Advanced Microbiome Analysis 2025

Contents

Part I

Introduction

Chapter 1

Welcome

Welcome to CBW's Advanced Microbiome Analysis 2025 Workshop!

1.1 AWS/UNIX primer

You can find the schedule and materials for the AWS/UNIX primer here.

1.2 Course Schedule

Time (PST)	Day 1	Time (PST)	Day 2
8:30	Arrivals & Check-in	8:30	Arrivals
9:00	Welcome (Coordinator)	9:00	Module 3 Lecture
9:30	Module 1 Lecture		
10:30	Break (30min)	10:00	Break (30min)
11:00	Module 1 Lab	10:30	Module 3 Lab
13:00	Lunch (1h)	12:30	Class photo + Lunch (1h)
14:00	Module 2 Lecture	13:30	Module 4 Lecture
15:00	Break (30min)	14:30	Break (30min)
15:30	Module 2 Lab	15:00	Module 4 Lab
		17:00	Survey and Closing Remarks
17:30	Finished	17:30	Finished

1.3 Meet Your Faculty

Coming soon!

Part II

Modules

Chapter 2

Module 1 Lab: Introduction to metagenomics and read-based profiling

This tutorial is part of the 2025 CBW Advanced Microbiome Analysis (held in Vancouver, BC, May 29-30). It is based on a version written for the 2024 CBW Advanced Microbiome Analysis workshop by Ben Fisher.

Author: Robyn Wright

2.1 Overview

The goal of this tutorial is to familiarise students with processes by which we analyse metagenomic data and classify taxa within our samples. Shotgun Metagenomic Sequencing, sometimes called MGS or WGS, is capable of capturing any DNA extracted from a given sample (however, this does not necessarily mean it captures ALL of the DNA). With MGS reads, we must consider that there could be significant host contamination, so - depending on our sample type - we must filter against host sequences in our pipeline. We will then classify the taxa in our samples using two popular approaches - an all reads approach (Kraken2 + Bracken) and a marker-gene approach (MetaPhlAn). In all steps of bioinformatics, there are many tools that can work to produce similar results, and there is never a one-size-fits-all solution. It is up to us to learn about the options that are available and choose which one is most appropriate for our application - people often have different opinions, so sometimes we need to educate ourselves, make a decision, and be able to justify this decision to others. This lab is therefore a foray into some popular tools and processes, and how to appropriately use them for our analyses.

Throughout this module, there are some questions aimed to help your understanding of some of the key concepts. You'll find the answers at the bottom of this page, but no one will be marking them. *You may wish to keep them open in another tab.*

2.2 1. Initial setup and pre-processing of metagenomic reads

2.2.1 Log into your instance

If you weren't in the beginner microbiome analysis workshop and need a refresher, check the notes in Beginner Module 1.

2.2.2 Working on the Command Line

Make a new folder in your workspace to work from:

```
cd workspace
mkdir amb_module1
cd amb_module1
```

If you weren't in the beginner workshop, feel free to take a look around the instance using the `ls`, `pwd` and `cd` commands, but make sure you come back to `~/workspace/amb_module1` before the next step.

Make a link to the files to your working directory.

```
ln -s ~/CourseData/MIC_data/AMB_data/raw_data_subsampled_subset/ raw_data
cp ~/CourseData/MIC_data/AMB_data/metagenome_metadata_subset.txt metagenome_metadata.t
```

2.2.3 About the samples

These samples are from a study that investigated gut microbiome functional capacity of chicken ceca. You can read more in the original study here: - Week-Old Chicks with High Bacteroides Abundance Have Increased Short-Chain Fatty Acids and Reduced Markers of Gut Inflammation

Or in the additional analysis done by John's lab here: - Metabolic modeling of microbial communities in the chicken ceca reveals a landscape of competition and co-operation

2.2.4 A Crash Course in GNU Parallel

First, activate our conda environment for this tutorial:

```
conda activate amb_module1
```

Sometimes in bioinformatics, the number of tasks you have to complete can get VERY large (e.g. when we have thousands of samples). Fortunately, there are several tools that can help us with this. One such tool is GNU Parallel. This tool can simplify the way in which we approach large tasks, and as the name suggests, it can iterate though many tasks in *parallel*, i.e. at the same time. If you aren't familiar with using `top` or `htop` to look at the number of processes that you have available to you, take a look at this section of the beginner workshop.

We can use a simple command to demonstrate how to use `parallel`:

```
parallel 'echo {}' ::: a b c
```

With the command above, the program contained within the quotation marks `' '` is `echo`. This program is run 3 times, as there are 3 inputs listed after the `:::` characters. What happens if there are multiple lists of inputs? Try the following:

```
parallel 'echo {}' ::: a b c ::: 1 2 3
```

Here, we have demonstrated how `parallel` treats multiple inputs. It uses all combinations of one of each from `a b c` and `1 2 3`. But, what if we wanted to use 2 inputs that were sorted in a specific order? This is where the `--link` flag becomes particularly useful. Try the following:

```
parallel --link 'echo {}' ::: a b c ::: 1 2 3
```

In this case, the inputs are "linked", such that only one of each is used. If the lists are different lengths, `parallel` will go back to the beginning of the shortest list and continue to use it until the longest list is completed. You do not have to run the following command, as the output is provided to demonstrate this.

```
parallel --link 'echo {}' ::: light dark ::: red blue green
```

Notice how `light` appears a second time (on the third line of the output) to satisfy the length of the second list.

Another useful feature is specifying *which* inputs we give parallel are to go *where*. This can be done intuitively by using multiple brackets `{ }` containing numbers corresponding to the list we are interested in. Again, you do not have to run the following command, as the output is provided to demonstrate this.

```
parallel --link 'echo {1} {3}; echo {2} {3}' ::: one red ::: two blue ::: fish
```

Finally, a handy feature is that `parallel` accepts files as inputs. This is done slightly differently than before, as we need to use four colon characters `::::` instead of three. Parallel will then read each line of the file and treat its contents as a list. You can also mix this with the three-colon character lists `:::` you are already familiar with. Using the following code, create a test file and use `parallel` to run the `echo` program:

```
echo -e "A\nB\nC" > test.txt
parallel --link 'echo {2} {1}' :::: test.txt ::: 1 2 3
```

Take a look inside `test.txt` with the `less` command if you like. Remember that you can use `q` to exit the file again.

And with that, you're ready to use parallel for all of your bioinformatic needs! We will continue to use it throughout this tutorial and show some additional features along the way. There is also a cheat-sheet here for quick reference.

2.3 Quality Control

Quality control is an important step in any pipeline. For this tutorial, we will use FastQC to inspect the quality of our samples.

2.3.1 Visualization with FastQC

First, make your desired output directory `fastqc_out`. Then, run FastQC as follows:

```
fastqc -t 4 raw_data/*fastq.gz -o fastqc_out
```

Run MultiQC on the FastQC results:

```
multiqc fastqc_out --filename multiqc.html
```

Remember that you can go to `http://##.uhn-hpc.ca/` (where ## is your personal number) to find all of the files that we're creating.

Copy across the `multiqc.html` file to your computer.

Click on the **.html** files to view the results for each sample. Let's look at the Per base sequence quality tab. This shows a boxplot representing the quality of the base call for each position of the read. In general, due to the inherent

degradation of quality with increased sequence length, the quality scores will trend downward as the read gets longer. However, you may notice that for our samples this is not the case! This is because for the purpose of this tutorial, your raw data has already been trimmed.

