UNDER CONSTRUCTION

Philip Leftwich

2021-04-20

Contents

1 Introduction **5**

2 Unix **7**
2.1 What is Unix/Linux . . . 7
2.2 Why Learn Unix? . . . 8
2.3 Getting started . . . 8
2.4 1.4 A few foundational rules . . . 8
2.5 1.5 Let's get started . . . 9
2.6 1.6 Unix File Structure . . . 11
2.7 1.7 Absolute vs relative file paths . . . 11
2.8 1.8 Moving around . . . 12
2.9 1.9 Summary . . . 13
2.10 1.10 Stretch yourself . . . 14

3 NGS sequence analysis **17**
3.1 Background . . . 17
3.2 The data . . . 17
3.3 Removing Primers . . . 18
3.4 DADA2 - switch to R console . . . 20
3.5 Quality trimming/filtering . . . 21
3.6 Dereplication . . . 22
3.7 ASV's . . . 23
3.8 Merging reads . . . 23
3.9 Count table . . . 23
3.10 Overview . . . 24
3.11 Assign taxonomy . . . 24
3.12 Standard goods . . . 25
3.13 Contaminants . . . 26

4 Analysis in R **27**

5 Applications **29**
5.1 Example one . . . 29

5.2 Example two 29

6 Final Words 31

Chapter 1

Introduction

This is a *sample* book written in **Markdown**. You can use anything that Pandoc's Markdown supports, e.g., a math equation $a^2 + b^2 = c^2$.

The **bookdown** package can be installed from CRAN or Github:

```r
install.packages("bookdown")
# or the development version
# devtools::install_github("rstudio/bookdown")
```

Remember each Rmd file contains one and only one chapter, and a chapter is defined by the first-level heading `#`.

To compile this example to PDF, you need XeLaTeX. You are recommended to install TinyTeX (which includes XeLaTeX): https://yihui.name/tinytex/.

Chapter 2

Unix

Unix is very likely the most foundational skillset we can develop for bioinformatics (and much more than bioinformatics). Many of the most common and powerful bioinformatics approaches happen in this text-based environment, and having a solid foundation here can make everything we're trying to learn and do much easier. This is a set of 5 introductory tutorials to help us get from being completely new to Unix up to being great friends with it

2.1 What is Unix/Linux

2.1.1 Some terms

Here are some terms worth knowing, don't worry about memorising them, it can just be useful to have these to refer to in the future.

Term	What it is
shell	what we use to talk to the computer; anything where you are pointing and clicking with a mouse
command line	a text-based environment capable of taking input and providing output
Unix	a family of operating systems (we also use the term "Unix-like" because one of the most popular
bash	the most common programming language used at a Unix command-line

You should be very familiar with using a GUI (RStudio), but remember we have spent a lot of time working with files and directories using the the command line (CLI) in R. This is useful practice, because most supercomputers lack a GUI, you must work entirely using the command line.

2.2 Why Learn Unix?

Most sequencing data files are large, and require a lot of RAM to process. As a result most of the work Bioinformaticians do is not hosted on their own computers, instead they "remote-connect" to high performance supercomputers or cluster computers.

The Linux Operating System is highly flexible, free, open-source (like R) and uses very little RAM to run (Unlike Windows OS) - as such you find most supercomputers run on Linux.

You already have some experience with using a Linux OS - every time you log into RStudio Cloud you are connecting to a supercomputer that runs on Linux. Normally we do not interact directly with the OS, instead we use R and RStudio directly.

But when you click on the RStudio Terminal it provides direct access to a command-line where we can execute commands and functions directly in Linux.

This allows us to start using programs other than R, and potentially use multiple programs & programming languages to work together.

> **Note**
>
> This series of practicals is designed for you to have a first introduction to Bioinformatics, it's about exposure, not memorising or mastering anything. Don't worry about the details!

2.3 Getting started

Before we get started we need a terminal to work in.

