Processing 16S data: an informal primer about 16S rRNA amplicon data

Scott Olesen

Foreword

Where this document came from

I wrote an early form of this document as a teaching aid for a workshop on 16S data processing that I gave for my research group, the Alm Lab at MIT. I eventually put this document up for the larger world because it was starting to be shared in wider circles.

What this document is for

My goal for this document is to help you understand the theory behind 16S data processing. I want you to be able to do the analysis that makes the most sense for you. Processing 16S data involves a lot of very small decisions (that probably have a small effect on your results) and a few big decisions (that certainly have a big effect on your results). The nice pipelines make it possible for you to shut your eyes to these complications. In contrast, I want to empower you to be able to critique and doubt other people's methods. That's part of how science advances!

Who this document is for

This document should be a fun read for advanced undergraduates, graduate students, and postdocs who are interested in getting down and dirty with 16S data. I expect you to already know that you want to use 16S data or you already have 16S data in hand and are trying to figure out what to make of it.

If you know too much, I expect this document will annoy you, because you will recognize its shortcomings.

What this document is not

This document is not perfect. It is full of my own ignorance, ideas, and opinions. Take it with a grain of salt. If you find errors, blame me and not my adviser or the other members of my lab. And do point them out to me!

This document is also not intended to be a literature review. Next-generation sequencing for microbial ecology is an enormous field, and here I just scratch the surface of that field. Some specific methods are included and they show my biases: I'm more familiar with my lab's work and with the way my labmates and I process our data.

This document is not a tutorial. If you desperately need to turn some raw data into OTU tables in the next 10 hours, don't read this document. If you want to know something more principled about *how* to turn fastq's into OTU tables, read on.

Where 16S data comes from

By "16S data" I mean amplicon sequencing of some section of the bacterial 16S gene. A lot of what I discuss in this document may also apply to other types of taxonomic marker sequences like the eukaryotic 18S or the fungal ITS.

The 16S gene

All bacteria (and archaea)[1] have at least one copy of the 16S gene in their genome.[2] The gene has some sections that are *conserved*, meaning that they are very similar across all bacteria, and some sections that are *variable* (or "hypervariable"). The idea behind 16S sequencing is that the variable regions are not under strong evolutionary pressure, so random mutations accumulate there. Closely-related bacteria will have more similar variable regions than distantly-related bacteria.

To get a feel for the variability at different positions in the gene, I looked at the aligned 94% OTUs in Greengenes. "Variability" (which I made up for the purposes of this figure) at a position in the gene is one minus the fraction of OTUs that had the most common nucleotide at that position. For example, if all OTUs had `A` at a position, the "variability" is 0. If half of the OTUs had `A` at that position, the "variability" is 0.5. The actual plot is very spiky, so I smoothed

[1]Chloroplasts, found in algae and other eukaryotes, have a ribosomal gene that is very similar to 16S and often ends up getting amplified in 16S data sets.

[2]It's not unusual for bacteria to have multiple copies of the 16S gene, and those copies might not be identical to one another. Some people are concerned by the effect this could have on interpretations of 16S data (e.g., Kembel *et al.*; doi:10.1371/journal.pcbi.1002743 and Case *et al.*; doi:10.1128/AEM.01177-06).

over 50 nucleotide windows. This shows you that there are subsequences of the gene that are more variable than others.

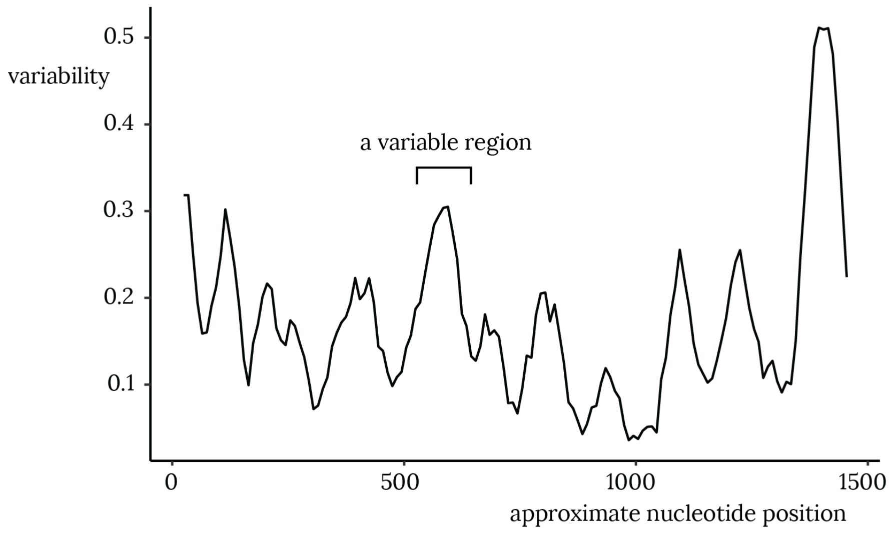

Figure 1: The 16S gene has variable regions.

Getting DNA from a sample

After a sample is taken, the cells in the sample are lysed, typically using some combination of chemical membrane-dissolving and physical membrane-busting. The DNA in the sample is *extracted*, meaning that all the protein, lipids, and other stuff in the sample is thrown away. From this pile of DNA spaghetti, we aim to collect information about the bacteria that were in the original sample.

Amplifying the gene

In the amplicon sequencing approach, PCR is used to amplify a section of the 16S gene. The size of the sequenced section is limited by the length of reads produced by high-throughput sequencing. The sections of the 16S gene that are amplified are named according to what variable regions of the gene are covered. There are nine variable regions, but there isn't an exact definition of where they begin and end. Different regions can provide different taxonomically resolution for different parts of the microbial tree of life.

Some regions I've seen amplified include:

- *V1-V2* (the first two variable regions). This section of the gene provides better taxonomic resolution for some bacteria associated with the skin

microbiome, so skin studies sometimes sequence V1-V2.
- *V4* (variable region 4). This section of the gene provides good taxonomic resolution for bacteria associated with the gut microbiome, so it is the most popular section. I get the sense that V4 is also the best "catch-all" region, but I don't know of a good reference to back up that sense.
- *V5-V6*. I've only seen this section in projects that aim to be particularly complete (e.g., the Human Microbiome Project).

PCR reactions on these regions have primers that match the constant regions around the variable regions. Papers should always mention which primers they used, and they usually also mention the amplified region. The primers have names like 8F (meaning, a forward primer starting at nucleotide 8 in the gene) and 1492R (meaning, a reverse primer starting at nucleotide 1492).

Amplicon sequencing vs. shotgun ("metagenomic") sequencing

First, a note on terminology. Technically, "metagenomic" means "related to more than one genome", that is, sampling from an entire community rather than from a single cell or a single colony. In common speech, I hear "metagenomics" used to mean *shotgun* metagenomics (i.e., trying to sequence all the DNA in a sample), as opposed to amplicon sequencing (even though amplicon sequencing, as we're talking about, is metagenomic, since it usually involves many different species).

In theory, amplicon sequencing gives you strictly less information that shotgun sequencing. If you shotgun sequence at a great enough sequencing depth, you should be able to reconstruct all the information that you could get from 16S amplicon sequencing.

In practice, the amplicon-based approach has some advantages. Because only bacteria and archaea have the 16S gene, a tube of 16S amplicon DNA mostly carries information about microbes. In contrast, the majority of shotgun reads from, say, a swab of human skin will be human DNA. This means that, for the same depth of sequencing, 16S amplicon sequencing will provide much more information about the bacterial community structure than will shotgun sequencing. Shotgun sequencing projects tend to be more expensive, since they need to sequence deeper to get the interesting information.

Amplicon sequence data is also easier to work with from a bioinformatic point of view. Amplicon sequences come "pre-aligned". (They're not *actually* "aligned" in the bioinformatic sense, but it's a good analogy.) This means that you know where each read came from: the 16S gene, from this nucleotide position to that nucleotide position. Every read therefore provides, roughly speaking, an equal amount of information about the composition of the sample's bacterial community. Shotgun sequences, in contrast, need to be assembled, which is far more complicated bioinformatic process than anything you will find in this document.