More often, per base sequence quality will look like the following. The FastQC documentation provides examples of "good" and "bad" data. These examples are also shown below:

Now, you may have also noticed that most of the samples fail the "Per Base Sequence Content" module of FastQC.

This specific module plots out the proportion of each base position in a file, and raises a warning/error if there are large discrepancies in base proportions. In a given sequence, the lines for each base should run in parallel, indicating that the base calling represents proper nucleotide pairing. Additionally, the A and T lines may appear separate from the G and C lines, which is a consequence of the GC content of the sample. The relative amount of each base reflects the overall amount of the bases in the genome, and should not be imbalanced. When this is not the case, the sequence composition is **biased**. A common cause of this is the use of primers, which throws off our sequence content at the beginning of the read. Fortunately, although the module error stems from this bias, according to the FastQC documentation for this module it will not largely impact our downstream analysis.

The other modules are explained in depth in the FastQC Module Help Documentation

2.3.2 Filtering with KneadData

KneadData is a tool which "wraps" several programs to create a pipeline that can be executed with one command. For this tutorial though, we will use KneadData to filter our reads for contaminant sequences against a human database. KneadData will: * Run Trimmomatic to remove adapter sequences * Run Bowtie2 to remove reads mapping to the reference genome and the PhiX genome (commonly used as a sequencing control)

We don't have paired-end data here, but if we did, it would also: * Check whether both pairs of a read exist * Check whether they both map to the reference genome

Bowtie2 needs a reference genome/index file for its mapping step. There are some pre-made indexes on this page.

MAKE A CHOICE!

You can now choose to use the database that we have already made (quicker option) or you can see the steps involved in building this for yourself (slower option).

Quicker option - use pre-made database

```
ln -s ~/CourseData/MIC_data/amb_module1/bowtie2_db/ .
```

Slower option - build the database yourself

First, download the NCBI RefSeq assembly summary:

```
wget ftp://ftp.ncbi.nlm.nih.gov/genomes/refseq/assembly_summary_refseq.txt
```

Have a look in this file. You'll see that it contains information on all of the genomes in NCBI RefSeq - we often need reference genomes for constructing various kinds of databases, so it's useful to get familiar with what is available on NCBI. You'll see that each assembly/accession has information on the NCBI taxonomy ID, whether it's a complete genome, whether it's a reference or representative genome, and a link to where we can find the associated files. Because the samples we are using are from chickens, we're going to look for the files that are associated with the chicken *Gallus gallus*:

```
grep "Gallus gallus" assembly_summary_refseq.txt
```

The first one has "reference genome" in the first column, so that's what we'll use. Copy the link that starts https://ftp.ncbi..... into a new browser window. You will see a bunch of different file types, so find the one that ends `_genomic.fna.gz`, right click, and click "Copy Link Address". Now use the `wget` command to download this to your instance.

Now we're going to do the same with PhiX. PhiX is added as a sequencing control and is often removed before we're given back our samples, but it's good to make sure. - Use `grep` to search for "phiX174" - Go to the link that you find - Download the `_genomic.fna.gz` file using `wget`

QUESTION!

Question 1: This one isn't really a question, but you can see the code needed for these steps in the answers section if you need to!

Now we'll combine these two files into one:

```
cat GCF_016699485.2_bGalGal1.mat.broiler.GRCg7b_genomic.fna.gz GCF_000819615.1_ViralProj14015_gen
```

And finally, we'll build our Bowtie2 database:

```
mkdir bowtie2_db
bowtie2-build chicken_Gallusgallus_phiX174.fna.gz bowtie2_db/chicken_Gallusgallus_phiX174 --threa
```

This will take about ten minutes to run. If this is taking too long, feel free to stop this command using `ctrl+c` and link to the one that we already made:

```
ln -s ~/CourseData/MIC_data/amb_module1/bowtie2_db/ .
```

2.3.2.1 Back to KneadData

Now we are ready to run KneadData using parallel:

```
parallel -j 1 --eta 'kneaddata -un {1} -o kneaddata_out -db bowtie2_db/chicken_Gallusgallus_phiX1
```

You'll see that after the first sample has finished, you'll get an estimate of the amount of time remaining.

While `kneaddata` is running, consider the following:

Note that if we had paired reads to start with, we'd also have some files containing reads that were unmatched between the forward and reverse files.

We'll also run the following to get counts of how many reads we had in each file after each step:

```
kneaddata_read_count_table --input kneaddata_out --output kneaddata_read_counts.txt
```

QUESTION!

Question 2: Take a look at this file. Are there any surprises? Which of the output files in 'kneaddata_out' will you use for analysis?
Question 3: How many reads are in each sample before and after KneadData?

We'll move the other output files that we're not interested into a new folder:

```
mkdir kneaddata_out/contam_seq
mv kneaddata_out/*_contam*.fastq kneaddata_out/contam_seq
```

2.4 2. Generating Taxonomic Profiles

2.4.1 Annotation with Kraken2/Bracken

Now that we have our reads of interest, we want to understand *what* these reads are. To accomplish this, we use tools which **annotate** the reads based on different methods and databases. There are many tools which are capable of this, with varying degrees of speed and precision. For this tutorial, we will be using Kraken2 for fast exact k-mer matching against a database.

We have also investigated which parameters impact tool performance in this Microbial Genomics paper. One of the most important factors is the contents of the database, which should include as many taxa as possible to avoid the reads being assigned an incorrect taxonomic label. Generally, the bigger and more inclusive database, the better. However, due to the constraints of our AWS cloud instances, we will be using a "PlusPF 8GB" index provided by the Kraken2 developers.

First of all, we want to download and extract the Kraken database:

```
wget https://genome-idx.s3.amazonaws.com/kraken/k2_pluspf_08gb_20250402.tar.gz
mkdir k2_pluspf_08gb
tar -xvf k2_pluspf_08gb_20250402.tar.gz -C k2_pluspf_08gb
```

Then we can clean-up by removing the original file:

```
rm k2_pluspf_08gb_20250402.tar.gz
```

NOTE!

First, you must create the appropriate output directories, or Kraken2 will not write any files. Use the 'mkdir' command to make the directories to match what we're using below. Using 'parallel', we will then run Kraken2 for our concatenated reads.

Note that it's always a good idea to first try out a `--dry-run` of parallel before you run any long jobs. If you're satisfied with what it's going to be running, remove the `--dry-run` flag.

```
parallel -j 1 --eta --dry-run 'kraken2 --db k2_pluspf_08gb --output kraken2_outraw/{/.
```

This process can take some time. While this runs, let's learn about what our command is doing! * We first specify our options for parallel, where: * the `-j 2` option specifies that we want to run two jobs concurrently; * the `--eta` option will count down the jobs are they are completed; * after the program contained in quotation marks, we specify our input files with `:::`, and use a regex to match all of the concatenated, unzipped `.fastq` files. * We then describe how we want `kraken` to run: * by first specifying the location of the database with the `--db` option; * then specifying the output directory for the raw kraken annotated reads; * notice that we use a special form of the brackets here, `{/.}`, this is a special function of `parallel` that will remove both the file path and extension when substituting the input into our `kraken` command. This is useful when files are going into different directories, and when we want to change the extension. * similarly, we also specify the output of our "report" files with the `--report` option; * the `-confidence` option allows us to filter annotations below a certain threshold (a float between 0 and 1) into the unclassified node. We are using `0` because our samples are already subset, however this should generally be higher. See our paper for more information. * and finally, we use the empty brackets `{}` for `parallel` to tell `kraken` what our desired input file is.

