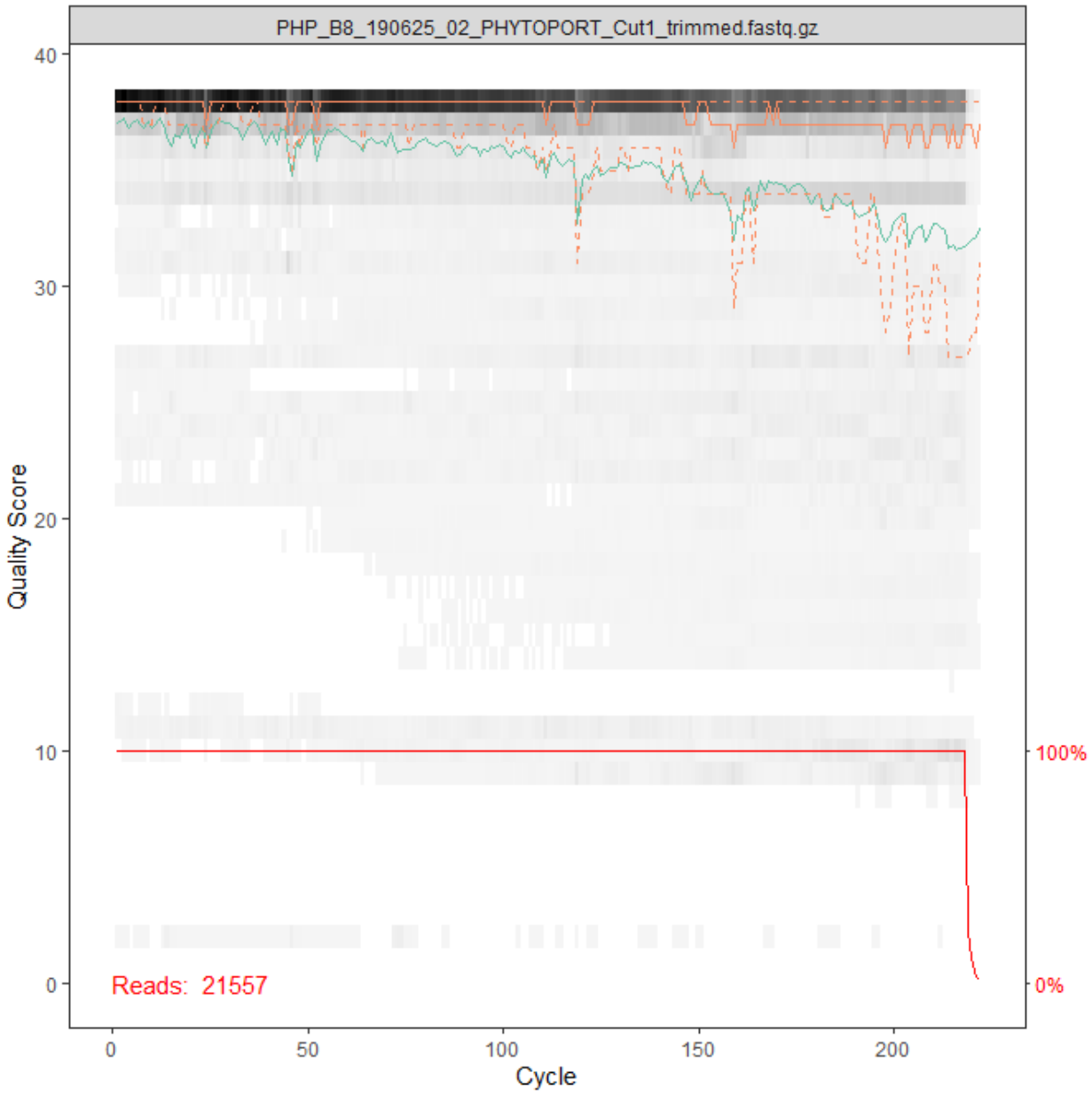

Figure 1: Visualization example of the quality plot

In `truncLen=c(xxx, yyy)` of the function `filterAndTrim()`, `xxx` refers to the forward read truncation length, whereas `yyy` refers to the reverse read truncation length. These parameters are determined from the inspection of quality profiles.