The amplified DNA is not exactly like the original DNA

The process of extracting DNA from bacterial cells and then amplifying a 16S region introduces certain biases into the resulting sequence data. These effects mean that it's better to look for changes in bacterial community structure rather than assert that such-and-such a species is more abundant than other-and-such a species. It also means that large effects, like variations over orders of magnitude, are to be trusted far more than smaller changes. The grains of salt to be kept in mind are *extraction bias*, *PCR bias*, PCR *chimeras*, and lab-specific effects.

Extraction bias

Different cells respond differently to different extraction protocols. Splitting a sample and using different extraction protocols on the different parts can produce markedly different results. There are many papers about this[3]. My takeaway is that, if you're comparing two data sets, it's important to know if they used the same extraction methodology, since differences in the 16S data could be due to differences in the microbes themselves or just in the methods used to extract the DNA.

PCR bias

I don't think *PCR bias* is a huge problem, but it's good to have heard about it. First, although we say the PCR primers bind a "constant" region, there is still variation in those regions. Thus, some bacteria in the sample will have different nucleotides at the primer binding site, meaning that the PCR primers will bind with different affinities to the DNA of different bacteria. This effect decreases the number of reads from bacteria whose constant regions don't match the primer.[4]

"PCR bias" encompasses other things beyond primer site binding bias. It's known that PCR has different efficiencies for different types of sequences, meaning that some 16S variable regions will amplify better than others. Also, statistical fluctuations can occur, especially in low-diversity samples. This means that a sequence that, by chance, gets lots of amplification in early PCR cycles could dominate the sample in late PCR cycles.

In general, PCR bias is not as bad when there is more DNA and (relatedly) when the PCR is run for fewer cycles.

In our lab, some folks have run experiments to quantify PCR bias: they synthesize some DNA that looks like bacterial 16S genes, mix that DNA in known

[3]A quick web search gave me: Salter *et al.* (doi:10.1186/s12915-014-0087-z), Walker *et al.* (doi:10.1371/journal.pone.0088982), Rochelle *et al.* (doi:10.1016/0378-1097(92)90188-T), etc. But compare Rubin *et al.* (doi:10.1002/mbo3.216).

[4]It may be that there are a lot of interesting bugs whose 16S sequences are so divergent that they don't match the typical primers (cf. Brown *et al.*; doi:10.1038/nature14486).

proportions, amplify the mixture, sequence the amplified DNA, and compare the sequencing data to the known proportions of input DNA. The errors are somewhere in the neighborhood of 1% to 10%. It's not high enough to make me think that 16S data is all garbage, but it is high enough to make me doubt small (say, two-fold) changes in composition.

Chimeras

PCR also creates weird artifacts called *chimeras*. When using PCR to amplify two DNA sequences *a* and *b*, you'll get a lot of *a*, a lot of *b*, and some sequences that have an *a* head and *b* tail (or vice versa). There are some things about your PCR protocol that you can adjust to decrease the prevalence of chimeras.

Lab-specific effects

There are also biases that arise from *any* DNA-based experiment, like the biases that result from the method of collection or storage. Some people run mini-studies to ask about the effects of storage at different temperatures for different times, the effect of the buffer used, etc. Regardless of what method of collection and storage is used, using the same method for every sample in a study is a good way to reduce biases.

I suspect that a stronger signal comes from reagent or lab-specific contamination. Commercial reagents often come pre-loaded with bacteria. (Like, not in a good way.) In our lab, we notice that *Halomonas* and *Shewanella* species often appear where they shouldn't, and the effect appears to depend on the particular extraction kit we use.

Multiplexing helps evaluate contamination (or other weirdness)

Next-generation sequencing became more helpful to microbial ecology when *sample multiplexing* (or "barcoding") was worked out in the early 2000s.[5] Before multiplexing, every sample had to be run on its own sequencing lane. This was expensive and bioinformatically annoying, since, especially in those dark ages of sequencing, weird stuff would frequently happen during sequencing runs, and it was hard to distinguish a bad lane from a weird sample.

Multiplexing, by contrast, adds a barcode or "tag" to the 16S amplicon. Each barcode corresponds to a sample, and all amplicons in that sample get that barcode. It's now common to multiplex 96 (or 384) samples and sequence them in one lane. Aside from making the sequencing 100-fold cheaper, multiplexing means that it's easier to include some controls in each lane:

[5] Cf., e.g., Binladen *et al.* (doi:10.1371/journal.pone.0000197).

- *Negative controls*: Usually just vehicle with no DNA, as a way to check for contamination from reagents or poor sample preparation.
- *Positive controls*: Mock communities of known composition, which can be used to check that the sequencing was not "weird". If you have a lot of samples from the same project and you need to run them in more than one lane, you can use the positive controls as an internal check that sequencing proceeded similarly across lanes.

The bioinformatic cost of multiplexing is that the reads must be *demultiplexed* in the analysis stage. This is easy.

Sequencing

A little more work has to be done before putting the sample in the sequencer. These steps will depend on the sequencing platform. Here I'll talk about Illumina because it's popular[6] and I have experience with it. If you're using a different sequencing platform, then you'll need to learn about the quirks of that platform elsewhere.

Samples to be sequenced on an Illumina machine need to have Illumina-specific *adapters* added in a third PCR (one for amplification, one to add the barcodes, and one to add the adapters). These adapters allow the DNA amplicons to bind the flowcell, where they are sequenced. It is sometimes also desirable to have a *diversity region* added between the adapter and the 16S primer. The Illumina sequencers expect to see a diversity of nucleotides at every read position. In amplicon sequencing, almost all the reads are the same through the primer region, which freaks out the sequencer. A diversity region is just some random nucleotides that helps the sequencer do its job. In our lab we use `YRYR`, where `Y` means `T` or `C` and `R` means `G` or `A`.

All of these pieces —the 16S region you're interested in, forward and reverse primers, barcodes, diversity region, and Illumina adapters— are all made into a single *PCR construct*, which is a single piece of DNA. The sequencer reads the nucleotides in the construct and uses its knowledge about the arrangement of the construct to infer which nucleotides are the region of interest and which are the barcode.

Data processing theory

Hooray, you have sequence data! Now what? I'll assume you did paired-end sequencing (but I'll make a note for those of you who didn't). Notably, all 454 sequencing is single-ended.

[6] Interestingly, it wasn't that long ago that 454 (or "Roche") sequencing led the next-generation field. Plenty of papers use "pyrosequencing" (the technical word for 454's sequencing methodology) as a synonym for "next-generation sequencing".

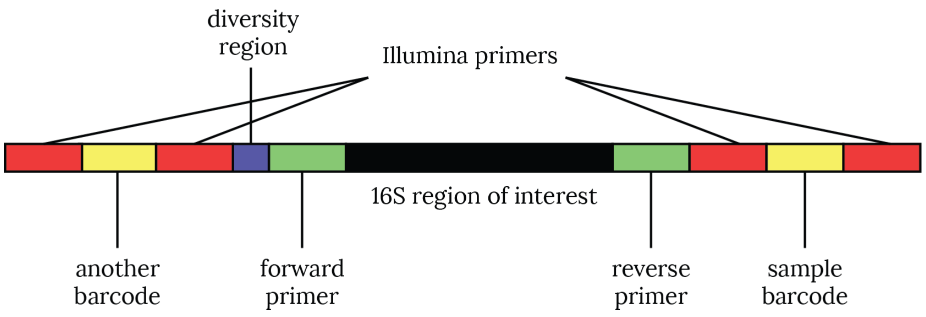

Figure 2: An example PCR construct.

Raw data and metadata

Before trying to process your dataset, be sure you have the appropriate raw data[7] and metadata. In all cases, this means you'll need one set of reads. For Illumina, this means a *fastq* file that might have a name like:

```
130423Alm_D13-1939_1_sequence.fastq
```

This filename is typical for raw sequencing data: it has information about the date, the group that requested the sequencing, then something about the specifics of the sequencing run. The `_1_` indicates that these were forward reads. (Forward read files might have `_R1_` in place of `_1_` in the filename.)

If you did paired-end sequencing, you will also need the *reverse reads*. This is a fastq file with a name like the forward reads except with `_2_` or `_R2_` in it. Every entry in the reverse reads should match an entry in the forward reads. In most cases, the two pairs of a read appear in analogous places in the two files (i.e., the forward read entry starting on line x in the forward read file corresponds to the reverse read starting on line x in the reverse reads file).