As Kraken runs, you should see it printing out a summary of the number of reads within each sample that it was able to classify.

With Kraken2, we have annotated the reads in our sample with taxonomy information. If we want to use this to investigate diversity metrics, we need to find the abundances of taxa in our samples. This is done with Kraken2's companion tool, Bracken (Bayesian Reestimation of Abundance with KrakEN).

Let's run Bracken on our Kraken2 outputs! **First, make the expected output directory**, then run the following:

```
parallel -j 2 --eta 'bracken -d k2_pluspf_08gb -i {} -o bracken_out/{/.}.species.bracken -r 150 -
```

Some notes about this command: * `-d` specifies the database we want to use. It should be the same database we used when we ran Kraken2; * `-i` is our input file(s); * `-o` is where and what we want the output files to be; * `-r` is the read length to get all classifications for, the default is 100 but our reads are 150bp; * `-l` is the taxonomic level at which we want to estimate abundances; * `-t` is the number of reads required prior to abundance estimation to perform re-estimation

Finally, let's merge our bracken outputs:

```
combine_bracken_outputs.py --files bracken_out/*.species.bracken -o merged_output.species.bracken
```

2.4.2 Annotation with MetaPhlAn

Another tool that is commonly used for taxonomic annotation of metagenomic sequences is MetaPhlAn. This tool is different from Kraken2 in that it uses a

database of marker genes, instead of a collection of genomes, and it identifies only these marker genes within our reads, rather than trying to classify all reads. It then attempts to estimate the abundance of the taxa it identified within our whole samples, but it's important to remember that this is an **estimation**, and not the actual number of reads classified. We will use MetaPhlAn 3 for this tutorial due to the constraints of the AWS instances, but a newer version (MetaPhlAn4) utilizing a larger database is available.

Usually, we would build this ourself so that we make sure to have the most recent version. This can take quite a while, so we will create a link to it:

```
ln -s ~/CourseData/MIC_data/ref_dbs/CHOCOPhlAn_201901 .
```

And then activate the environment:

```
conda activate metaphlan3
```

Now, let's make an output folder `metaphlan_out`. Then, we can run the following:

```
parallel -j 1 --eta 'metaphlan --bowtie2db CHOCOPhlAn_201901 --input_type fastq -o meta
```

This might take a while to run, so move on to the next step while this runs and then come back to this after.

2.5 3. Visualisation of results in R

Now we'll go to RStudio server: `http://##.uhn-hpc.ca:8080` (remember to replace ## with your own instance number). If you weren't in the beginner microbiome workshop, you may be prompted to enter your username and password. It will also take a little while to log in the first time, but will be quicker afterwards.

2.5.1 Create the R Notebook

Using the menus, click `File > New File > R Notebook`, which will open an Untitled R markdown (Rmd) document. R notebooks are helpful in that you can run several lines, or ***chunks*** of code at the same time, and the results will appear within the document itself *(in the whitespace following the chunk).*

The default R Notebook contains a header and some information you may find helpful. Try running the chunk containing `plot(cars)` to see what happens!

You do not need to preserve most of the information in the new, untitled document. Select all of its contents by click+dragging your cursor or entering the **ctrl+a** (Windows) / **cmd+a** (Mac) shortcut, and press **backspace** or **delete** to clear the document.

The **chunks** are distinguished by the grey shading. Everything between the first ```{r} and subsequent``` belongs to the chunk. Anything written in the white space surrounding the chunk is meant to be annotation. *Although you can run lines of code outside of the chunks, the chunks are useful for running multiple lines in series with one click.*

2.5.1.1 Adding new chunks

To add a new chunk into your R notebook by navigating to `Code > Insert Chunk` from the toolbar, or clicking on the little green C and selecting R.

And then save the document - it doesn't really matter what you call it, but something like `amb_module1.rmd` would be sensible.

2.5.2 Setup, Importing and Formatting Data

Now we'll start building our R markdown notebook. Paste the following into a chunk, and then click the little green "play" button on the top right of the chunk to run it.

```
library(phyloseq)
library(vegan)
library(ggplot2)
library(gridExtra)

setwd("~/workspace/amb_module1/")
```

In this chunk, we've imported the libraries/packages that we're going to use (phyloseq, vegan, ggplot2 and gridExtra) and we've told R which directory it should be working from.

Now we're going to read in the Bracken output, and do some formatting of it:

```
bracken <- read.csv("merged_output.species.bracken", sep="\t") #read in the file as a dataframe
rownames(bracken) = bracken[,1] #set the row names to those in the first column
cols = c() #make an empty list/vector
for (colname in colnames(bracken)) { #for each of the columns in bracken
  if (grepl("bracken_num", colname, fixed = TRUE)) { #if "bracken_num" is in the column name
    cols <- c(cols, colname) #add it to the columns list - we're going to use this to filter/subs
  }
```

```
}
bracken = bracken[,cols] #subset the dataframe to only the columns in the cols list (i
colnames(bracken) = sapply(strsplit(colnames(bracken),"_"), `[`, 1) #rename the column
```

If you're not familiar with R or Python, anything after the # can be used for making comments, as it won't be read by them. So you can see what I've written about each step.

It's good practice to take a look at what we're changing each time! In the "Environment" section of RStudio, you should be able to see all of these objects. Click on them to have a look. I won't always tell you to click on them, but it's a good idea to either click on them or print them out so that you can understand what is changing each time. There are some other ways to look at this too:

```
View(bracken)
print(bracken)
bracken
```

Some of these options are more or less appropriate depending on what you're doing, but regularly printing things out is the easiest way to troubleshoot a script that isn't working. If you were wanting to print something out within a loop, you'd need to use the `print()` function.

Now we're going to get the genus name of each of the species in our dataframe:

```
all_genus = c() #make an empty list/vector
for (species in rownames(bracken)) { #for each of the rows (species) in the bracken dat
  this_species = species
  if (grepl("uncultured", this_species, fixed = TRUE)) { #if "uncultured" is in the nar
     this_species = gsub("uncultured ", "", this_species) #rename the species by removi
  }
  this_genus = strsplit(this_species, " ")[[1]][1] #get the first word of the species n
  all_genus = c(all_genus, this_genus) #add this genus name to our list
}
tax_table = data.frame(genus=all_genus, species=rownames(bracken)) #make a new table wi
rownames(tax_table) = rownames(bracken) #and add the rownames to it
```

Next, we'll read in our metadata and reformat it:

```
metadata = read.csv("metagenome_metadata.txt", sep="\t") #read in the file as a datafra
samples = data.frame(metadata[,2], stringsAsFactors = FALSE) #get only the second colun
rownames(samples) = metadata[,1] #add the sample names
colnames(samples) = c('group') #change the column name
samples['classifier'] = 'kraken' #add a new column that shows us that these samples wer
```

And now we'll combine these objects to a phyloseq object:

```
BRACKEN = otu_table(bracken, taxa_are_rows = TRUE) #convert bracken to an otu_table
SAMPLE = sample_data(samples) #convert the samples dataframe to a phyloseq sample_data format
TAX = tax_table(tax_table) #convert the tax_table dataframe to a phyloseq tax_table format
taxa_names(TAX) <- rownames(bracken) #get the taxa names from the bracken row names
physeq_bracken = phyloseq(BRACKEN, TAX, SAMPLE) #make the phyloseq object
```

Print out `physeq_bracken` to see if it has all of the taxa and samples that you are expecting.