These parameters should be stored in a .csv file (*dada2_pipeline_parameters.csv*) located in your workspace.

WARNING: THESE PARAMETERS ARE NOT OPTIMAL FOR ALL DATASETS. Make sure you determine the trim and filtering parameters for your data. The settings used here are appropriates for MiSeq runs that are 2x250 bp.

```
#!/bin/bash
#SBATCH --job-name=dad2_02_filter_trim_a.qsub               # job name
#SBATCH --mail-type=BEGIN,END                               # send a mail at the begining/end of job
#SBATCH --mail-user=balric@sb-roscoff.fr                    # where to send mail
#SBATCH --cpus-per-task 4                                   # number of CPUs required per task
#SBATCH --mem 10GB                                          # memory per processor
#SBATCH --workdir=/shared/projects/indigene/PHYTOPORT/18SV4 # set working directory
#SBATCH -p fast                                             # partition
#SBATCH -o dada2_02_filter_trim_a.out                       # output file
#SBATCH -e dada2_02_filter_trim_a.err                       # error file

module load r/4.0.3

srun Rscript /shared/projects/indigene/PHYTOPORT/18SV4/script/dada2_02_filter_trim_a.R
```

```
# ------------------------------------------------------------------ #
# This section corresponds to the Rscript dada2_02_filter_trim_a.R #
# ------------------------------------------------------------------ #


# Some packages must be installed and loaded
library(dada2)
library(data.table)
library(magrittr)


# # Set up pathway to forward and reverse fastq files
path <- "/shared/projects/indigene/PHYTOPORT/18SV4"
pathF <- "/shared/projects/indigene/PHYTOPORT/18SV4/FWD"
pathR <- "/shared/projects/indigene/PHYTOPORT/18SV4/REV"

# Create folders for dada2 results
dir.create((paste0(path, "/dada2/log")))

# Load the table of dada2 parameters
dada2_param <- read.csv2(paste0(path, "/dada2_pipeline_parameters.csv"), header = TRUE, stringsAsFactors

# Get a list of all fastq files in the work directory and separate FWD and REV
# Forward and reverse fastq files have format:
# SAMPLENAME_Cut1_trimmed.fastq.gz and SAMPLENAME_Cut1_trimmed.fastq.gz
fastqFs <- sort(list.files(pathF, pattern = "fastq.gz"))
```

```
fastqRs <- sort(list.files(pathR, pattern = "fastq.gz"))

# Select file with a size above to 1000 bytes
fastqFs <- fastqFs[file.size(file.path(pathF, fastqFs)) > 1000]
fastqRs <- fastqRs[file.size(file.path(pathR, fastqRs)) > 1000]
if (length(fastqFs) != length(fastqRs)) {
  stop("Forward and reverse files do not match.")
}

# Arguments for filter and trim
args <- dada2_param[dada2_param$step == "step01", ]
trunc_fwd <- args[args$vairable == "TRUNC_FWD", 3] %>% as.numeric
trunc_rev <- args[args$vairable == "TRUNC_REV", 3] %>% as.numeric
maxee <- args[args$vairable == "MAXEE", 3] %>% as.numeric

# Set file paths where files will be stored
pathF <- "/shared/projects/indigene/PHYTOPORT/18SV4/FWD"
pathR <- "/shared/projects/indigene/PHYTOPORT/18SV4/REV"
filtpathF <- file.path(pathF, "filtered")
filtpathR <- file.path(pathR, "filtered")

# Filtering and trimming
for (i in fastqFs) {
  dada2::filterAndTrim(fwd = file.path(pathF, i), filt = file.path(filtpathF, i),
                                    rev = file.path(pathR, i), filt.rev = file.path(filtpathR, i),
                       truncLen = c(trunc_fwd, trunc_rev),
                       maxEE = maxee, maxN = 0, compress = TRUE, verbose = TRUE, multithread = FALSE) ->

  # Save filtered and trimmed table in the folder of dada2 results
  data.table(out, keep.rownames = TRUE) %>%
    setnames(names(data.table(out, keep.rownames = TRUE))[1], c("sample")) %>%
    fwrite(paste0(path, "/dada2/log/filter_", sub("_trimmed.+$","",basename(i)), ".csv"), sep = ";", co
rm(out)
}

# Version of packages used to build this document
sessionInfo()
```