In some cases, the data might be delivered to you having already been split into samples (or *demultiplexed*). This means your (forward, or reverse) reads are not in one big file, they are in multiple files, each named (or in a subfolder that is named) that indicates that those are reads belonging to a certain sample. If you data are not demultiplexed (i.e., you only have one big file of forward reads), then you will also need the *barcode* or *index reads*. Depending on your Illumina version, this information might be in different places. In some datasets, it's in the file with the forward reads. For example, the file I mentioned above, the first line is

[7]In what follows, I'll talk about "raw data", by which I mean data that you would get from the sequencing center. There is actually a more raw kind of data that comes right out of the sequencing machine that gets processed right away. On the Illumina platform, this data is CASAVA. Different version of CASAVA produce slightly different output, and every sequencer might have a different version of CASAVA, so be prepared for slight variations in the format of your raw data.

```
@MISEQ578:1:1101:15129:1752#CCGACA/1
```

The barcode read is `CCGACA` (between the `#` and the `/1`). In other datasets, you might find the index reads in a file with a name that has `_R3_`, `_I_`, or `_I1_` in it.

If you have barcodes, you will also need a *barcode map*. This is the information about what barcode goes with what sample. A common gotcha in 16S data processing is that the barcode map might have barcodes that are reverse complements of what are in the samples.

Finally, you might also need the *primer sequences*. These are the sequences of the primers used in the 16S amplification. In some cases, these have already been removed when you get your data. In other cases, you will have to remove them yourself. Two of the most common gotchas I experience in 16S data processing have to do with primers: (i) you think the primers have been pre-removed but they are actually still present and (ii) you think you have the sequences of the primers but you actually have the reverse complements of those primers.

In summary, you might need as much as:

- Forward reads
- Reverse reads
- Barcode reads
- Barcode map
- Primer sequences

In general, it's very useful to ask for or search for this entire list of things when you're starting to analyze a data set from scratch. If you download a dataset or get it from a collaborator, it might come in an already-processed format. Depending on your purposes, it might be fine to use that data, which will look similar to something we'll get a little further down this pipeline. It's useful to know exactly what the content of the data is before you start working on it. If it lacks any of these parts, make sure to find them before you try to get to work!

Overview of the processing pipeline

16S data analysis breaks down into a few steps:

Phase 0 (pre-processing).

This is the practical, devoid-of-theory step in which you get (a) the data you have into (b) the format you need it in so that you can (c) feed it into whatever software you are using for the next step. This usually means converting the raw data and metadata into a file format that your software knows how to handle, sifting out the barcode reads, that sort of thing.

Phase I

These steps take the raw data and turn it into biologically relevant stuff. There is some freedom about the order in which they can be done.

- *Removing* (or "trimming") *primers*. The primers are a man-made thing. If there was a mismatch between the primer and the DNA of interest, you'll only see the primer. In this sense, the primer sequence masks the true biological content of the DNA of interest. Removing the primers is conservative in the sense that you won't come to any false conclusions about the content of the DNA of interest, but the cost is that, for the most part, the primer sequences matched the biological DNA pretty well. Regardless of the tradeoff, common practice is to cut off the primers.
- *Quality filter* (or "trim") reads. For every nucleotide in every read, the sequencer gives some indication of its assuredness that that base is in fact the base the sequencer reported. This assuredness is called *quality*: if a base has high quality, you can be sure that that base was called correctly. If it has low quality, you should be more skeptical. In general, reads tend to decrease in quality as they extend, meaning that we get less sure that the sequence is correct the further away from the primer we go. Some reads also have overall low quality.[8] Quality filtering removes sequences or parts of sequences that we think we cannot trust.
- *Merging* (or "overlapping" or "assembling" or "stitching") read pairs. When doing paired-end sequencing, it's desirable for the two reads in the pair to overlap in the middle. This produces a single full-length read whose quality in the middle positions is hopefully greater than the quality of either of the two reads that produced it. There's no such thing as "merging" for single-end sequencing.
- *Demultiplexing* (or "splitting"). The man-made barcode sequences are replaced by the names of the samples the sequences came from.

Phase II

These steps compact the data and make it easier to work with when calling OTUs. They can happen simultaneously.

- *Dereplicating*. There are fewer *sequences* (strings of `ACGT`) than there are *reads*. This step identifies the set of unique sequences, which is usually much smaller than the number of reads.
- *Proveniencing* (or "mapping" or "indexing"). How many reads of each

[8]Annoyingly, it's my experience that the first reads in the raw data are substantially worse than most of the reads in the file. In the dataset I'm looking at, the first 3,000 or so reads (of 13.5 million total) have an average quality that is about half of what's typical for that dataset. This means that, if you want to check the quality of the sequences in a dataset, you can't just look at the first few entries, you need to go some ways down into the file.

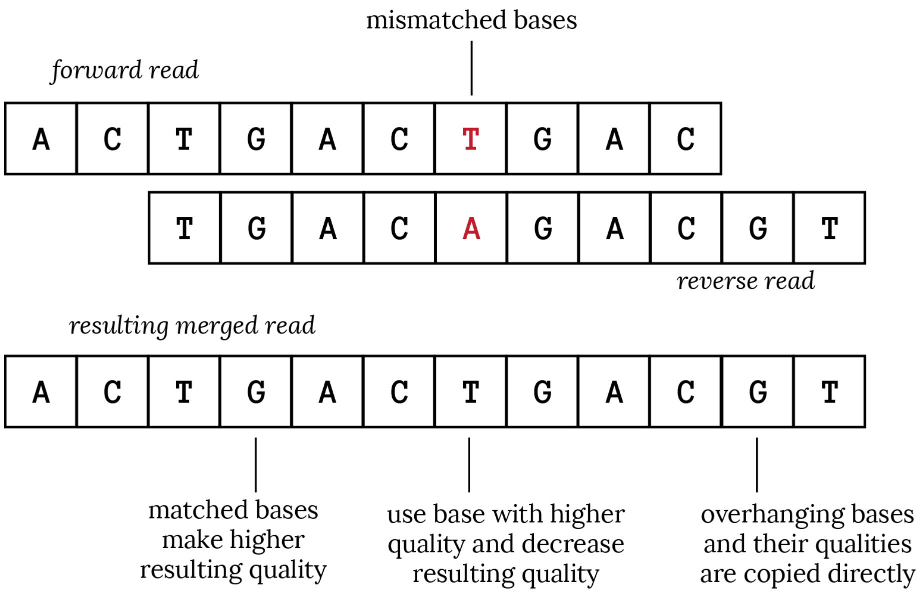
mismatched bases
forward read
A C T G A C T G A C
T G A C A G A C G T
reverse read
resulting merged read
A C T G A C T G A C G T
matched bases make higher resulting quality
use base with higher quality and decrease resulting quality
overhanging bases and their qualities are copied directly

Figure 3: Merging aligns reads, makes a new sequence, and computes new quality scores. Adapted from the `usearch` manual.

sequence were in each sample? (Only I call it "proveniencing"[9]. I find all the other names I've heard confusing.)

Phase III: OTU calling

This is a complex enough endeavor that I will break it out into a separate section. Roughly, OTU calling (or "picking") assigns every dereplicated sequence to a group (called an OTU). You can combine the sequence-to-OTU information from OTU calling with the sequence-to-sample-counts information from proveniencing to make an OTU table that shows how reads mapping to each OTU appear in each sample.

Phase IV: Fun & profit.

The part where you actually use your data! This part is outside the scope of this work. I do, however, encourage you to use the same intellectual attitude that's promulgated here: don't just use an analytical tool because it's popular or because someone else used it.

[9]In archaeology, an artifact's *provenience* is the place within the archaeological site where it was found.

Details of each step

Removing primers

In an ideal world, this is straightforward: you find the piece of your read that matches the primer, and you pop it off. In practice, there are two important considerations:

- *Where do you look for the primer?* Does the primer start at the very first nucleotide of the read, or a little further in? You can put a lot of flexibility in this step without a lot of negative effects, but it's good to know what's going on in your data.
- *What does "match" mean?* How many mismatched nucleotides do you allow between the read and your primer sequence before you consider the read "bad"? In practice, it's common to only keep reads that have at most one error in the primer match.