And collapse the table at the genus level:

```
physeq_bracken_genus = tax_glom(physeq_bracken) #collapse at the genus level (the default is the
taxa_names(physeq_bracken_genus) = data.frame(tax_table(physeq_bracken_genus))$ta1 #add the genus
```

Have a look at it:

```
View(physeq_bracken_genus)
```

2.5.3 Filtering and rarefying

Now we can take a look at how many reads have been classified within our samples:

```
sample_sums(physeq_bracken_genus)
print(c("Minimum sample size:",min(sample_sums(physeq_bracken_genus)), "Maximum sample size:", ma
```

QUESTION!

Question 4: Roughly what proportion of the reads you have in the samples after KneadData have been classified? Is this surprising to you? **Remember that you can look in the 'kneaddata_read_counts.txt' file to see how many reads the files all ended up with!**

It's also always good to have a look at a rarefaction curve. This tells us about whether we've sequenced our samples sufficiently to capture all of the diversity within them:

```
rarecurve(as.data.frame(t(otu_table(physeq_bracken_genus))), step=50,cex=0.5,label=TRUE, ylim = c
```

QUESTION!

Question 5: How do these look to you? If these were your samples, would you be happy with these?

Let's also take a look at the number of reads assigned to each taxon within our samples:

```
length(taxa_sums(physeq_bracken_genus))
print(c("Minimum reads:",min(taxa_sums(physeq_bracken_genus)), "Maximum reads:", max(ta
taxa_sums(physeq_bracken_genus)
```

You'll see that we have a minimum of 1 read assigned to a genus, and a maximum of >14M reads. Usually, we'll remove some of the very low abundance taxa in our samples.

```
physeq_bracken_genus<-prune_taxa(taxa_sums(physeq_bracken_genus)>=10,physeq_bracken_ge
```

If you take a look at this, you'll see that we've already gone from >1500 genera to only 490. But we can reduce this a little further:

```
physeq_bracken_genus<-prune_taxa(taxa_sums(physeq_bracken_genus)>=100,physeq_bracken_g
```

Now we only have 112 genera, and with a minimum of a sum of 100 reads (average 10 per sample), this seems fairly reasonable for filtering.

Let's take a look at the rarefaction curves again:

```
rarecurve(as.data.frame(t(otu_table(physeq_bracken_genus))), step=50,cex=0.5,label=TRU
```

QUESTION!

Question 6: How do you feel about the samples now?

Next we'll rarefy the samples prior to our diversity calculations - the number of classified reads varies a lot between our different samples!

```
set.seed(711)
rarefied_bracken_genus<-rarefy_even_depth(physeq_bracken_genus, rngseed = FALSE, sample.size = mi
```

See that the samples are all the same size now:

```
sample_sums(rarefied_bracken_genus)
```

NOTE!

We are only going to use this rarefied object for the alpha diversity analyses!

2.5.4 Alpha and beta diversity analyses

We're not going to go into explaining all of these again as we did that in the beginner workshop, but we'll take a look at the samples with a few different methods.

First, we'll look at the alpha diversity:

```
plot_richness(physeq = rarefied_bracken_genus, x="group", color = "group", measures = c("Observed
```

QUESTION!

Question 7: What do you notice about the alpha diversity differences between the samples?

Now we'll convert the non-rarefied phyloseq object to relative abundance:

```
relabun_bracken_genus <- transform_sample_counts(physeq_bracken_genus, function(x) x*100 / sum(x)
```

And then we'll use this for a Bray-Curtis PCoA plot:

```
ordination<-ordinate(relabun_bracken_genus, "PCoA", "bray")
plot_ordination(relabun_bracken_genus, ordination, color = "group") +
  geom_point(size=10, alpha=0.1) + geom_point(size=5) + stat_ellipse(type = "t", linetype = 2) +
```

QUESTION!

Question 8: do the samples group how you expected them to based on their alpha diversity?

Next we'll make a bar plot. Plotting all 112 of the taxa on a barplot is a bit useless, so first we'll filter to include only the 20 most abundant genera:

```
relabun_bracken_genus_top<-prune_taxa(names(sort(taxa_sums(relabun_bracken_genus),decre

plot_bar(relabun_bracken_genus_top, fill="ta1") + facet_wrap(c(~group), scales="free_x"
```

QUESTION!

Question 9: What do you notice about the composition of these samples? What are the samples from the Non-Bacteroides group dominated by?

Often heatmaps are a little easier to look at than a stacked barplot, and it's easier to see differences between samples:

```
plot_heatmap(relabun_bracken_genus_top, method = "PCoA", distance = "bray", low = "whit
```

This function is grouping our samples by their Bray-Curtis distance from one-another, although note that this may not look quite the same as it would in the PCoA plot because we're only looking at the 20 most abundant genera.

That was the plotting inside phyloseq, but sometimes we also want a dendrogram to show how our samples cluster together, so let's also try something else:

```
heatmap(otu_table(relabun_bracken_genus_top))
```

Note that this has just clustered our taxa together, and the tree that we see for the genera is not a phylogenetic tree!

QUESTION!

Question 10: What can you notice about our samples now? Particularly the taxonomic composition of the Non-Bacteroides samples?

2.5.5 Now lets do the same with the MetaPhlAn3 output

MetaPhlAn 3 should have finished running by now! So go back to your terminal window.

You may see a warning come out, but as long as it ran, that doesn't matter!

Similar to Kraken2, MetaPhlAn will output individual files for each sample. We can use a utility script from MetaPhlAn to merge our outputs into one table.

```
merge_metaphlan_tables.py metaphlan_out/*.mpa > metaphlan_merged_out.txt
```

If we view this new combined table, we will see three key things: 1. First, the output format is different to that of Kraken2, where the full taxonomic lineages are expanded line by line. 2. Second, MetaPhlAn only outputs relative abundances for each taxonomic node, whereas Kraken2 (before re-analysis with Bracken) will output absolute numbers of reads assigned to each taxonomic node. 3. Third, the number of taxa that MetaPhlAn finds is *much* smaller than Kraken2. This is partially due to us using a low confidence threshold with Kraken2, but this discrepancy between the two tools tends to hold true at higher confidence thresholds as well. See our paper for more info about how these tools perform compared to each other.

We're going to perform some similar steps on this MetaPhlAn output as we did on the Bracken output, but we'll be able to just keep the lines of the file that are at the genus level, rather than grouping by genus. I haven't explained things much here as you can see more explanation above, so see if you can work out what each of the lines of code is doing as you go.