NOTE: The script dada2_02_filter_trim_a.R produces for each sample one file .csv. Here we merge all files in one table with rows as samples and three columns correspoding to sample names, reads.in, reads.out.

```
#!/bin/bash
#SBATCH --job-name=dada2_02_filter_trim_b.qsub              # job name
#SBATCH --mail-type=BEGIN,END                               # send a mail at the begining/end of job
#SBATCH --mail-user=balric@sb-roscoff.fr                    # where to send mail
#SBATCH --cpus-per-task 1                                   # number of CPUs required per task
#SBATCH --mem 4GB                                           # memory per processor
#SBATCH --workdir=/shared/projects/indigene/PHYTOPORT/18SV4 # set working directory
#SBATCH -p fast                                             # partition
```

```
#SBATCH -o dada2_02_filter_trim_b.out                          # output file
#SBATCH -e dada2_02_filter_trim_b.err                          # error file

module load r/4.0.3

srun Rscript /shared/projects/indigene/PHYTOPORT/18SV4/script/dada2_02_filter_trim_b.R
```

```
# ------------------------------------------------------------------ #
# This section corresponds to the Rscript dada2_02_filter_trim_b.R #
# ------------------------------------------------------------------ #


# Some packages must be installed and loaded
library(data.table)
library(magrittr)


# Merge all files in one table
path <- "/shared/projects/indigene/PHYTOPORT/18SV4/dada2/log"
myfiles <- list.files(path = path, pattern = "*.csv", full.names = TRUE)
filter <- lapply(myfiles, read.csv2)
filter <- do.call('rbind', filter)
filter %>%
  data.table() %>%
  fwrite(paste0(path, "/filter.csv"), sep = ";", col.names = TRUE)

# Version of packages used to build this document
sessionInfo()
```