Merging

Merging is the most complex part of the pre-OTU-calling steps. Merging requires you to:

- *Find the "best" position for merging.* If you were sure all your amplicons are exactly the same size, then this is trivial: just overlap them at the right length. However, even in amplicon sequencing, there are insertions and deletions in the 16S variable regions, so we can't be sure that all merged reads will be the exactly the same length.
- *Decide if the "best" position is good enough.* If you have two reads that don't overlap at all, should you even include it in the downstream analysis?[10] How good is good enough?
- *Compute the quality of the nucleotides in the merged read using the qualities in the original reads.* This requires some basic Bayesian statistics. It's not super-hard, but it was hard enough to be the subject of (among others) a 2010 Alm Lab paper and a 2015 paper from the maker of `usearch`.

I think it's worth noting that `usearch`, which will come up later, is a very popular tool for merging sequences.

Quality filter

Sequences tend to vary in overall quality (some good, some bad) and the number of bases they have that are good. Inherent in the Illumina technology is a trend for sequences to decrease in quality as you move down the sequence.

[10]If you had two paired-end reads that didn't overlap but you were somehow sure of the final amplicon size, then you could insert a bunch of `N`'s in between. This is an advanced and specialized topic.

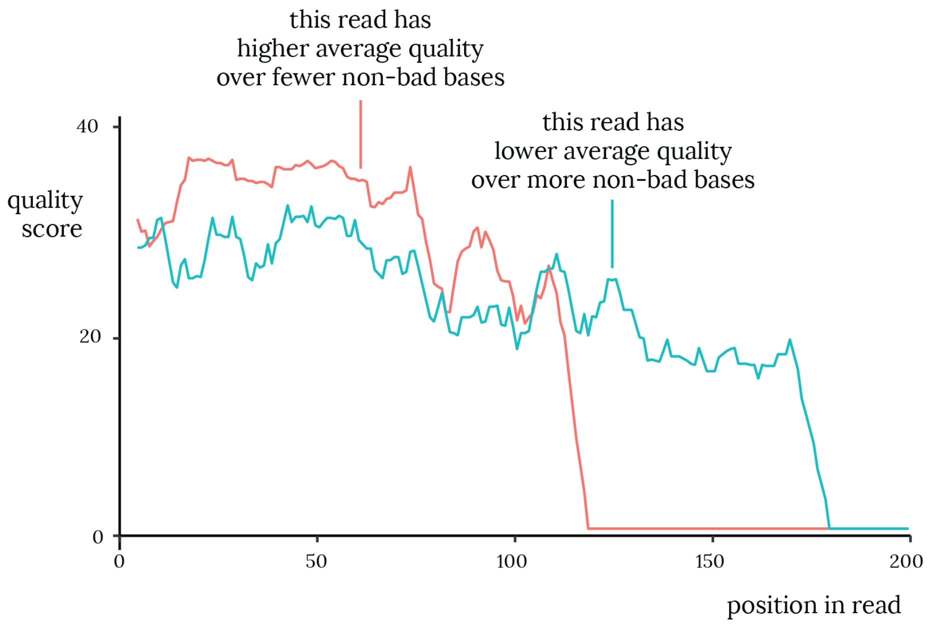

Figure 4: Reads differ in overall quality and number of "non-bad" reads. (data smoothed over 10 bases)

The sequencer will give you a sort of quality report about your sequences' average quality. It will give you a sense of whether your sequencing run as a whole was good, and it will give you a sense of whether you got the sort of good-quality length you were hoping for.

The big quality report gives you a sense of whether you should do the whole sequencing run over again. Even in a good sequencing run there are bad sequences that should be filtered out. There are two common ways to quality filter:

- *Quality trimming* means truncating your reads at some nucleotide after which the sequence is "bad".[11] A common approach is to truncate everything after the first nucleotides whose quality falls below some threshold.
- *Global quality filtering* means discarding an entire read if the average quality of the read is too low. Maybe no individual nucleotide falls below your trim threshold, but the general poor quality of the read means that you'd rather not include it in analysis. This criterion is expressed equivalently as "average quality" or "expected number of errors".[12] When I work with 250

[11]Confusingly, "trimming" also refers to a different process when, if you're doing unpaired amplicon sequencing, you pick a length, discard all reads shorter than that, and truncate all the longer sequences at that length. It is essential to do this when using certain *de novo* OTU calling methods, and it's probably beneficial to do with reference-based OTU calling and or taxonomy assignment methods.

[12]There's a nice paper by Edgar & Flyvbjerg (doi:10.1093/bioinformatics/btv401) that shows

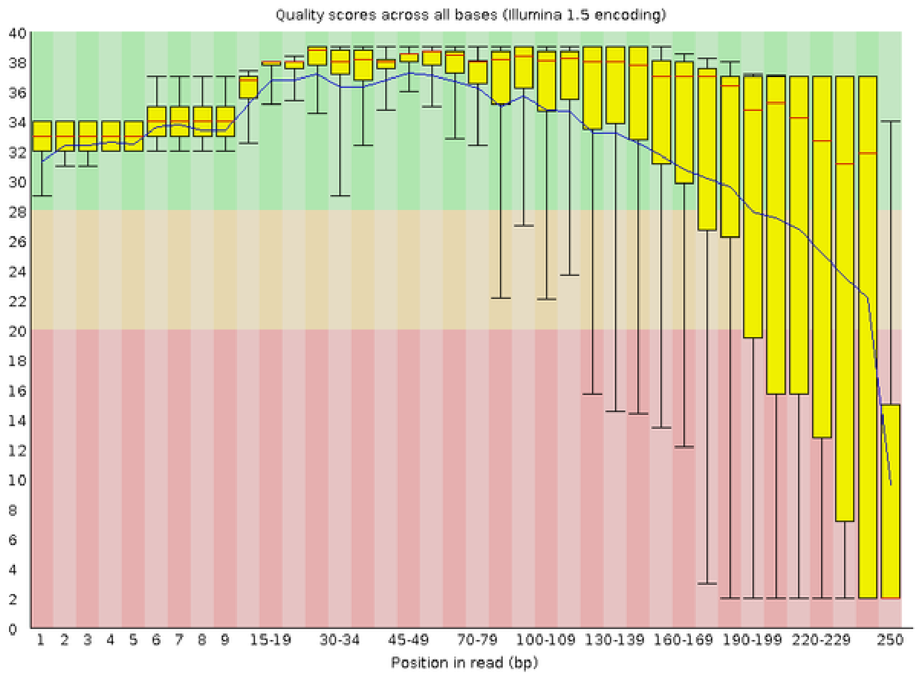

Figure 5: A read quality report delivered by the Illumina software.

bp amplicons, I like to throw out reads that have more than two expected errors.

I like to merge then global quality filter. Other people quality trim then merge. I think it makes more sense to see merge, see what the resulting qualities look like, and make a decision based on that (rather than decide what you think would make a good merged product before you even do the merge).

Demultiplexing

This step is similar to primer removal:

- *Which of the known barcodes is the best match for this barcode read?* That's a pretty straightforward answer. (What if there's a tie, you say? Barcodes are chosen with an error-correcting code so that a tie implies that you have at least two errors in the read.)
- *Is the match with the known barcode good enough?* A common approach is, given a barcode read, to compare that read with all the known barcodes (i.e., the barcodes you're looking for). If the known barcode that matches best has more than one mismatch with the barcode read, call that read "bad" and discard it.

Dereplicating, denoising, and proveniencing

Proveniencing is simple: you just look through the list of unique sequences (i.e., the dereplicated reads) and the list of all reads, counting up how many times each sequence appears in each sample. Dereplication has one practical question associated with it:

- *How many times does a sequence have to appear for me to believe it?* In most data sets, there is a "long tail" of sequences: there are a many sequences that have a small number of counts. I often use a dumb way to decide which sequences are "real": I just drop the sequences that only appear once in the entire data set.

My simple choice is a hack that does some simple denoising and reduces the size of the resulting data. More intelligent denoising uses a model to infer which reads in the dereplicated set are "real" and which ones are due to sequencing error. There is a whole literature about denoising, so I will just mention *DADA*, which seems to be the best algorithm available.[13]

how to compute this.

[13] In case you didn't get the pun, Dada is the name of an an art movement about, in part, creating irrational, chaotic art.

Chimera removal (or "slaying")

As mentioned early on, PCR can produce chimeric sequences.[14] Depending on your choices about OTU calling, you may want to remove chimeras after dereplicating. Chimera removal checks to see which of your dereplicated sequences can be made by joining the first part of one sequence (the "head") with the last part of another sequence (the "tail").