Read in the file and filter to only the relevant lines:

```
metaphlan <- read.csv("metaphlan_merged_out.txt", sep="\t", skip=1)
rownames(metaphlan) = metaphlan[,1]

keeping = c()
genus = c()
for (row in rownames(metaphlan)) {
  if (grepl("g__", row, fixed = TRUE)) {
    if (grepl("s__", row, fixed = TRUE)) {
      skip_this = TRUE
    } else {
      keeping = c(keeping, row)
      genus = c(genus, strsplit(row,"__")[[1]][7])
    }
  }
}
```

```
metaphlan = metaphlan[keeping,]
metaphlan = metaphlan[,c(-1,-2)]
rownames(metaphlan) = genus
colnames(metaphlan) = sapply(strsplit(colnames(metaphlan),"_"), `[`, 1)
```

Read in the metadata again:

```
metadata = read.csv("metagenome_metadata.txt", sep="\t")
samples = data.frame(metadata[,2], stringsAsFactors = FALSE)
rownames(samples) = metadata[,1]
colnames(samples) = c('group')
samples['classifier'] = 'metaphlan'
```

Make a taxonomy table:

```
tax_table = data.frame(genus=rownames(metaphlan))
rownames(tax_table) = rownames(metaphlan)
```

Combine these into a phyloseq object:

```
METAPHLAN = otu_table(metaphlan, taxa_are_rows = TRUE) #convert asv_table_num to an otu
SAMPLE = sample_data(samples)
TAX = tax_table(tax_table)
taxa_names(TAX) <- rownames(metaphlan)
physeq_metaphlan_genus = phyloseq(METAPHLAN, TAX, SAMPLE)
```

We're going to skip some of the first steps looking at read depths and the number of reads assigned to different species, as we've identified marker genes using MetaPhlAn3 and not reads, and this is relative abundance and not a count.

There are only two alpha diversity metrics that we can look at without counts:

```
plot_richness(physeq = physeq_metaphlan_genus, x="group", color = "group", measures = c
```

Make a PCoA plot with Bray-Curtis distance:

```
ordination<-ordinate(physeq_metaphlan_genus, "PCoA", "bray")
plot_ordination(physeq_metaphlan_genus, ordination, color = "group") +
  geom_point(size=10, alpha=0.1) + geom_point(size=5) + stat_ellipse(type = "t", linety
```

QUESTION!

Question 11: Are these results for alpha and beta diversity as you expect them to be from the Kraken/Bracken results?

We don't have very many taxa in our MetaPhlAn3 output, so we won't bother filtering this before plotting the stacked barplot:

```
plot_bar(physeq_metaphlan_genus, fill="ta1") + facet_wrap(c(~group), scales="free_x", nrow=1) + t
```

Then make the first of the heatmaps again:

```
plot_heatmap(physeq_metaphlan_genus, method = "PCoA", distance = "bray", low = "white", high = "r
```

You'll probably notice a lot of grey spots here - that means that genus wasn't found in the sample at all.

And the second one:

```
heatmap(otu_table(physeq_metaphlan_genus))
```

QUESTION!

Question 12: What do you notice here compared with the Kraken/Bracken results?

2.5.6 Combine Kraken and MetaPhlAn output

First we'll rename both of the Bracken and MetaPhlAn phyloseq objects so that we don't have duplicated sample names in them:

```
sample_names(relabun_bracken_genus) = paste(sample_names(relabun_bracken_genus), "_bracken", sep=
sample_names(physeq_metaphlan_genus) = paste(sample_names(physeq_metaphlan_genus), "_metaphlan",
```

Hopefully you can see how we're just pasting together each of the sample names with either "_bracken" or "_metaphlan"

Now join the two phyloseq objects:

```
physeq_combined = merge_phyloseq(relabun_bracken_genus, physeq_metaphlan_genus)
```

Make two PCoA plots, one with Jaccard distance (looking at presence/absence only) and then one with Bray-Curtis dissimilarity (that also accounts for the abundance of each genus:

```
ordination<-ordinate(physeq_combined, "PCoA", "binary")
plot1 <- plot_ordination(physeq_combined, ordination, color = "group", shape = "classi
  geom_point(size=10, alpha=0.1) + geom_point(size=5) + stat_ellipse(type = "t", linety
ordination<-ordinate(physeq_combined, "PCoA", "bray")
plot2 <- plot_ordination(physeq_combined, ordination, color = "group", shape = "classi
  geom_point(size=10, alpha=0.1) + geom_point(size=5) + stat_ellipse(type = "t", linety
grid.arrange(plot1, plot2, ncol=2)
```

Does this look weird? Click the little button on the right between the code and the plot to "Show in new window".

QUESTION!

Question 13: Anything surprising here? What do you notice about the two plots?

We had included all of the taxa previously, but Bracken has a lot more genera identified than MetaPhlAn. So we'll filter the Bracken output to be a little more similar:

```
relabun_bracken_genus_top<-prune_taxa(names(sort(taxa_sums(relabun_bracken_genus),decre
physeq_combined = merge_phyloseq(relabun_bracken_genus_top, physeq_metaphlan_genus)
```

Now our PCoA plot looks a little nicer:

```
ordination<-ordinate(physeq_combined, "PCoA", "bray")
plot_ordination(physeq_combined, ordination, color = "group", shape = "classifier") +
  geom_point(size=10, alpha=0.1) + geom_point(size=5) + stat_ellipse(type = "t", linety
```

Let's have a look at the taxa in the samples:

```
plot_bar(physeq_combined, fill="ta1") + facet_wrap(c(~"group", "classifier"), scales="
```

Can you see all of the taxa? Try filtering to only the top 20, modifying the code that we used above.

And look at the heatmap:

```
heatmap(otu_table(physeq_combined))
```

QUESTION!

Question 14: What differences do you notice between the two classifiers?

2.6 Answers

Question 1: This one isn't really a question, but you can see the code needed for these steps in the answers section if you need to!

```
grep "phiX174" assembly_summary_refseq.txt
#go to https://ftp.ncbi.nlm.nih.gov/genomes/all/GCF/000/819/615/GCF_000819615.1_ViralProj14015/
wget https://ftp.ncbi.nlm.nih.gov/genomes/all/GCF/000/819/615/GCF_000819615.1_ViralProj14015/GCF_
```

Question 2: Take a look at this file. Are there any surprises? Which of the output files in `kneaddata_out` will you use for analysis? Nope! There are no samples where we seem to lost a really large number of reads at any step - we don't lose many reads that couldn't be trimmed (difference between `raw single` and `trimmed single`), and not too many of the reads are host-associated (difference between `trimmed single` and `decontaminated chicken_Gallusgallus_phiX174 single`). We'll use the `sample_kneaddata.fastq` files for analysis, and not the `sample_kneaddata_chicken_Gallusgallus_phiX174_bowtie2_contam.fastq` files.

Question 3: How many reads are in each sample? The original number of reads per sample ranges between 931,686 and 1,017,130 (average 964,187) and after KneadData ranges between 912,712 and 998,264 (average 950,470).

Question 4: Roughly what proportion of the reads you have in the samples after KneadData have been classified? Is this surprising to you? We have between 52,405 (`min(sample_sums(physeq_bracken_genus))`) and 446,368 (`max(sample_sums(physeq_bracken_genus))`) - average 221,440 (`mean(sample_sums(physeq_bracken_genus))`). 221,440/950,470 = ~23% of reads classified. This does seem relatively low, although it's not uncommon for only a small proportion of the reads in samples not from humans (i.e. from less well-characterised environments) to be classified. In this case, it's likely mainly due to the really small database that we've used for taxonomic classification. We've used an 8GB database due to the constraints of these AWS instances,

but we recommend using the largest database that you possibly can - the ones that we typically use in our lab are 800GB-1.1TB in size.

Question 5: How do these look to you? If these were your samples, would you be happy with these? These curves don't look great - they don't saturate at all.