5 Step 3: Learn error rates

Errors can be introduced by PCR amplification and sequencing. In this part of the pipeline `dada2` will learn to distinguish error from biological differences using a subset of our data as a training set. The `learnErrors()` method learns this error model from the data, by alternating estimation of the error rates and inference of sample composition until they converge on a jointly consistent solution. he error parameters typically vary between sequencing runs and PCR protocols, so this method provides a way to estimate those parameters from the data itself.

```
#!/bin/bash
#SBATCH --job-name=dada2_03_learn_error.qsub                # job name
#SBATCH --mail-type=BEGIN,END                               # send a mail at the begining/end of job
#SBATCH --mail-user=balric@sb-roscoff.fr                    # where to send mail
#SBATCH --cpus-per-task 6                                   # number of CPUs required per task
#SBATCH --mem 10GB                                          # memory per processor
#SBATCH --workdir=/shared/projects/indigene/PHYTOPORT/18SV4 # set working directory
#SBATCH -p fast                                             # partition
#SBATCH -o dada2_03_learn_error.out                         # output file
#SBATCH -e dada2_03_learn_error.err                         # error file

module load r/4.0.3

srun Rscript /shared/projects/indigene/PHYTOPORT/18SV4/script/dada2_03_learn_error.R
```

```r
# ------------------------------------------------------------ #
# This section corresponds to the Rscript dada2_03_learn_error.R #
# ------------------------------------------------------------ #


# Some packages must be installed and loaded
library(dada2)
library(data.table)
library(doParallel)
library(foreach)
library(magrittr)
registerDoParallel(cores = 6)


# Load the table of dada2 parameters
path <- "/shared/projects/indigene/PHYTOPORT/18SV4"
dada2_param <- read.csv2(paste0(path, "/dada2_pipeline_parameters.csv"), header = TRUE, stringsAsFactors

# Define the following path variable so that it points to the extracted directory
pathF <- "/shared/projects/indigene/PHYTOPORT/18SV4/FWD"
pathR <- "/shared/projects/indigene/PHYTOPORT/18SV4/REV"

# Arguments for denoising and merging
args <- dada2_param[dada2_param$step == "step02", ]
min_read_nb <- args[args$vairable == "MIN_READ_NUM", 3]

# Track reads through the pipeline
# As a final check of our progress, we'll look at the number of reads that made it through each step in
getN <- function(x) sum(getUniques(x))
getA <- function(x) length(getUniques(x))

# selection based on the number of reads
tmp <- fread(paste0(path, "/dada2/log/filter.csv"))
tmpR <- tmpF <- tmp[reads.out >= min_read_nb, sample]

# Identify the name of the run (here one run: PHYTOPORT)
runs <- sub("^PHP_[B, F, H]\\d+_\\d+_\\d+_([^_]+).+$", "\\1", tmp$sample) %>% unique
runs <- c(paste(runs, "Cut1", sep = "_"), paste(runs, "Cut2", sep = "_"))

# File parsing
filtpathF <- file.path(pathF, "filtered")
filtpathR <- file.path(pathR, "filtered")
filtFsall <- paste(filtpathF, tmpF, sep = "/")
filtRsall <- paste(filtpathR, tmpR, sep = "/")
sample.names.all <- sub("_trimmed.+$", "", basename(filtFsall))
sample.namesR.all <- sub("_trimmed.+$" ,"", basename(filtRsall))
if(!identical(sample.names.all, sample.namesR.all)) {
  stop("Forward and reverse files do not match.")
}
names(filtFsall) <- sample.names.all
names(filtRsall) <- sample.names.all

# Set seed to ensure that randomized steps are replicatable
```

```
set.seed(100)

# Loop allowing to determinate learn error rates and infer sample composition
foreach(i = runs, .packages = c("data.table","dada2")) %dopar% {
  filtFs <- filtFsall[grep(i, names(filtFsall))]
  filtRs <- filtRsall[grep(i, names(filtRsall))]
  sample.names <- sub("_trimmed.+$","", basename(filtFs))
  # Learn forward error rates
  errF <- learnErrors(filtFs, nbases = 1e8, multithread = FALSE)
  pdf(paste0("dada2/log/errF_", i, ".pdf"))
  print(plotErrors(errF, nominalQ = TRUE))
  dev.off()
  # Learn reverse error rates
  errR <- learnErrors(filtRs, nbases = 1e8, multithread = FALSE)
  pdf(paste0("dada2/log/errR_", i, ".pdf"))
  print(plotErrors(errR, nominalQ = TRUE))
  dev.off()
  save(errF, file = paste0(path, "/dada2/log/errF_",i,".rda")) # CHANGE ME to where you want sequence t
  save(errR, file = paste0(path, "/dada2/log/errR_",i,".rda")) # CHANGE ME to where you want sequence t
}

# Version of packages used to build this document
sessionInfo()
```

Visualization example of the estimates error rates

NOTE: We want to make sure that the machine learning algorithm is learning the erroe rates properly. In plot of learning error rates, the red line represents what we should expect the learned error rates to look like for each of the 16 possible base transitions (A->A, A->C, A->G, etc.). Grey points are the obeserved error rates for each consensus quality score and black lines shown the error rates expected under the nominal definiton of the Q-score. The expected error rates should have a good fit with the observed rates, and the error rates must drop with increased quality. If black lines and red lines are very far off from each other, it may be a good idea to indrease the `nbases` parameter. This allows the machine leraning algorithm to train on a larger portion of your data and may help improve the fit.