- Open the Bioinformatics RStudio Cloud Project in the 5023Y workspace
- Click on the Terminal tab next to console in the bottom-left pane of the RStudio GUI

```r
par(mar = c(4, 4, .1, .1))
plot(pressure, type = 'b', pch = 19)
```

- This is our "command line" where we will be typing all of our commands.
- If you need to, you can exit the Terminal and start a new session easily with options in RStudio

2.4 1.4 A few foundational rules

- Just like in R spaces are special, spaces break things apart, as a rule it is therefore better to have functions and file names with dashes (-) or

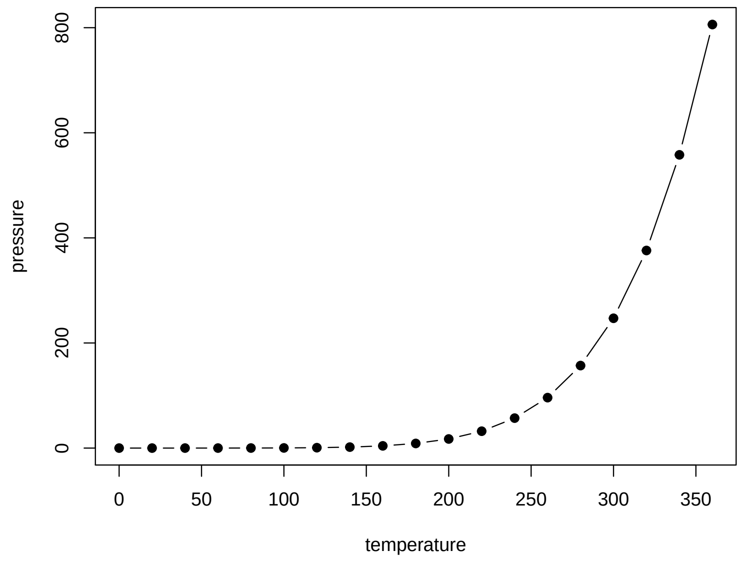

Figure 2.1: Here is a nice figure!

underscores (_) - e.g. "draft_v3.txt" is preferred to "draft v3.txt".

- The general syntax on the command line is: `command argument`. Again this is very similar to R except we don't use brackets e.g. R we are used to `command(argument)`
- Arguments can be **optional** e.g. if their is a default argument you may not have to write anything. Some functions *require* that arguments are specified. Again this is just like R.

2.5 1.5 Let's get started

We will perform a very simple function and get a flavour of the similarities and differences to working in '`R`. date' is a command that prints out the date and time.

```
date
```

https://www.geeksforgeeks.org/date-command-linux-examples/#:~:text=date%20command%20is%20used%20to,lin

```
TZ=Europe/London date

TZ=Europe/London date
```

```
date --date="next tue"
```

2.5.1 Downloading data

- curl is a command line tool for transferring data to and from the server here we will use this to download data from an online repository.
- tar will *unpack* the data from a compressed file format
- cd change the directory so we *land* in the new folder we have made

```
curl -L -o unix_intro.tar.gz https://ndownloader.figshare.com/files/15573746
tar -xzvf unix_intro.tar.gz && rm unix_intro.tar.gz
cd unix_intro
```

Unlike date, most commands require arguments and won't work without them. head is a command that prints the first lines of a file, so it requires us to provide the file we want it to act on:

```
head example.txt
```

Here "example.txt" is the required argument, and in this case it is also what's known as a positional argument. Whether things need to be provided as positional arguments or not depends on how the command or program we are using was written. Sometimes we need to specify the input file by putting something in front of it (e.g. some commands will use the -i flag, but it's often other things as well).

There are also optional arguments for the head command. The default for head is to print the first 10 lines of a file. We can change that by specifying the -n flag, followed by how many lines we want:

```
head -n 5 example.txt
```

How would we know we needed the -n flag for that? There are a few ways to find out. Many standard Unix commands and other programs will have built-in help menus that we can access by providing –help as the only argument:

```
head --help
```

Again this is very similar to the logic in which R commands are strucutred e.g. `??ggplot()` The synatx is similar even if the specific icons or arguments are different.