Question 6: How do you feel about the samples now? This is much better - and we can see that our sample rarefaction curves not being saturated is down to these very low abundance taxa. While some of them could be important, anything present at abundances this low could be due to false-positive noise, and we didn't do any filtering with Bracken.

Question 7: What do you notice about the alpha diversity differences between the samples? There are huge differences between our Bacteroides and Non-Bacteroides sample groups. The samples in the Bacteroides group have much lower diversity than the Non-Bacteroides samples.

Question 8: do the samples group how you expected them to based on their alpha diversity? Yes. All of the Bacteroides samples group away from the Non-Bacteroides samples, which isn't surprising based on how different their alpha diversity is.

Question 9: What do you notice about the composition of these samples? What are the samples from the Non-Bacteroides group dominated by? The Bacteroides samples have very low abundance of anything other than Bacteroides, while the Non-Bacteroides samples have many other genera also present. There isn't really any one genus that dominates the Non-Bacteroides samples, although a couple do seem to have high Escherichia.

Question 10: What can you notice about our samples now? Particularly the taxonomic composition of the Non-Bacteroides samples? There is very little of anything but Bacteroides in the Bacteroides samples. The Non-Bacteroides are not particularly similar to one-another, with a range of taxa present in mid-high abundances in each of them.

Question 11: Are these results for alpha and beta diversity as you expect them to be from the Kraken/Bracken results? Yes - the Bacteroides and Non-Bacteroides samples are still separating from one another and the alpha diversity is much lower in Bacteroides than Non-Bacteroides samples.

Question 12: What do you notice here compared with the Kraken/Bracken results? These samples are very sparse! Lots of the taxa are only identified in one or a few samples. This is one of the draw-backs of the marker-gene approach used by MetaPhlAn - we can typically be very confident in the taxa that it identifies as present, but it is likely to miss lots of taxa, too (low recall, high precision).

Question 13: Anything surprising here? What do you notice about the two plots? Jaccard distance only takes into account presence/absence of taxa, while Bray-Curtis also accounts for abundance. It's surprising how similar the two

groups (Bacteroides/Non-Bacteroides) look with Bray-Curtis when they are so different with Jaccard, but this means that the most abundant taxa must be similar between the two different classification methods.

Question 14: What differences do you notice between the two classifiers? The Bacteroides samples look very similar between the two methods, but the Non-Bacteroides samples seem to be more different. Escherichia seems to dominate three MetaPhlAn samples but only two Kraken samples. There are lots of other differences too! See if you can come up with some explanations for these differences.

Chapter 3

Module 3 Lab: Assigning Functions

In this lab, we will predict and briefly examine the functions encoded in the assembled metagenome from the previous labs. Optional context and questions are provided as collapsible sections like so:

Click me

Optional context is provided to make this workshop as inclusive as possible. Feel free to skip any topics that you are already familiar with. Questions and answers will also use this collapsible format.

All instructions required to completed this lab are provided below, so you are free to work asynchronously at your own pace. Active discussions with both peers and instructors are encouraged. Say hi to the person next to you!

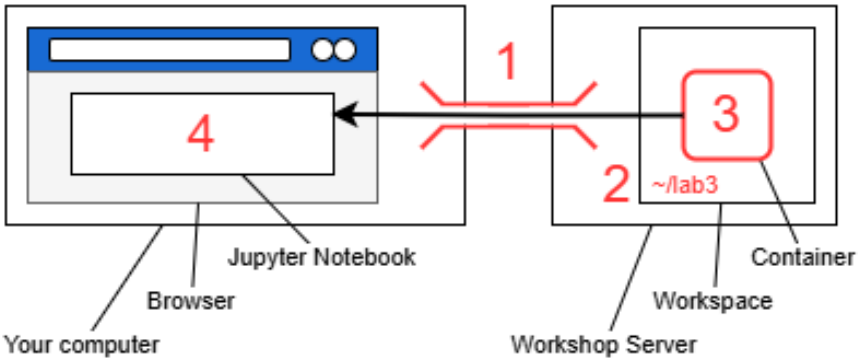
3.1 Setting up the workspace

Before we begin, this section will guide you through the set up of your workspace and compute environment. The software tools used in this lab all have dependencies that must be installed separately. Constructing the environment for each tool is non-trivial and error-prone. The difficulty of replicating compute environments for bioinformatics software is one of the key contributing factors to the reproducibility crisis [1].

We will instead be using prepackaged compute environments called "containers", implemented by Apptainer (formerly Singularity) [2]. One of these containers will enable the use of jupyter notebooks along with a collection of utilities.

1. Mangul S, Martin LS, Eskin E, Blekhman R. Improving the usability and archival stability of bioinformatics software. Genome Biol. 2019;20(1):47. https://doi.org/10.1186/s13059-019-1649-8
2. Kurtzer GM, Sochat V, Bauer MW. Singularity: Scientific containers for mobility of compute. PLOS ONE. 2017;12(5):e0177459. https://doi.org/10.1371/journal.pone.0177459

3.1.0.1 Steps for setup:



1. Open an additional port via SSH for the new jupyter notebook
2. Create a new workspace directory for this lab
3. Unpack and start the container
4. Open the jupyter notebook in your browser

3.1.1 SSH port forwarding

We will use SSH port forwarding to open the additional port.

3.1.1.1 Terminal (Unix/MacOS)

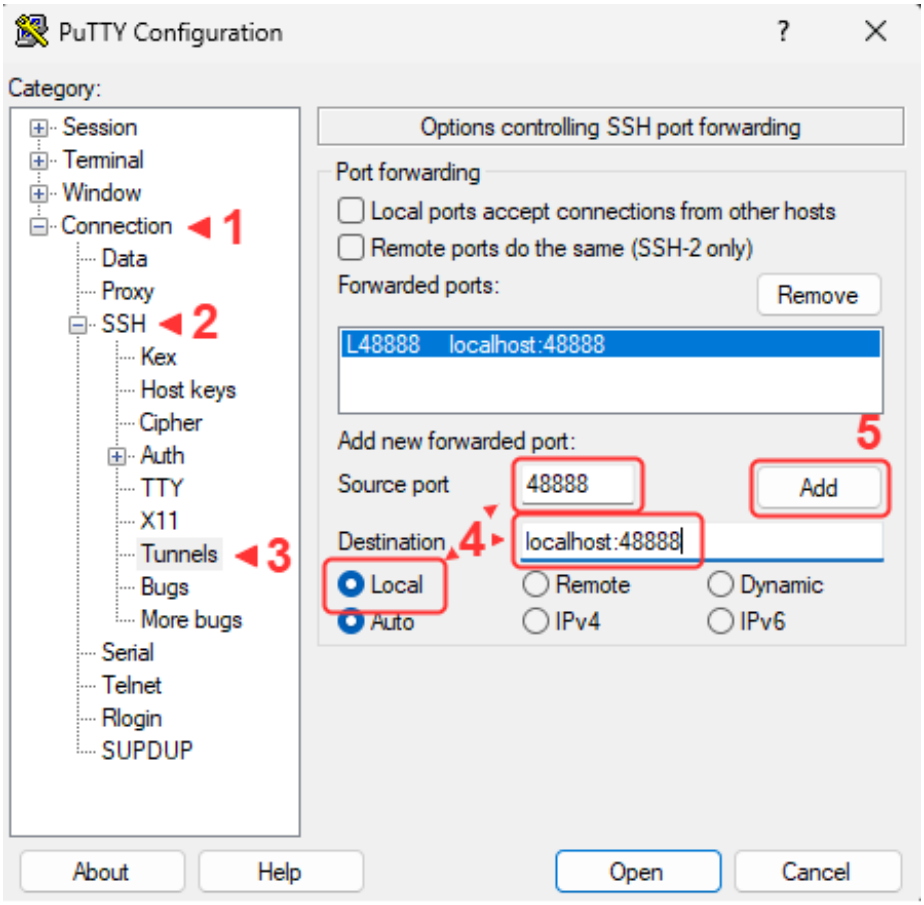
Add `-L 48888:localhost:48888` to the SSH command you use to connect to the server. For example:

```
ssh -L 48888:localhost:48888 ubuntu@${your_number}.uhn-hpc.ca -i ${path_to_your_key}
```

This means port `48888` on the remote server will be forwarded to your machine at `localhost:48888`.