---

6 Steps 4-7: Denoising, and merging

After it understands the error rates, a dereplication step is required to condense the data by collapsing together all reads that encode the same sequence, which significantly reduces later computation times. Then, using the dereplicated data and error rates, `dada2` will infer the ASVs in our data.

NOTE: In `dada` algorithm, for each run, the samples are pooled together with the argument `pool` = TRUE, which can increase the sensitivity to rare variants.

Finally, we will merge the corresponding forqard and reverse reads to create a list of the fully denoised sequences and create a sequence table for the result.

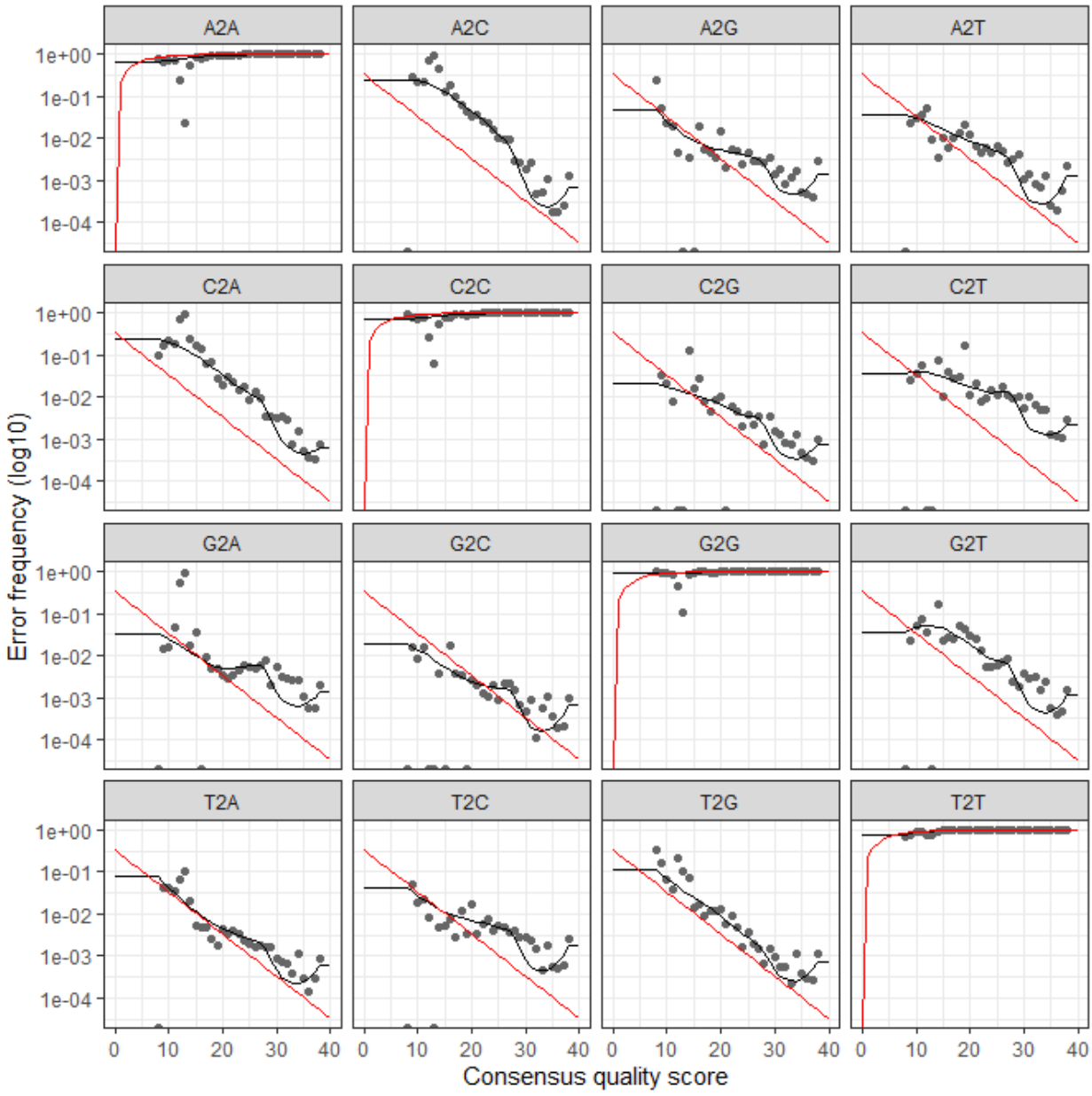

Figure 2: Visualization example of the estimates error rates

NOTE: Merging is performed by aligning the denoised forward reads with the reverse-compelment of the corresponding denoised reverse reads, and then constructing the merged "contig" sequences. By default, merged sequences are only output if the forward and reverse reads overlap by at least 12 bases, and are identical to each other in the overlap region.

```
#!/bin/bash
#SBATCH --job-name=dada2_04_denoising_merging.qsub          # job name
#SBATCH --mail-type=BEGIN,END                               # send a mail at the begining/end of job
#SBATCH --mail-user=balric@sb-roscoff.fr                    # where to send mail
#SBATCH --cpus-per-task 6                                   # number of CPUs required per task
#SBATCH --mem 10GB                                          # memory per processor
#SBATCH --workdir=/shared/projects/indigene/PHYTOPORT/18SV4 # set working directory
#SBATCH -p fast                                             # partition
#SBATCH -o dada2_04_denoising_merging.out                   # output file
#SBATCH -e dada2_04_denoising_merging.err                   # error file

module load r/4.0.3

srun Rscript /shared/projects/indigene/PHYTOPORT/18SV4/script/dada2_04_denoising_merging.R
```

```
# ------------------------------------------------------------------------ #
# This section corresponds to the Rscript dada2_04_denoising_merging.R #
# ------------------------------------------------------------------------ #


# Some packages must be installed and loaded
library(dada2)