Remember just like with R, one of your best friends is Google! As you get familiar with any language or OS we might remember a few flags or specific options, but searching for options and details when needed is definitely the norm!

2.6 1.6 Unix File Structure

There are two special locations in all Unix-based systems: the “root” location and the current user’s “home” location. “Root” is where the address system of the computer starts; “home” is where the current user’s location starts.

IMAGE

We tell the command line where files and directories are located by providing their address, their “path”. If we use the pwd command (for print working directory), we can find out what the path is for the directory we are sitting in. And if we use the ls command (for list), we can see what directories and files are in the current directory we are sitting in.

```
pwd
ls
```

2.7 1.7 Absolute vs relative file paths

You should be used to these concepts from your work with R projects.

There are two ways to specify the path (address) of the file we want to do something to:

- An **absolute path** is an address that starts from an explicitly specified location: either the “root” `/` or the “home” `~/` location. (Side note, because we also may see or hear the term, the “full path”, is the absolute path that starts from the “root” /.)
- A **relative path** is an address that starts from wherever we are currently sitting. For example, let’s look again at the head command we ran above:

```
head example.txt
```

What we are actually doing here is using a relative path to specify where the “example.txt” file is located. This is because the command line **automatically looks** in the current working directory if we don’t specify anything else about its location.

We can also run the same command on the same file using an **absolute path**:

```
    head ~/unix_intro/example.txt
```

The previous two commands both point to the same file right now. But the first way, head example.txt, will only work if we are entering it while "sitting" in the directory that holds that file, while the second way will work no matter where we happen to be in the computer.

It is **important to always think about where** we are in the computer when working at the command line. One of the most common errors/easiest mistakes to make is trying to do something to a file that isn't where we think it is. Let's run head on the "example.txt" file again, and then let's try it on another file: "notes.txt":

```
head example.txt
head notes.txt
```

Here the head command works fine on "example.txt", but we get an error message when we call it on "notes.txt" telling us no such file or directory. If we run the ls command to list the contents of the current working directory, we can see the computer is absolutely right – spoiler alert: it usually is – and there is no file here named "notes.txt".

The ls command by default operates on the current working directory if we don't specify any location, but we can tell it to list the contents of a different directory by providing it as a positional argument:

```
    ls
    ls experiment
```

We can see the file we were looking for is located in the subdirectory called "experiment". Here is how we can run head on "notes.txt" by specifying an accurate relative path to that file:

```
    head experiment/notes.txt
```

2.8 1.8 Moving around

We can also move into the directory containing the file we want to work with by using the `cd` command (**c**hange **d**irectory). This command takes a positional argument that is the path (address) of the directory we want to change into. This can be a relative path or an absolute path. Here we'll use the relative path of the subdirectory, "experiment", to change into it

```
cd experiment/
pwd
ls
head notes.txt
```

Great. But now how do we get back "up" to the directory above us? One way would be to provide an absolute path, like `cd ~/unix_intro`, but there is also a handy shortcut. `..` are special characters that act as a relative path specifying "up" one level – one directory – from wherever we currently are. So we can provide that as the positional argument to cd to get back to where we started:

```
cd ..
pwd
ls
```

Moving around the computer like this might feel a bit cumbersome and frustrating at first, but after spending a little time with it, you will get used to it, and it starts to feel more natural.

Get used to using **tab** to perform **tab-completion** often this will auto-complete file names!

- Press tab twice quickly and it will print all possible combinations

2.9 1.9 Summary

While maybe not all that exciting, these things really are the foundation needed to start utilizing the command line – which then gives us the capability to use lots of tools that only work at a command line, manipulate large files rapidly, access and work with remote computers, and more! These are the fundamental tools that every scientist needs to work with **big data**.