3.1.1.2 Putty (Windows)

Modern versions of Windows should have `ssh` available through Powershell. Putty is available as a backup.



3.1.2 Workspace

Create a new workspace in your home directory.

```
cd ~
mkdir amb_lab3
cd amb_lab3
```

Link the resource folder for easy access.

```
ln -s ~/CourseData/MIC_data/amb_module3/ ./lib
```

3.1.3 Start container

Most containers for this lab were obtained from the biocontainers project, which provides a large collection of bioinformatics software in a standard format. For each tool, we will provide URIs for the the specific container image used and where to get future versions. The `pull` command of `apptainer` can be used to download images (docs: https://apptainer.org/docs/user/main/cli/apptainer_pull.html).

Here is an example which downloads `prodigal` as the container image `example.sif` from `docker://quay.io/biocontainers/prodigal:2.6.3--h7b50bb2_10`. The image URI takes the form of `<protocol>://<address>:<tag>`. Tags are used to identify versions https://quay.io/repository/biocontainers/prodigal?tab=tags.

```
1 apptainer pull example.sif docker://quay.io/biocontainers/prodigal:2.6.3--h7b50bb2_10
```

We can now run the prodigal executable inside the container using `exec` (execute) like so.

```
apptainer exec ./example.sif prodigal -h
```

All containers for this lab were pre-downloaded in the resource folder now linked as `./lib`, including a main container with Jupyter and miscellaneous utilities. The main container provides an automated setup script named `unpack`. Let's run it.

```
apptainer exec ./lib/amb_lab3.sif unpack
```

The following will dedicate the current terminal to running the juptyer notebook on port 48888. You may want to connect another terminal to maintain terminal access.

```
./start_lab
```

3.1.4 Jupyter

JupyterLab should now be accessible at http://localhost:48888/lab. Create a new python notebook and import dependencies.

```
import pandas as pd
from pathlib import Path
from local.ipc import Shell

LIB = Path("./lib")
CPUS = 4 # your workshop instance has been alloted 4 cores
```

- `pandas` enables parsing and reading of tables
- `pathlib` enables manipulation of filesystem paths
- Pre-downloaded container images are in `LIB`
- Pre-computed outputs for this lab are available at `LIB/"outputs"`.

Since containers can not be run inside other containers, the provided `Shell` function grants access to the terminal outside of the main container.

```
Shell(f"""
pwd -P
echo {LIB}
""")
```

Python comments `# ...`

Comments are free text not meant to be interpreted as python code and are indicated by a preceeding `#`. They are useful for leaving notes.

```
# <helpful information>
```

Python variables `CPUS = 4`

Useful values can be saved to variables using `=` for later use.

```
x = 1 # the "value" 1 is assigned to the "variable" x
```

Python functions `Shell(...)`

Functions are saved procedures that can be repeatedly executed through "calls". Some functions recieve additional information as "arguments" or "return" information back when called. Called functions execute their procedure and are

"evaluated" to their return value. Functions without a return evaluate to the dedicated null value `None`.

```
# function definition syntax
def function_name(): # no arguments
    pass # do nothing, return nothing

# define the function "add"
def add(a, b):
    return a + b

# call function "add"
x = add(1, 2) # add evaluates to 1 + 2, then to 3, then is assigned to x
x # x is 3
```

`Shell` accepts text and executes it as a bash command.

Python `import` and dot notation `local.ipc`

```
from local.ipc import Shell # access "ipc" from "local"
```

`local` is a folder containing the file `ipc.py`, which in turn defines the function `Shell`. `local` is a python "module" and `ipc.py` is a python "script". `import` makes python components available from other files, typically called "libraries". Dot notation (the period between `local` and `ipc`) retrieves the named components of complex objects. An alternative syntax to the above is:

```
import local.ipc.Shell # retrieve Shell from ipc in local
```

imported components can be renamed using the `as` keyword:

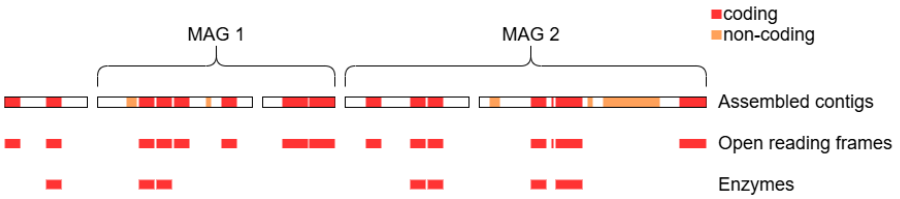
```
import local.ipc.Shell as Terminal # Shell is now "Terminal"
```

Python strings `f"..."`

Text values in python (strings), such as the contents of a bash command, must be declared in quotes like `"text"` or `'text'`. Triple quotes allows multi-line strings. The `f` prefix enables the evaluation of python variables in the string. For example, `f"echo {LIB}"` evaluates to `echo ./lib`. This is a convient way to simplify commands by saving repeated sections to variables.

3.2 Overview

This lab will guide you through finding and identifying the functional units (represented by the red and orange features below) of a metagenomic assembly, with a focus on proteins and enzymes.



We will begin by exploring techniques related to sequence similarity, which seeks to identify known genes through comparison with reference databases.

- Prodigal
- Diamond
- Bakta
- Resistance Gene Identifier (RGI)

Then we will transition to machine-learning tools, which move away from the "lookup table" approach of sequence similarity.

- Kofamscan
- deepFRI
- deepEC
- proteinBERT

Finally, we will use gapseq to construct toy metabolic models using techniques to predict the existence of enzymes.

- gapseq

3.3 Prodigal

The container image for prodigal:

https://quay.io/repository/biocontainers/prodigal?tab=tags

```
# apptainer pull ./lib/prodigal.sif docker://quay.io/biocontainers/prodigal:2.6.3--h7b50bb2_10
prodigal_sif = LIB/"prodigal.sif"
```

The goal of prodigal is to predict the protein coding features (open reading frames; ORFs) in a given nucleotide sequence. Recall that this corresponds to the red features in the overview diagram, neglecting the orange features for now. It does this by optimizing predictions based on a set of tuned rules (heuristics) which take into account gene length and overlap, start and stop codons, ribosome binding sites, and GC content [1].