library(data.table)
library(doParallel)
library(foreach)
library(magrittr)
registerDoParallel(cores = 6)


# Load the table of dada2 parameters
path <- "/shared/projects/indigene/PHYTOPORT/18SV4"
dada2_param <- read.csv2(paste0(path, "/dada2_pipeline_parameters.csv"), header = TRUE, stringsAsFactors

# Define the following path variable so that it points to the extracted directory
pathF <- "/shared/projects/indigene/PHYTOPORT/18SV4/FWD"
pathR <- "/shared/projects/indigene/PHYTOPORT/18SV4/REV"
pathErr <- "/shared/projects/indigene/PHYTOPORT/18SV4/dada2/log"

# Arguments for denoising and merging
args <- dada2_param[dada2_param$step == "step02", ]
min_read_nb <- args[args$vairable == "MIN_READ_NUM", 3]

# Track reads through the pipeline
# As a final check of our progress, we'll look at the number of reads that made it through each step in
getN <- function(x) sum(getUniques(x))
getA <- function(x) length(getUniques(x))
```

```
# selection based on the number of reads
tmp <- fread(paste0(path, "/dada2/log/filter.csv"))
tmpR <- tmpF <- tmp[reads.out >= min_read_nb, sample]

# Identify the name of the run (here one run: PHYTOPORT)
runs <- sub("^PHP_[B, F]\\d+_\\d+_\\d+_([^_]+).+$", "\\1", tmp$sample) %>% unique
runs <- c(paste(runs, "Cut1", sep = "_"), paste(runs, "Cut2", sep = "_"))

# File parsing
filtpathF <- file.path(pathF, "filtered")
filtpathR <- file.path(pathR, "filtered")
filtFsall <- paste(filtpathF, tmpF, sep = "/")
filtRsall <- paste(filtpathR, tmpR, sep = "/")
sample.names.all <- sub("_trimmed.+$", "", basename(filtFsall))
sample.namesR.all <- sub("_trimmed.+$" ,"", basename(filtRsall))
if(!identical(sample.names.all, sample.namesR.all)) {
  stop("Forward and reverse files do not match.")
}
names(filtFsall) <- sample.names.all
names(filtRsall) <- sample.names.all
filtErr <- sort(list.files(pathErr, pattern = ".rda"))

# Set seed to ensure that randomized steps are replicatable
set.seed(100)

# Loop allowing to determinate learn error rates and infer sample composition
foreach(i = runs, .packages = c("data.table","dada2")) %dopar% {