2.9.1 Terms

Term	What it is
path	the address system the computer uses to keep track of files and directories
root	where the address system of the computer starts, /
home	where the current user's location starts, ~/
absolute path	an address that starts from a specified location, i.e. root, or home
relative path	an address that starts from wherever we are
tab-completion	our best friend

2.9.2 Commands

Command	What it is
date	prints out information about the current date and time
head	prints out the first lines of a file
pwd	prints out where we are in the computer (print working directory)
ls	lists contents of a directory (list)
cd	change directories

2.9.3 Special characters

Command	What it is
Characters	Meaning
/	the computer's root location
~/	the user's home location
../	specifies a directory one level "above" the current working directory

2.10 1.10 Stretch yourself

https://bioinformaticsworkbook.org/Appendix/Unix/unix-basics-1.html#gsc.tab=0

http://evomics.org/learning/unix-tutorial/

Reference a figure by its code chunk label with the `fig:` prefix, e.g., see Figure 2.1. Similarly, you can reference tables generated from `knitr::kable()`, e.g., see Table 2.5.

```
knitr::kable(
  head(iris, 20), caption = 'Here is a nice table!',
  booktabs = TRUE
)
```

You can write citations, too. For example, we are using the **bookdown** package (Xie, 2020) in this sample book, which was built on top of R Markdown and **knitr** (Xie, 2015).

Something

Something

Table 2.5: Here is a nice table!

Sepal.Length	Sepal.Width	Petal.Length	Petal.Width	Species
5.1	3.5	1.4	0.2	setosa
4.9	3.0	1.4	0.2	setosa
4.7	3.2	1.3	0.2	setosa
4.6	3.1	1.5	0.2	setosa
5.0	3.6	1.4	0.2	setosa
5.4	3.9	1.7	0.4	setosa
4.6	3.4	1.4	0.3	setosa
5.0	3.4	1.5	0.2	setosa
4.4	2.9	1.4	0.2	setosa
4.9	3.1	1.5	0.1	setosa
5.4	3.7	1.5	0.2	setosa
4.8	3.4	1.6	0.2	setosa
4.8	3.0	1.4	0.1	setosa
4.3	3.0	1.1	0.1	setosa
5.8	4.0	1.2	0.2	setosa
5.7	4.4	1.5	0.4	setosa
5.4	3.9	1.3	0.4	setosa
5.1	3.5	1.4	0.3	setosa
5.7	3.8	1.7	0.3	setosa
5.1	3.8	1.5	0.3	setosa

Something

Chapter 3

NGS sequence analysis

3.1 Background

Info on sequencing

General application

Don't have to follow everything!

3.2 The data

For a quick overview of the example data we'll be using and where it came from, we are going to work with a subset of the dataset published here. We were exploring an underwater mountain ~3 km down at the bottom of the Pacific Ocean that serves as a low-temperature (~5-10°C) hydrothermal venting site. This amplicon dataset was generated from DNA extracted from crushed basalts collected from across the mountain with the goal being to begin characterizing the microbial communities of these deep-sea rocks. No one had ever been here before, so as is often the purpose of marker-gene sequencing, this was just a broad-level community survey. The sequencing was done on the Illumina MiSeq platform with 2x300 paired-end sequencing using primers targeting the V4 region of the 16S rRNA gene. There are 20 samples total: 4 extraction "blanks" (nothing added to DNA extraction kit), 2 bottom-water samples, 13 rocks, and one biofilm scraped off of a rock. None of these details are important for you to remember, it's just to give some overview if you care.

In the following figure, overlain on the map are the rock sample collection locations, and the panes on the right show examples of the 3 distinct types of rocks collected: 1) basalts with highly altered, thick outer rinds (>1 cm); 2) basalts that were smooth, glassy, thin exteriors (~1-2 mm); and 3) one calcified carbonate.

PICTURE

Altogether the uncompressed size of the working directory we are downloading here is ~300MB - this is about 10% of the full dataset - we are using a reduced dataset to minimise system requirements and speed up the workflow.

To get started, be sure you are in the "Terminal" window. We will be working here for the first step of importing the data, and removing the primers from our data. We can import our data using the `curl` function, we will then remove the primers using a program called `cutadapt` which is written in Python.

Make sure when you open the terminal you are in the project directory (and refer to last weeks notes if you need to check how to do this).