Chimera removal methods come in two main flavors: *reference-based* and *de novo*. In reference-based methods, you look for the head and tail sequences in some database. Popular databses include Greengenes, SILVA, the Broad Institute's ChimeraSlayer (or "Gold") database, and RDP's "Gold" database.

In *de novo* methods, you ask which of your sequences have could be generated by combining other (typically more abundant) sequences from that same dataset. In the past, this was a computationally-expensive undertaking, but there are ever-improving methods, notably, the UPARSE algorithm, which is now `usearch`'s standard way of simultaneously calling *de novo* OTUs and doing *de novo* chimera detection.

OTU-calling theory

Operational taxonomic units (OTUs) are often the fundamental unit used in 16S data analysis. In most data processing pipelines, OTUs and their abundances in the samples are the output.

OTU-calling methods are (to my eyes) surprisingly diverse, and the choice of method can have a huge impact on the the results of your analysis. Any OTU-based information, plot, or analysis must be interpreted in the context of how those OTUs were called.

A pet peeve of mine is when someone asks "how many OTUs" were in some sample. That number, on its own, means very little. It matters how the OTUs were called. Asking "how many OTUs" is like asking how many kinds of board games there are. The answer depends on how you define "kinds".

An abridged history of the OTU

This is my short and probably incorrect history of the OTU concept.

In the 1980s, Carl Woese showed that the 16S gene could be used as a molecular clock. Using 16S data, he re-drew the tree of life, breaking up the older Monera[15]

[14]The Chimera was a monster in Greek mythology. It had the head of a lion and the tail of a snake. It was slain by the hero Bellerophon.

[15]I'm shocked that, attending public high school in the early 2000s, we were *still* taught about Monera and Protista rather than about Bacteria, Archaea, and Eukaryotes.

into Bacteria and Archaea, showing that Eukaryotes and Archaea are closer cousins than are Archaea and Bacteria. The 16S gene was therefore a promising practical candidate for distinguishing bacterial *species*.[16]

The species concept is easy to define for sexual macroorganisms: two living things of opposite sex are in the same species if they can produce fertile offspring together. Bacteria don't have sex, but they do perform homologous recombination. Homologous recombination requires some sequence similarity, so it came about that a common definition of a bacterial species was all those strains whose isolated DNA was 70% DNA-DNA-hybridization similar.[17]

In the 1990s, people sequenced the 16S genes of the strains grouped into species by the hybridization assay. It emerged as a rule of thumb that two bacteria were the same species if their 16S genes had 97% nucleotide identity.

Because of this history, a lot of discussion around OTUs involves finding 97% clusters, and some people will take "OTU" to mean "97% clusters". I try here to be a little more open-minded: I say an OTU is whatever thing comes out of your method of combining unique 16S sequences into some taxonomically-motivated unit that you think is meaningful for your problem.[18]

Why call OTUs?

Historically, people called OTUs because for a few reasons.

Some reasons were practical. OTU calling tends to involve some combination of data reduction and denoising. For some people, the data reduction was really important. Dereplication can give you hundreds of thousands of unique sequences, which can be intellectually or computationally overwhelming. One might also hope that the denoising aspect of some OTU calling improves the quality of the data.

Some reasons are philosophical[19] or analytical. If you want to study bacterial species and are a firm believer in the idea that a 97% cluster is the best approximation of a species, then you'd want to organize your data into those approximate-species and go from there. More generally, you'll want to organize

[16]In 1991, PCR-amplified portions of the 16S gene were used to identify known species. The paper has a prescient final sentence: "While this [i.e., PCR] should not be a routine substitute for growing bacteria, picking individual colonies, and confirming their phenotypic and biochemical identities, it will enable experiments to be performed that were not previously possible." (Weisburg *et al.*, *J Bacteriol* **173** [1991])

[17]There is still a large debate about the microbial species "concept".

[18]This nomenclature is not universally accepted. For example, some people use "phylotype" to mean a group of sequences that was grouped together because of their similarity to a database sequence and reserve "OTU" for a group of sequences that was made by comparing sequences in a dataset against one another. This also means that I call oligotyping (doi:10.1111/2041-210X.12114), which is breaks your OTUs down into finer units, just another kind of OTU-calling.

[19]There is a fun review (jstor:10.1086/506237) of the different ways ecological units are viewed from ontological and functional perspectives.

your sequences into some operational unit (i.e., OTU) that works well with the kind of analysis you want to do. If you're interested in broad changes in community composition, you might want to call OTUs that are your best approximations of phyla. If you're interested in what individual organisms are doing, you'll probably want to do very little (if any) grouping of sequences into OTUs, since the unique sequences are, in a sense, the best information you have about those organisms.

Regardless of how you call OTUs, I think you should call them in a way that doesn't throw away information that could be useful or interesting. Only throw away information that you are sure you won't find interesting for *any* downstream analysis. Clustering your sequences into a small number of OTUs can make it easier to think about your data, but beware: you want those clusters to be meaningful. You want them to be the called in the way that makes them most useful for answering the question you want to answer.

OTU calling is not the same as lineage assignment

Calling OTUs means assigning your unique 16S sequences to OTUs. Often, each OTU has a sequence associated with it. If an OTU's sequence is the same as one of its members, that member is called the *representative sequence*.

Often the OTU sequence itself is not very interesting. It's just a string of letters. We'd rather know "who" that sequence is. A common way to get this information is to assign *lineages* (or "taxonomies") to each OTU. A lineage is usually an assignment of that sequence to the taxonomic *ranks:* kingdom (or "domain"), phylum, class, order, family, genus, and species.[20]

Confusingly, in some cases, OTUs are in fact called using lineage assignments. It's useful to keep these two concepts separate. For example, it's common to call OTUs in some way, then assign lineages to OTUs, then do a second round of OTU calling in which you merge OTUs that have the same lineage. Now the OTUs are labelled by the lineages. This is how the ubiquitous taxa plots are made.

Common and uncommon OTU-calling methods

This is a survey of OTU-calling methods that are out there in the literature. This list is not exhaustive. The methods are listed are in the order I thought was easiest to explain.

[20]Weird stuff can happen here: there are other ranks like subclass, and sometimes a sequence could, say, get assigned to a genus but not a class. (This is the difference between the two types of RDP classifier output: `allrank` and `fixrank`.)

Dereplication

It may sound a little crazy, but simple dereplication is a kind of OTU calling: every unique sequence is its own OTU. Sometimes this approach is called "100% identity OTUs" to emphasize that all sequences in an OTU are 100% similar, that is, that there is only one sequence in each OTU.

The advantage of dereplication is that it's quick and conceptually straightforward. You need not wrangle over whether the OTU calling method has introduced any weird bias into your data since, roughly speaking, dereplicated sequence *are* your data.

***De novo* clustering**

As the name suggests, *de novo* clustering means making your own OTUs from scratch. There is an enormous diversity of *de novo* clustering methods. Some importants ones to know are UCLUST and its newer cousin UPARSE, which are implemented in the software program `usearch`. UCLUST may be the most popular *de novo* clustering algorithm. It is popular because it's relatively fast and, more importantly, it's part of the QIIME software pipeline (which will come up later). UPARSE is probably better and will probably displace UCLUST.

Briefly, these algorithms try to identify a set of OTUs that are at some distance from one another. (As you might guess, 97% OTUs are popular.) In some cases, the OTU's representative sequence will be the sequence of its most abundant member; in other cases, the OTU's representative sequence is some mish-mash of its member sequences.

De novo clustering suffers from some insidious and very serious disadvantages. First, *de novo* methods are more computationally expensive than other methods. Second, it is becoming increasingly clear that many methods produce *de novo* OTUs that are not *stable*, meaning that small changes in the sequence data you feed into the algorithm can lead to large changes in the number of OTUs, the OTUs' representative sequences, and the assignment of reads to OTUs. Third, it is difficult to incorporate new data into a dataset that has been processed into *de novo* OTUs. It usually requires calling OTUs all over again. It's also difficult to compare *de novo* OTUs across datasets: you and I might have lots of the same sequences but our OTUs might differ.

The principle advantage of *de novo* clustering is that it won't throw out abundant sequences from your data. Why would that happen? Read on!