  filtFs <- filtFsall[grep(i, names(filtFsall))]
  filtRs <- filtRsall[grep(i, names(filtRsall))]
  sample.names <- sub("_trimmed.+$","", basename(filtFs))
  # Select the FWD and REV error files
  err <- filtErr[grep(i, filtErr)]
  load(paste(pathErr, err[grep("errF", err)], sep = "/"))
  load(paste(pathErr, err[grep("errR", err)], sep = "/"))
  # Sample inference and merger of paired-end reads
  mergers <- vector("list", length(sample.names))
  names(mergers) <- sample.names
  track <- vector("list", length(sample.names))
  names(track) <- sample.names
  for(sam in sample.names) {
    cat("Processing:", sam, "\n")
    # Dereplication of forward reads+
    derepF <- derepFastq(filtFs[[sam]])
    # Infer sample composition of forward reads
    ddF <- dada(derepF, err = errF, multithread = FALSE, pool = TRUE)
    # Dereplication of revers reads
    derepR <- derepFastq(filtRs[[sam]])
    # Infer sample composition of reverse reads
    ddR <- dada(derepR, err = errR, multithread = FALSE, pool = TRUE)
    # Merge forward/reverse reads
    merger <- mergePairs(ddF, derepF, ddR, derepR, trimOverhang = TRUE, verbose = TRUE)
    mergers[[sam]] <- merger
    track[[sam]] <- data.table(sample = sam,
```

```
                                denoisedF.read = getN(ddF), denoisedR.read = getN(ddR), merged.read = ge
                                denoisedF.seq = getA(ddF), denoisedR.seq = getA(ddR), merged.seq = getA(
  }
  rm(derepF); rm(derepR)
  # Construct sequence table
  track <- rbindlist(track)
  seqtab <- makeSequenceTable(mergers)
  saveRDS(track, paste0(path, "/dada2/log/track_",i,".rds")) # CHANGE ME to where you want sequence tab
  saveRDS(seqtab, paste0(path, "/dada2/seqtab_",i,".rds")) # CHANGE ME to where you want sequence table
  rm(track)
  rm(seqtab)
  for(i in 1:length(mergers)) {
    mergers[[i]] <- data.frame(samples = names(mergers)[i], mergers[[i]])
  }
  mergers <- rbindlist(mergers)
  saveRDS(mergers, paste0(path, "/dada2/log/mergers_",i,".rds")) # CHANGE ME to where you want sequence
}

# Version of packages used to build this document
sessionInfo()
```