Don't switch over to R (the "Console" tab in the Binder/RStudio environment) until noted. You can download the required dataset and files by copying and pasting the following commands into your command-line terminal:

```
curl -L -o dada2_amplicon_ex_workflow.tar.gz https://ndownloader.figshare.com/files
tar -xzvf dada2_amplicon_ex_workflow.tar.gz
rm dada2_amplicon_ex_workflow.tar.gz
cd dada2_amplicon_ex_workflow/
```

In our working directory there are 20 samples with forward (R1) and reverse (R2) reads with per-base-call quality information, so 40 fastq files (.fq). It is a good idea to have a file with all the sample names to use for various things throughout, so here's making that file based on how these sample names are formatted.

```
ls *_R1.fq | cut -f1 -d "_" > samples
```

3.3 Removing Primers

To start, we need to remove the primers from all of these (the primers used for this run are in the "primers.fa" file in our working directory), and here we're going to use cutadapt to do that at the command line ("Terminal" tab). Cutadapt operates on one sample at at time, so we're going to use a wonderful little bash *loop* to run it on all of our samples.

3.3.1 Loops

Loops are extremely powerful way of controlling iteration. We can specify that a line of code is repeated across multiple objects. In this example we use the sample file we made earlier as the list of files across which we want this function of removing primers to loop. These same lines will then repeat until all the specified iterations are complete.

We won't break down exactly how this loop works - but they are used across all programming languages (including R) and you can check out the R4DS book for an introduction to building your own loops (and custom functions!) here.

For now just copy and paste this code exactly into the Terminal.

```
for sample in $(cat samples)
do

    echo "On sample: $sample"

    cutadapt -a ^GTGCCAGCMGCCGCGGTAA...ATTAGAWACCCBDGTAGTCC \
    -A ^GGACTACHVGGGTWTCTAAT...TTACCGCGGCKGCTGGCAC \
    -m 215 -M 285 --discard-untrimmed \
    -o ${sample}_sub_R1_trimmed.fq.gz -p ${sample}_sub_R2_trimmed.fq.gz \
    ${sample}_sub_R1.fq ${sample}_sub_R2.fq \
    >> cutadapt_primer_trimming_stats.txt 2>&1

done
```

Here's a before-and-after of one of our files

```
### R1 BEFORE TRIMMING PRIMERS
head -n 2 B1_sub_R1.fq
# @M02542:42:000000000-ABVHU:1:1101:8823:2303 1:N:0:3
# GTGCCAGCAGCCGCGGTAATACGTAGGGTGCGAGCGTTAATCGGAATTACTGGGCGTAAAGCGTGCGCAGGCGGTCTTGT
# AAGACAGAGGTGAAATCCCTGGGCTCAACCTAGGAATGGCCTTTGTGACTGCAAGGCTGGAGTGCGGCAGAGGGGGATGG
# AATTCCGCGTGTAGCAGTGAAATGCGTAGATATGCGGAGGAACACCGATGGCGAAGGCAGTCCCCTGGGCCTGCACTGAC
# GCTCATGCACGAAAGCGTGGGGAGCAAACAGGATTAGATACCCGGGTAGTCC

### R1 AFTER TRIMMING PRIMERS
head -n 2 B1_sub_R1_trimmed.fq
# @M02542:42:000000000-ABVHU:1:1101:8823:2303 1:N:0:3
# TACGTAGGGTGCGAGCGTTAATCGGAATTACTGGGCGTAAAGCGTGCGCAGGCGGTCTTGTAAGACAGAGGTGAAATCCC
# TGGGCTCAACCTAGGAATGGCCTTTGTGACTGCAAGGCTGGAGTGCGGCAGAGGGGGATGGAATTCCGCGTGTAGCAGTG
# AAATGCGTAGATATGCGGAGGAACACCGATGGCGAAGGCAGTCCCCTGGGCCTGCACTGACGCTCATGCACGAAAGCGTG
# GGGAGCAAACAGG
```