Reference-based methods

In reference-based OTU calling, the OTUs are specified ahead of time. Usually these OTUs are in a database like Greengenes, which was made by calling *de* *novo* OTUs on some large set of data. Greengenes is such a popular database that sometimes people use "OTU" to mean "the 97% OTUs in Greengenes".

The principle advantages of reference-based calling are:

- *Stability.* Similar inputs should produce similar outputs, since you're just comparing to a fixed reference.
- *Comparability.* If you and I called our OTUs using the same reference, it's easy for us to check if we have similar sequences in our datasets. We can even combine our datasets in a snap.
- *Computational cheapness.* Unlike *de novo* OTU calling, reference-based methods only need to hold one sequence in memory at a time. This makes them cheap (in terms of memory) and embarrassingly parallelizable[21].
- *Chimeras need not be slain.* If you're only keeping sequences that align to some database, which has hopefully been pre-screened for chimeras, then you don't need to worry about them yourself.

The major weakness of reference-based methods can be dastardly and insidious: if a sequence in your query dataset doesn't match a sequence in the database, what do you do? Frighteningly, many methods just throw it out without telling you. If you work in the human gut microbiome, this might not bother you, since the gut is the best-studied ecosystem and databases like Greengenes have heaps of gut data baked into them. If you work in environmental microbiology or even in mice, however, many of your sequences might not hit Greengenes.

Reference-based methods also suffer a converse problem: what if your sequence is an equally good match to more than one database entry? This can happen in amplicon sequencing: the Greengenes OTUs are the entire 16S gene (about 1400 bases), but you only have a little chunk of it (say, 250 bases). The Greengenes OTUs are, say, 97% similar (i.e., 3% dissimilar) across the *entire gene*, but they might be identical over the stretch that aligns to your little chunk.

USEARCH, a popular algorithm[22] for matching sequences to a database (and one of QIIME's tool for reference-based OTU calling). Because USEARCH is so popular and because it can have unexpected effects, I will devote some space in the next chapter to discussing its workings.

Open-reference calling

The process I described about —just throwing out non-matching sequences— is called *closed-reference* calling. If you're interested in those non-matching sequences, you could gather them up and group then into *de novo* clusters, then combine your reference-based OTUs with your *de novo* OTUs. This mix of reference-based and *de novo* is named *open-reference* calling.

[21] I didn't make that up; it's a real computer science term.

[22] Confusingly, USEARCH is the name of an alignment algorithm and `usearch` is the name of the program that does USEARCH, UPARSE, and other stuff.

Lineage-based assignments

Reference OTUs tend to have unsatisfying names. For example, the Greengenes OTUs are labeled with numbers. It's common (but not necessarily good) practice to do a second round of OTU calling: the new OTUs are made by combining the old OTUs that have the same lineage.

How does this work? Greengenes associates a taxonomy with each of its OTUs. This is relatively easy for sequences that came from isolates: give that OTU the classification you would have given the isolate. It gets more tricky for OTUs that aren't just taken from isolates: it requires some sort of phylogenetic inference. This means that you construct a tree of all your sequences, figure out where the taxonomic clades are, and assign taxonomies to OTUs based on what you found.

Someone at Greengenes has done all the hard work of assigning taxonomies to their reference OTUs, but I want to make it clear that this is not a foolproof process. All lineage assignments should be treated with a healthy skepticism.

The most popular alternative to Greengenes for lineage assignments is the Ribosomal Database Project (RDP) Naive Bayesian Classifier. Rather than comparing a sequence to existing OTUs, the RDP classifier breaks up the sequence into *k*-mers (all subsequences of the original sequence that have length *k*) and compares the *k*-mer content of that sequence to a big database that knows how *k*-mer content relates to taxonomy. The practical advantage to this is that RDP gives *confidences*[23] to each level of the taxonomic assignment. For example, a sequence might definitely be from some phylum (99%), but it might be difficult to specify its class (80%) and nearly impossible to identify its order (30%). In contrast, using the Greengenes approach, the same sequence might happen to hit an OTU that is classified all the way down to the species, and you would mistakenly think that your sequence had a lot of taxonomic information in it.

Distribution-based methods

All the algorithms mentioned only look at the list of unique sequences; they don't take any notice of how those sequences are distributed among the samples.[24] Some Alm Lab work has shown that you get more accurate OTUs (i.e., OTUs that better reflect the composition of a known, mock community) if you take the sequence provenances into account. If an abundant sequence and a sequence-similar, rare sequence are distributed the same way across samples, the rare sequence is probably sequencing error and should be put in the same OTU

[23] I'm excited for when Robert Edgar, maker of `usearch`, will publish a paper about UTAX, which he says will be faster than RDP and will give "informative" confidence values.

[24] This isn't stricly true: some *de novo* methods will take into account the abundance of a sequence in the entire dataset and will treat more abundant sequences differently from less abundant ones. As far as I know, however, the algorithm described here is the only one that accounts for differences in the distribution of each sequence across samples.

with the abundant one. Conversely, if two very similar sequence are never found together, they probably represent ecologically-distinguishable bacteria, so they should be kept in separate OTUs. This approach is confusingly called *distribution-based* OTU calling or, less confusingly, *ecologically-based* OTU calling.

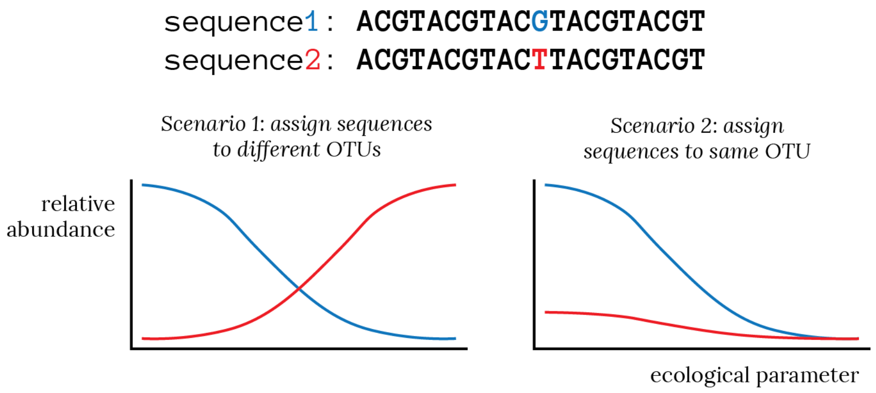

Figure 6: Distribution-based OTU calling separates similar sequences if they are distributed differently across samples. Adapted from Preheim *et al.*

Practice

As you can see, there are a lot of in's, out's, and what-have-you's (IOWHYs) to processing 16S data. Rather than giving you the fish of some monolithic pipeline, I'd rather try to teach you as much about fish as you'll find useful. There are plenty of websites that can give you fish but not many that can introduce you to the IOWHYs.

In the rest of this document, I'll assume you're in a Unix/Linux/Mac world. If you're on Windows, you'll essentially need to get to the Unix world with a tool like Cygwin. I didn't make up those rules, but that's how it is: all the serious bioinformatics are done in the non-Windows world.

File formats

The two important file formats are the above-mentioned *fastq* and the *fasta*. Fastq is an Illumina-specific raw data format. Fasta is the industry-standard way to display processed sequence data.

Fastq format

Fastq files usually have the extension `.fastq` or `.fq`. Fastq files are human-readable. They are made up of *entries* that each correspond to a single read. Every entry is four lines.[25] A well-formed fastq file has a number of lines that is a multiple of four. You can see how many lines are in a file using the terminal command `wc -l foo.fq`. Divide that by four to get the number of entries. The lines in the entry are:

1. The *header* (or "at") line
2. The *sequence* line
3. The *plus* line
4. The *quality* line.

The header line gives an identifier for the read. The line must begin with the at-character `@`. The format of the rest of the line depends on the Illumina software version, but in general it gives information about the read: the name of the instrument it was sequenced on, the flowcell lane, the position of the read on the cell, and whether it is a forward or reverse read. In some version of Illumina, the barcode read is in the read's header. You can put whatever information you want in this line. For example, fastqs on NCBI might have `length=1234` somewhere after all the Illumina stuff.