7 Step 8: Remove Chimeras

Although `dada2` has searched for indel errors and substitutions, there may still be chimeric sequences in our dataset that are another important source of spurious sequences in amplicon sequencing. Chimeras are sequences that are derived form forward and reverse sequences from two different organisms becoming fused together during PCR and/or sequencing. To identify chimeras, we search for rare sequences variants that can be reconstructed by combining left-hand and right-hand segments from two more abudant "parent" sequences.

NOTE: The chimera detection method has to be `pooled` instead of `consensus`.

```
#!/bin/bash
#SBATCH --job-name=dada2_05_chimeras.qsub                    # job name
#SBATCH --mail-type=BEGIN,END                                 # send a mail at the begining/end of job
#SBATCH --mail-user=balric@sb-roscoff.fr                     # where to send mail
#SBATCH --cpus-per-task 4                                    # number of CPUs required per task
#SBATCH --mem 100GB                                          # memory per processor
#SBATCH --workdir=/shared/projects/indigene/PHYTOPORT/18SV4 # set working directory
#SBATCH -p fast                                              # partition
#SBATCH -o dada2_05_chimeras.out                             # output file
#SBATCH -e dada2_05_chimeras.err                             # error file

module load r/4.0.3

srun Rscript /shared/projects/indigene/PHYTOPORT/18SV4/script/dada2_05_chimeras.R
```

```
# ------------------------------------------------------------ #
# This section corresponds to the Rscript dada2_05_chimeras.R #
# -------------------------------------------- ------------ #
```

```r
# Some packages must be installed and loaded
library(dada2); packageVersion("dada2")
library(data.table)
library(magrittr)
library(digest)


# Define the following path variable so that it points to the extracted directory
pathFilt <- "/shared/projects/indigene/PHYTOPORT/18SV4/dada2/log"

# Merge multiple runs (if necessary)
x <- grep("seqtab_.+\\.rds$", dir("/shared/projects/indigene/PHYTOPORT/18SV4/dada2/"), value = TRUE)
x <- paste0("/shared/projects/indigene/PHYTOPORT/18SV4/dada2/", x)
st.all <- mergeSequenceTables(tables = x)

# Remove chimeras
seqtab.nochim <- removeBimeraDenovo(st.all, method = "pooled", multithread = TRUE)
seqtab.nochim2 <- seqtab.nochim %>% t %>% data.table

# load statistics
x <- grep("track_.+\\.rds$",dir("/shared/projects/indigene/PHYTOPORT/18SV4/dada2/log/"), value = TRUE)
x <- paste0("/shared/projects/indigene/PHYTOPORT/18SV4/dada2/log/", x)

stattab <- lapply(x, readRDS) %>%
  rbindlist

stattab[, nochim.read := sapply(sample, function(X){
  sum(seqtab.nochim2[, get(X)])
})]

stattab[, nochim.seq := sapply(sample, function(X){
  sum(seqtab.nochim2[, get(X)] != 0)
})]

stattab <- stattab[, list(sample, denoisedF.read, denoisedR.read, merged.read,
                          nochim.read, denoisedF.seq, denoisedR.seq, merged.seq, nochim.seq)]

filter <- read.csv2(paste0(pathFilt, "/filter.csv"), header = TRUE, stringsAsFactors = FALSE)
filter$sample <- sub("_trimmed.+$","", filter$sample)
stattab <- merge(stattab, filter, by.x = "sample", by.y = "sample")
stattab <- stattab[, c("sample", "reads.in", "reads.out",
                       "denoisedF.read", "denoisedR.read", "merged.read", "nochim.read",
                       "denoisedF.seq", "denoisedR.seq", "merged.seq", "nochim.seq")]
fwrite(stattab, "/shared/projects/indigene/PHYTOPORT/18SV4/dada2/log/statistics.tsv", sep = "\t")

rm(stattab, seqtab.nochim2)

# Construct sequence table
tmp <- seqtab.nochim %>% t %>% data.table(keep.rownames = TRUE)
setnames(tmp, "rn", "sequence")
rm(seqtab.nochim)
tmp <- data.table(amplicon = sapply(tmp[, sequence], digest, algo = "sha1"), tmp)
fwrite(tmp, "/shared/projects/indigene/PHYTOPORT/18SV4/dada2/seqtab_all.tsv", sep = "\t")
```

```r
# Version of packages used to build this document
sessionInfo()
```

At this stage, you have on ASV table which is the final product of the `dada2` pipeline. In this ASV table each row corresponds to a processes sample, and each column corresponds to an non-chimeric sample sequence (a more precise analogue to the common "OTU table"). But, the sequences variants are not yet annoted, i.e. assigning a taxonomy to the sequence variants. The `dada2` package provides a native implementation of the naive Bayesian classifer method for this purpose.