You can look through the output of the cutadapt stats file we made ("cutadapt_primer_trimming_stats.txt") to get an idea of how things went. Here's a little one-liner to look at what fraction of reads were retained in each sample (column 2) and what fraction of bps were retained in each sample (column 3):

```
paste samples <(grep "passing" cutadapt_primer_trimming_stats.txt | cut -f3 -d "(" | tr -d ")
```

```
# B1    96.5%   83.0%
# B2    96.6%   83.3%
# B3    95.4%   82.4%
# B4    96.8%   83.4%
# BW1   96.4%   83.0%
# BW2   94.6%   81.6%
# R10   92.4%   79.8%
# R11BF 90.6%   78.2%
# R11   93.3%   80.6%
# R12   94.3%   81.4%
# R1A   93.3%   80.5%
# R1B   94.0%   81.1%
# R2    94.0%   81.2%
# R3    93.8%   81.0%
# R4    95.5%   82.4%
# R5    93.7%   80.9%
# R6    92.7%   80.1%
# R7    94.4%   81.5%
# R8    93.2%   80.4%
# R9    92.4%   79.7%
```

We would expect to lose around 13-14% of bps just for cutting off the primers, and the remainder of lost bps would be from the relatively low percent of those reads totally removed (~92-97% across the samples).

With primers removed, we're now ready to switch R and start using DADA2!

3.4 DADA2 - switch to R console

```
library(dada2)

setwd("~/dada2_amplicon_ex_workflow")

list.files() # make sure what we think is here is actually here

## first we're setting a few variables we're going to use ##
  # one with all sample names, by scanning our "samples" file we made earlier

samples <- scan("samples", what="character")

  # one holding the file names of all the forward reads
forward_reads <- paste0(samples, "_sub_R1_trimmed.fq.gz")

  # and one with the reverse
reverse_reads <- paste0(samples, "_sub_R2_trimmed.fq.gz")
```

```
  # and variables holding file names for the forward and reverse
  # filtered reads we're going to generate below
filtered_forward_reads <- paste0(samples, "_sub_R1_filtered.fq.gz")
filtered_reverse_reads <- paste0(samples, "_sub_R2_filtered.fq.gz")
```

3.5 Quality trimming/filtering

We did a filtering step above with cutadapt (where we eliminated reads that had imperfect or missing primers and those that were shorter than 215 bps or longer than 285), but in DADA2 we'll implement a trimming step as well (where we trim reads down based on some quality threshold rather than throwing the read away). Since we're potentially shortening reads further, we're again going to include another minimum-length filtering component. We can also take advantage of a handy quality plotting function that DADA2 provides to visualize how you're reads are doing, plotQualityProfile(). By running that on our variables that hold all of our forward and reverse read filenames, we can easily generate plots for all samples or for a subset of them. So let's take a peak at that to help decide our trimming lengths: It's good to try to keep a bird's-eye view of what's going on. So here is an overview of the main processing steps we'll be performing with cutadapt and DADA2. Don't worry if anything seems unclear right now, we will discuss each at each step.

```
plotQualityProfile(forward_reads[17:20])

plotQualityProfile(reverse_reads[17:20])
```

All forwards look pretty similar to eachother, and all reverses look pretty similar to eachother, but worse than the forwards, which is common – chemistry gets tired…

On the plots produced

- the x axis is the nucleotide bases starting from the beginning of the read moving to the end
- the y axis is the average quality score for the base in that position
- the green line is the median quality score of the base at that position
- the orange lines are quartiles

TALK ABOUT PHRED SCORES

Here, I'm going to cut the forward reads at 250 and the reverse reads at 200 – roughly where both sets maintain a median quality of 30 or above – and then see how things look. But we also want to set a minimum length to filter out truncated sequences, so we will set a minimum acceptable read length of 175bp (any reads shorter than this will be discarded).

In DADA2, this quality-filtering step is done with the `filterAndTrim()` function:

```
    filtered_out <- filterAndTrim(forward_reads, filtered_forward_reads,
                    reverse_reads, filtered_reverse_reads, maxEE=c(2,2),
                    rm.phix=TRUE, minLen=175, truncLen=c(250,200))
```

This function made a bunch of output files "filtered_forward_reads" and "filtered_reverse_reads" we can see these in our project pane. Or if we were working on a server without a GUI we could use `list.files()` in R or `ls` in our Terminal.