Like the name suggests, the sequence line is a series of letters encoding the sequencing data, one character per nucleotide. Note that the letter might not all be `ACGT`. Other letters are allowed to say that it might be any of a group of bases. For example, `R` means purine (`A` or `G`). `N` means "no idea, any base possible".[26]

I call line 3 the "plus" line because it must begin with the plus-character `+`. Bizarrely, the rest of the line can be anything. Most people set it to the same thing as the at-line (which is a waste of space) or nothing.[27]

The quality line gives information about the quality of the base calls shown in the sequence line. Each character gives information about the quality of one base call. Confusingly, the encoding has changed in overlapping and sometimes non-redundant ways.[28] In the newest Illumina format, the encoding goes, from

[25]Confusingly, the definition of a "line" on Windows is different from Unix/Linux/Mac. If you ever pass data between these two computer systems, make sure that have adjusted the line endings using `dos2unix` (or `unix2dos`). You can also look up how to do this with `tr`, etc.

[26]These other options are the IUPAC nucleotide abbreviations. There is a one-letter code for every possible combination of the four nucleotides.

[27]The original fastq specification (doi:10.1093/nar/gkp1137) allowed the sequence and quality information to run over multiple lines (like in the fasta format). This led to a lot of confusion, since `+` and `@` appear in some quality encodings. So it's now recommended to *not* spread the sequence and quality information over multiple lines. Thus, the plus line is there for backward compatibility.

[28]Technically, these different encodings are named by their ASCII "offsets". ASCII is a system that associates individual characters with integers. The system I show here has an ASCII offset of 33: the character `I` has an ASCII value of 73, and the offset means you subtract the offset 33 from the ASCII value 73 to get the quality score 40. The other common offset is

low quality to high quality:

```
!#$%&'()*+,-./0123456789:;<=>?@ABCDEFGHI
```

The letter `I`, the highest-quality mark, means that there is a $10^{-4.0}$ probability that this base is wrong (i.e., a 99.99% chance that it is correct). `H`, the second-highest quality, means that there is a $10^{-3.9}$ chance; `G` means $10^{-3.8}$, and so on.[29] The exclamation point is special: it means a quality of zero, i.e., that the sequencer has no idea what that base is.[30]

Here's an example fastq entry. The sequence and quality lines are too long to fit on the page, so I cut out some letters in the middle and put those dots instead.

Notice how the quality of the read decreases (all the way down to quality 2, encoded by the hash symbol `#`) toward the end of the read. The barcode read is `CCGACA`. The `/1` at the end of the first line shows that this is a forward read.

```
@MISEQ578:1:1101:15129:1752#CCGACA/1
TATGGTGCCAGCCGCCGCGGTA...GCGAAGGCGGCTCACTGGCTCGATACTGACGCTGAG
+
>1>11B11B11>A1AA0A0OE/...FF@@<@FF@@@@FFFFFFEFF@;FE@FF/9-AAB##
```

Some things to note about this read:

- The first line gives information about the machine used, where the read was on the flow cell, etc. `CCGACA` was the barcode read, and the final `/1` means it was a forward read.
- The plus line has been left blank.
- The quality of the read decreases from `>`, encoding quality 29 ($10^{-2.9} =$ 0.1% probability of error), all the way down to `#`, or quality 2 ($10^{-0.2} = 63\%$ probability of error).

64: in that case, the character `I` encodes quality 9. Illumina used a 64-offset for a while, but newer machines use a 33-offset. You can tell the difference getting familiar with the characters used and just looking by eye, by running your pipeline and seeing if it breaks, or by using a tool like `usearch -fastq_chars`.

[29] This conversion between quality Q (the integer) and the probability p of incorrect calling is the "Phred" or "Sanger" system: $Q = -10 \log_{10} p$, or equivalently, $p = 10^{-Q/10}$. There is at least one other way of converting between Q and p, the "Solexa" system.

[30] Even more confusingly, in some versions of Illumina, the very low ASCII scores were used to mean special things. In the older, 64-offset Illumina encoding, quality 3 (character `C`) was the lowest possible, and the quality 2 character (character `B`) was instead used to show that the Illumina software had done its own internal quality trimming and had decided that, starting with the first `B`, that the rest of the sequence was "bad". To my relief, this weird system is not used in the newer Illumina (versions 1.8 and above). I personally think the sequencer should give you *just* raw data, not some combination or raw data and partially-processed data.

Fasta format

Fasta files usually have the extension `.fasta`, `.fa`, or `.fna`.[31] Fasta files are human-readable. They are made up of *entries* that each correspond to a single sequence. Each entry has this format:

1. A *header* line.
2. One or more *sequence* lines.

The *header* line must start with the greater-than symbol `>`. This signals the start of a new entry. The stuff after `>` is usually some kind of ID that names this sequence. There might be other information there too.

In a fastq file, each linebreak was meaningful. In a fasta file, only the linebreak at the end of the header line is meaningful. Linebreaks in the following sequence lines are all ignored. The sequence for this entry just keeps on going until you hit another `>`.[32] The early recommendation was to wrap all lines at 80 characters to make them easy to read on old-school terminals. Many people and software tools conform to this recommendation; others make each entry just two lines, one header and one sequence. Both are allowed.

Fasta files provide less information than fastq files, so normally you'll move to a fasta format when you've decided you aren't going to look at the quality data again, i.e., after quality filtering.

Here's an example fasta entry in the traditional format (i.e., each line no more than 80 characters and the sequence is spread over multiple lines).

```
>seq1;size=1409414;
TACGGAGGATCCGAGCGTTATCCGGATTTATTGGGTTTAAAGGGAGCGTAGGCGGGTTGT
TAAGTCAGTTGTGAAAGTTTGCGGCTCAACCGTAAAATTGCAGTTGATACTGGCATCCTT
GAGTACAGTAGAGGTAGGCGGAATTCGTGGTGTAGCGGTGAAATGCTTAGATATCACGAA
GAACTCCGATTGCGAAGGCAGCCTGCTGGACTGTAACTGACGCTGATGCTCGAAAGTGTG
```

Note that the header line includes an identifier for this read (`seq1`) as well as some other data (the read appeared 1.4 million times in this data).

Other formats

I've been talking about Illumina, but the other important data type comes from 454 sequencing. Raw 454 data comes in *sff* format. Unlike fastq files, sff files are not human-readable. Luckily, you can convert sff files to fastq files pretty

[31] The last `a` in `fasta` stands for "all", meaning nucleotide or amino acid or whatever. The `n` in `fna` stands for "nucleotide".

[32] Technically, a new record should begin only when you see a `>` at the start of a new line. However, not all fasta parsers are this careful. The reasoning is that you usually don't have `>` characters in the sequence data, so they get lazy and say "any time I see `>` it means there's a new entry".

easily.[33] Unluckily, 454 sequence data has its own in's and out's that are beyond the scope of this document. Caveat sequentor.[34]

Existing tools

Everything in this document up to this point will help you be critical of an analysis pipeline. This is true whether you use an existing pipeline, work with someone who has a pipeline, or make your own pipeline. The more you want to be involved in looking at and working with your own data, the more computer skills you'll need. You'll also need more of your own skills if you want to do more creative analysis or be more sure of exactly what's being done with your data.[35]

QIIME

The great and mighty QIIME is a big part of the 16S data processing landscape. QIIME was one of the two big pipelines used to analyze the HMP data. I see the taxa plots produced by QIIME in many papers.

usearch

The great and mighty `usearch` underlies a lot of 16S data processing. A lot of QIIME is built on the back of `usearch`. `usearch` is a single program that has an ever-growing number of functionalities (including, the algorithms USEARCH, UCLUST, and UPARSE).

Because USEARCH is so widely used, I will take a moment to dive a little more deeply into its workings. I have seen a lot of confusion arise because of USEARCH, mostly because users (like me!) expected USEARCH to do one thing when, in fact, it does something else. The caveats I'm going to lay out apply if you're using USEARCH via `usearch`, QIIME, or another pipeline.

Critically, USEARCH is a *heuristic* algorithm. This means that it applies some shortcuts to achieve faster speed (i.e., orders of magnitude faster than simple BLAST). In the paper, Mr. Edgar explains that USEARCH gets its speed-up from three places.

[33]454 sequencing is a fundamentally different technology from Illumina, so converting the raw sff to fastq involves some inference and processing. sff files are not human-readable, so you'll probably want to use a ready-made tool like `biopython`'s `convert` command or a tool like `sff2fastq`.

[34]That's a joke for Latin scholars. It's meant to mean "let the one processing the data beware".

[35]This all being said, I don't fault anyone for using tools that actually *work*. Some of the theoretically "nicer" tools aren't coded in a way that makes them easy to use, robust, or capable of working on big datasets. Many theoretically nice tools don't have good documentation, which makes them essentially unusable.

1. Search for a match according to a decreasing expected sequence similarity.
2. Use heuristics to speed up the sequence alignments.
3. Apply stopping criteria. The default is to stop after a single hit that meets the accepting criteria.

The first rule means that your query sequence will be compared to database sequences according to some order; the third rule means that the first of these database sequences that is a sufficiently good match to your query will be delivered as the result. For example, if you are assigning your OTUs by making a search to the 97% OTUs in Greengenes, then the first database sequence that is at least 97% similar to your query will be considered its parent OTU.

This would be of no concern if the comparisons were performed in the order of decreasing sequence similarity. Then the first hit would be the best one. This, however, puts the cart before the horse: you need to somehow *guess* which sequence will be the best match before actually performing the alignment.

USEARCH guesses the sequence similarity between two sequences using the U value (which is the U in USEARCH). U is the number of unique *words* shared by two sequences. You get these words by looking for length w (default is 8) subsequences starting at positions spaced μ apart. For example, if $\mu = 1$, then your words are all the w-mers in the sequence. If $\mu = w$, then your words are the first w nucleotides, the next w nucleotides, and so forth. Conveniently, sensible values for μ are inferred using tables of optimal choices derived from running USEARCH on databases of sequences using different values of the identity threshold.

This heuristic selection of database entries for comparison can lead to some quirky results. When working on a mouse microbiome project, I found that many sequences in my dataset were very similar (say, one nucleotide different in a 250 nucleotide amplicon) but ended up in different 97% OTUs. I've heard stories of people who discovered this quirk when they called OTUs *de novo* and using reference-based calling. They expected that since their *de novo* and reference-based OTUs were both 97%, they should be about the same "size", except that reference-based OTU calling would miss some of the OTUs that *de novo* calling would catch. In fact, this approach usually leads to *more* OTUs in the reference-based calling.

In light of these weird effects, I caution you to carefully read the `usearch` documentation to get a rough sense of what it's doing. I got weird results for some years before I started to dig into what `usearch` was actually doing. Don't expect it to do something that it doesn't do!

Overall, the nice thing about `usearch` is that it's built by someone who's very concerned about making a fast, optimized tool. The potentially nice thing about `usearch` is that it's pretty low-level: if you want to do some medium-lifting (like primer removal and demultiplexing) but not heavy-lifting (like merging, quality filtering, clustering, or sequence alignment), then `usearch` is for you.

A potentially bad thing about `usearch` is that it's made by a single person and is closed source. It also comes with a funny caveat: newer versions of `usearch` come in two flavors, free and wildly expensive. This means that mere mortals (i.e., non-Broadies) must be content to use the free version, which will only process 2 Gb of data at a time.[36] If you need to call OTUs *de novo* from an enormous data set, then you're sunk; if you merely want to merge, quality filter, or do alignments, then it's merely annoying.[37] QIIME comes with an older version of `usearch` that doesn't have as many features but which also doesn't have the memory restriction.

mothur

Similar to QIIME, mothur aims to be "a single piece of open-source, expandable software to fill the bioinformatics needs of the microbial ecology community". Their website says mothur is "currently the most cited bioinformatics tool for analyzing 16S rRNA gene sequences." Mothur was the other big pipeline used to analyze the HMP data.

Your own tools

I've kept yammering about doing things on your own, being critical of existing tools, bla bla. How are you supposed to do that?

The eyes in your head

The best tool in your belt is a curious attitude. Be critical of your data at every step in the pipeline. Does it look the way you expect? How can you check? It's wise to *look* at your data, and look at it a lot, before moving on to the next step. It will also save you some headaches later on by knowing what happened in the middle.

The sweat of your brow

What does it mean to "look" at your data? The answer is that you'll need some computational skills. By "computational" I mean "command-line". This means familiarity with the basic tools of the Unix terminal (`cat`, `cd`, `cp`, `ls`, `mkdir`, `mv`,

[36]The closed-source and the wildly-expensive problems might be solved by an open-source implementation of USEARCH and the other algorithms, as is being developed under the creative name VSEARCH, where I think the V stands for "versatile".

[37]Most other things you do with 16S data are parallelizable: you don't need to hold the entire dataset in memory at once, you just need little parts of it. For example, if you want to use `usearch` to merge a big paired-end dataset, you can split the forward and reverse read files into smaller chunks and merge each pair of chunks one-by-one.

`rm`, `head`, `less`, `sed` or `awk`, `wc`, `grep`, and `vi` or `emacs`) and the ability to use some programming language.

You *can* open OTU tables in Excel or whatever, but if you try to use the more familiar office-sytle software to do the heavy lifting here, you will be disappointed. Opening a 5 Gb fastq in Notepad or TextEdit won't be fun. Maybe one day we'll have sexy drag-and-drop, hologram-style data processing for 16S, but for the foreseeable future it's going to look like the scene in *Jurassic Park* where Samuel L. Jackson is hunched over a computer muttering "Access main program... Access main security... Access main program grid... Please! Goddamn it! Hate this hacker crap!"

A programming language

Those simple command-line tools will get you pretty far, but as some point you'll want to ask a question of your data or do something to your data that isn't exactly standard. At that point, you'll need to be able to write a program.

Here are some criteria to help you decide what language is best for you:

- *Support.* You'll have a better time using a language that has good documentation, a large global community of users (who produce informal documentation in places like StackOverflow), and a local community of users, i.e., the person down the hall who can help you figure out what that syntax error means.
- *Bioinformatics packages.* It's nice to not re-invent the wheel. Some programming languages have mature, extensive bioinformatics toolkits.
- *Appropriateness for your purpose.* Compiled programming languages run fast but are slow to develop; scripting languages run slow but are fast to develop. If you're going to be crunching huge datasets that will take days to process, think about a compiled language designed for parallelization. If you're just working on a few small datasets on your own computer, think about a scripting language that's easier to use.

My language of choice for 16S data processing is Python. It's a popular language with great documentation. The people around me use it. It has a good bioinformatics package (`biopython`). It's relatively slow but I don't care because I spend much more time programming that I do actually running data.

I think Perl used to be the most popular bioinformatics programming language, and I think it's being displaced by Python. `R` is a great language for analyzing making plots and doing statistics on the resulting OTU tables, but it's not super-handy for working with raw sequences. Some people will swear by working in C or C++, but I think you should wait until you really need a 10-fold speedup before going that deep. Apparently Matlab has some bioinformatics capability. Julia might one day be a cool option.

Some early challenges

Rolling your own

I hear a lot of people say they want to build computational skills. The way you do that is by learning more and building stuff. Personal anecdote: I worked for a year in a lab that used Fortran to do most of its computation. I had previously done most of my computational work in Mathematica (I was a physics major). After a few months, I took off one full week of work to read and do all the exercises in Mark Lutz's *Learning Python*. It was one of the best investments I've ever made. You learn more by putting in the time. It will pay off.

Another personal anecdote: after working with QIIME and then with another grad student's codebase, I finally decided to write my own 16S processing software from scratch. I learned most of what's in this document as I learned what I had to do to write that software. If starting with a real big kid script that does real data processing sounds like too much, try working on the problems at Rosalind. They start out easy and get progressively harder. It's a nice way to do something for a half hour and get the instant gratification of a gold star.

Looking at real data

If you're reading this packet, you probably have your own data set in hand. If not, two great places to get some data to play with are the HMP project's raw sequences or the data generated in Caporaso *et al.* from MG-RAST.

Second opinions

I don't know of another resource designed to fill the same purpose as this document. That's part of why I wrote it. That being said, there are other places you can go to get something similar (and/or meaningfully different) from what I've spouted here.

`usearch` documentation

The documentation for `usearch` is extensive and fascinating. The documentation for an indivudal `usearch` function is often linked to a separate page explaining the theory behind that step. I learned a lot from reading that documentation.

Pipeline tutorials

Maybe you read this hoping to get a step-by-step about how to actually process data. The QIIME and mothur website both have a lot of documentation about how to use their tools. They are good places to go for step-by-step instructions. There are also some broader lessons about how 16S processing works in general; I did a lot of my early learning about 16S data from reading the QIIME documentation.