We also generated an R file called filtered_out. This is a simple matrix holding how many reads went *in* for each file and how many came *out*.

Check it in R.

```
    filtered_out
```

We can take a look at the filtered reads visually - we expect to have trimmed off that section where quality drops

```
    plotQualityProfile(filtered_reverse_reads[17:20])
```

Looking Good!

3.6 Dereplication

Dereplication is a common step in many amplicon processing workflows. Instead of keeping 100 identical sequences and doing all downstream processing to all 100 -costing computer processing power and time, you can keep/process just one of them, and just attach the number x100 to it. Now this acts as a representative for 100 identical sequences.

```
derep_forward <- derepFastq(filtered_forward_reads, verbose=TRUE)
names(derep_forward) <- samples # the sample names in these objects are initially the
derep_reverse <- derepFastq(filtered_reverse_reads, verbose=TRUE)
names(derep_reverse) <- samples
```

3.7 ASV's

This is where we start to take our raw sequence data and infer *true* biological sequences. It uses an algorithm to look at the consensus quality score and abundance for each *unique* sequence. It then determines whether this sequence is more likely to be of biological origin or a spurious sequencing error.

```
dada_forward <- dada(derep_forward, err=err_forward_reads, pool="pseudo")

dada_reverse <- dada(derep_reverse, err=err_reverse_reads, pool="pseudo")
```

3.8 Merging reads

Now DADA2 merges the forward and reverse ASVs to reconstruct our full target amplicon requiring the overlapping region to be identical between the two. By default it requires that at least 12 bps overlap, but in our case the overlap should be much greater. If you remember above we trimmed the forward reads to 250 and the reverse to 200, and our primers were 515f–806r. After cutting off the primers we're expecting a typical amplicon size of around 260 bases, so our typical overlap should be up around 190. That's estimated based on E. coli 16S rRNA gene positions and very back-of-the-envelope-esque of course, so to allow for true biological variation and such I'm going ot set the minimum overlap for this dataset for 170. I'm also setting the trimOverhang option to TRUE in case any of our reads go passed their opposite primers (which I wouldn't expect based on our trimming, but is possible due to the region and sequencing method).

```
merged_amplicons <- mergePairs(dada_forward, derep_forward, dada_reverse,
                    derep_reverse, trimOverhang=TRUE, minOverlap=170)

  # this object holds a lot of information that may be the first place you'd want to look if you
class(merged_amplicons) # list
length(merged_amplicons) # 20 elements in this list, one for each of our samples
names(merged_amplicons) # the names() function gives us the name of each element of the list

class(merged_amplicons$B1) # each element of the list is a dataframe that can be accessed and mar

names(merged_amplicons$B1) # the names() function on a dataframe gives you the column names
# "sequence"  "abundance" "forward"   "reverse"   "nmatch"    "nmismatch" "nindel"    "prefer"
```

3.9 Count table

Now we can generate a count table with the makeSequenceTable() function. This is one of the main outputs from processing an amplicon dataset. You may have also heard this referred to as a biome table, or an OTU matrix.

```
seqtab <- makeSequenceTable(merged_amplicons)
class(seqtab) # matrix
dim(seqtab) # 20 2521
```

3.10 Overview

The developers' DADA2 tutorial provides an example of a nice, quick way to pull out how many reads were dropped at various points of the pipeline. This can serve as a jumping off point if you're left with too few sequences at the end to help point you towards where you should start digging into where they are being dropped. Here's a slightly modified version:

```
  # set a little function
getN <- function(x) sum(getUniques(x))

  # making a little table
summary_tab <- data.frame(row.names=samples, dada2_input=filtered_out[,1],
               filtered=filtered_out[,2], dada_f=sapply(dada_forward, getN),
               dada_r=sapply(dada_reverse, getN), merged=sapply(merged_amplicons, getN
               nonchim=rowSums(seqtab.nochim),
               final_perc_reads_retained=round(rowSums(seqtab.nochim)/filtered_out[,1]

summary_tab
```

And it might be useful to write this table out of R, saving it as a regular file

```
write.table(summary_tab, "read-count-tracking.tsv", quote=FALSE, sep="\t", col.names=N
```

3.11 Assign taxonomy

```
## downloading DECIPHER-formatted SILVA v138 reference
download.file(url="http://www2.decipher.codes/Classification/TrainingSets/SILVA_SSU_r1

## loading reference taxonomy object
load("SILVA_SSU_r138_2019.RData")
```

Running the following taxonomy assignment step took ~30 minutes on a 2013 MacBook Pro. So feel free to load the stored R objects with load("amplicon_dada2_ex.RData") to skip this step if you'd like.

```
## loading DECIPHER
library(DECIPHER)

## creating DNAStringSet object of our ASVs
dna <- DNAStringSet(getSequences(seqtab.nochim))

## and classifying
tax_info <- IdTaxa(test=dna, trainingSet=trainingSet, strand="both", processors=NULL)
```

3.12 Standard goods

The typical standard outputs from amplicon processing are a fasta file, a count table, and a taxonomy table. So here's one way we can generate those files from your DADA2 objects in R:

```
  # giving our seq headers more manageable names (ASV_1, ASV_2...)
asv_seqs <- colnames(seqtab.nochim)
asv_headers <- vector(dim(seqtab.nochim)[2], mode="character")

for (i in 1:dim(seqtab.nochim)[2]) {
  asv_headers[i] <- paste(">ASV", i, sep="_")
}

  # making and writing out a fasta of our final ASV seqs:
asv_fasta <- c(rbind(asv_headers, asv_seqs))
write(asv_fasta, "ASVs.fa")

  # count table:
asv_tab <- t(seqtab.nochim)
row.names(asv_tab) <- sub(">", "", asv_headers)
write.table(asv_tab, "ASVs_counts.tsv", sep="\t", quote=F, col.names=NA)

  # tax table:
  # creating table of taxonomy and setting any that are unclassified as "NA"
ranks <- c("domain", "phylum", "class", "order", "family", "genus", "species")
asv_tax <- t(sapply(tax_info, function(x) {
  m <- match(ranks, x$rank)
  taxa <- x$taxon[m]
  taxa[startsWith(taxa, "unclassified_")] <- NA
  taxa
}))
colnames(asv_tax) <- ranks
rownames(asv_tax) <- gsub(pattern=">", replacement="", x=asv_headers)
```

```
write.table(asv_tax, "ASVs_taxonomy.tsv", sep = "\t", quote=F, col.names=NA)
```

3.13 Contaminants

Command	What it is
cutadapt/filterAndTrim()	remove primers and quality trim/filter
learnErrors()	generate an error model of our data
derepFastq	dereplicate sequences
dada()	infer ASVs on both forward and reverse reads independently
mergePairs()	merge forward and reverse reads to further refine ASVs
makeSequenceTable()	generate a count table
removeBimeraDenovo()	screen for and remove chimeras
IdTaxa()	assign taxonomy

Q. Why isn't this data tidy?

The variable `year` is split across six columns.

We want to create a single column **year** that will have the values (2007-2012) in rows. This will make the dataframe much longer as individuals will be repeated six times. At the same time the values in the six columns should be assigned to a new column **length**.

Chapter 4

Analysis in R

```
library("phyloseq")
library("vegan")
library("DESeq2")
library("ggplot2")
library("dendextend")
library("tidyr")
library("viridis")
library("reshape")
```

Make graphs

Answer some questions?

Chapter 5

Applications

Some *significant* applications are demonstrated in this chapter.

5.1 Example one

5.2 Example two

Chapter 6

Final Words

We have finished a nice book.

Bibliography

Xie, Y. (2015). *Dynamic Documents with R and knitr*. Chapman and Hall/CRC, Boca Raton, Florida, 2nd edition. ISBN 978-1498716963.

Xie, Y. (2020). *bookdown: Authoring Books and Technical Documents with R Markdown.* R package version 0.21.