1. Hyatt D, Chen GL, LoCascio PF, Land ML, Larimer FW, Hauser LJ. Prodigal: prokaryotic gene recognition and translation initiation site identification. BMC Bioinformatics. 2010;11(1):119. https://doi.org/10.1186/1471-2105-11-119

Based on this, we would expect prodigal to:

- accept a nucleotide sequence as input
- output an itemized list of predicted open reading frames

The input sequence will be the metagenomic assembly since we would otherwise neglect some contigs that were not assigned to MAGs, thereby causing information loss. Let's have a look at prodigal's help message to get an idea of specific inputs and outputs.

```
Shell(f"apptainer exec {prodigal_sif} prodigal -h")
```

This looks like the input:

- `-i: Specify FASTA/Genbank input file (default reads from stdin).`

These look like they are useful for the output:

- `-a: Write protein translations to the selected file.`
- `-d: Write nucleotide sequences of genes to the selected file.` These will be helpful next lab for aligning transcripts to genes.
- `-f: Select output format (gbk, gff, or sco). Default is gbk.` We will choose `gff` since the genbank output of prodigal is not in a standard formatted.
- `-o: Specify output file (default writes to stdout).`

After a bit of investigation, it also looks like we should specify that the input is a metagenomic assembly.

- `-p: Select procedure (single or meta). Default is single.`

Have a go at compiling the final command.

```
out_dir = "./prodigal"
Shell(f"""
mkdir -p {out_dir}/
apptainer exec {prodigal_sif} \
    prodigal \
        -i {LIB}/inputs/assembly.fa \
        ...
""")
```

The purpose of \

Starting from `apptainer exec`, the command must be written in a single line. The backslash \ is used to ignore the newline character and continue the command on the next line. This let's us write break up a long command into multiple lines for readability. —

Suggested answer

```
out_dir = Path("./prodigal")
prodigal_aa = out_dir/"orfs.faa"
prodigal_gff = out_dir/"orfs.gff"
Shell(f"""\
mkdir -p {out_dir}/
apptainer exec {prodigal_sif} \
    prodigal \
        -p meta \
        -i {LIB}/inputs/assembly.fa \
        -a {prodigal_aa} \
        -d {prodigal_aa.with_suffix('.fna')} \
        -f gff \
        -o {prodigal_gff} \
""")
```

Let's take a peek at the `gff`.

```
gff = pd.read_csv(out_dir/"orfs.gff", sep="\t", header=None, comment="#")
print(gff.shape)
gff.head(2)
```

From left to right, the columns are:

0. contig id
1. name of the program that generated this feature
2. feature type, where `CDS` stands for coding sequence
3. start position (first nucleotide is 1)
4. end position
5. raw prodigal score
6. strand, where `+` is forward and `-` is reverse
7. codon frame, being one of 0, 1 or 2
8. a semicolon-separated list of tag-value pairs, providing additional information about each feature

Selecting entries in python dataframes and matrices

Entries (rows, columns, or cells) in pandas dataframes and numpy matricies can be selected in multiple ways.

Example matrix:

```
mat = np.array([1, 2, 3, 4, 5])
```

Select by single index, where negative indexes count from the end:

```
mat[0]      # 1
mat[-1]     # 5
```

a slice in the form of `start:end`, where `start` is inclusive and `end` is exclusive:

```
mat[1:4]    # [2, 3, 4]
mat[:2]     # [1, 2]
mat[2:]     # [3, 4, 5]
mat[-1:]    # [5]
```

a list of indexes:

```
mat[[0, 2, 4]] # [1, 3, 5]
```

a boolean mask:

```
mat[[False, False, False, True, True]] # [4, 5]
```

Let's investigate how many ORFs were not on MAGs. To do this, we will first create a blacklist for mag contig IDs.

```
mag_ids = set()
# loop through the paths to each of the MAG files
for mag_file in [
    LIB/"inputs/HMP2_MAG_00001-contigs.fa",
    LIB/"inputs/HMP2_MAG_00002-contigs.fa",
]:
    # we can use SeqIO to iterate through each sequence in a fasta file
    for contig in SeqIO.parse(mag_file, "fasta"):
        mag_ids.add(contig.id) # add the contig id of each contig to mag_ids
len(mag_ids)
```

Python `for` loops

Loops enable repeated execution of a block of code for each element in a collection.

```
collection = [1, 2, 3] # a "list" collection
for element in collection:
    print(element) # do something with each element

# result:
# 1
# 2
# 3
```

Then filter the gff based on the MAG blacklist.

```
_filter = ~gff[0].isin(mag_ids) # column 0 is the contig id
non_mag_orfs = gff[_filter]
non_mag_orfs.shape
```

How many ORFs were not from MAGs?

1060 of 7176, or about 15%

3.4 Sequence similarity

In this section, we will use tools to find proteins with known sequences that are similar to that of the predicted ORFs. By assuming that similar sequences equate to similar functions, we can assign functions through reference databases of known proteins. While not perfect [1, 2], this is a common first-pass approach to assign functions to genes in metagenomic assemblies.

1. Omelchenko MV, Galperin MY, Wolf YI, Koonin EV. Non-homologous isofunctional enzymes: A systematic analysis of alternative solutions in enzyme evolution. Biol Direct. 2010;5(1):31. https://doi.org/10.1186/1745-6150-5-31
2. Seffernick JL, de Souza ML, Sadowsky MJ, Wackett LP. Melamine deaminase and atrazine chlorohydrolase: 98 percent identical but functionally different. J Bacteriol. 2001;183(8):2405–10. https://doi.org/10.1128/JB.183.8.2405-2410.2001

3.4.1 Diamond

https://quay.io/repository/biocontainers/diamond?tab=tags

```
# docker://quay.io/biocontainers/diamond:2.1.11--h5ca1c30_2
diamond_sif = LIB/"diamond.sif"
```

Diamond is highly optimized for finding similar protein sequences between a list of queries and a list of subjects [1]. It has a similar command line interface (CLI) to the classic BLAST, requiring that we first build a database from the list of subjects, before searching against it with a list of queries.

1. Buchfink B, Xie C, Huson DH. Fast and sensitive protein alignment using DIAMOND. Nat Methods. 2015;12(1):59–60. https://doi.org/10.1038/nmeth.3176

We expect to run diamond in two steps:

1. Build a reference database
    - inputs: a list of protein sequences
    - outputs: a database file
2. Search the database
    - inputs: a list of protein sequences, the compiled database
    - outputs: a list of hits

```
Shell(f"apptainer exec {diamond_sif} diamond --help")
```

We will build a database for transporters using the transporter classification database (TCDB) [1]. TCDB has been predownloaded from these URLs.

- protein sequences: https://www.tcdb.org/public/tcdb
- sequences are grouped into families: https://www.tcdb.org/cgi-bin/projectv/public/families.py
- transporter substrates: https://www.tcdb.org/cgi-bin/substrates/getSubstrates.py

1. Saier MH Jr, Tran CV, Barabote RD. TCDB: the Transporter Classification Database for membrane transport protein analyses and information. Nucleic Acids Res. 2006;34(suppl_1):D181–6. https://doi.org/10.1093/nar/gkj001

```
Shell(f"apptainer exec {diamond_sif} diamond makedb")
```

The important arguments appear to be:

- `--db database file`
- `--in input reference file in FASTA format/input DAA files for merge-daa`

Have a go at compiling the command to build the diamond database. Running a your command is a good way to check for correctness and resulting error messages can be informative.

```
tcdb_fasta = LIB/"transporter_classification_db/tcdb.faa"
tcdb_db = Path("./diamond/tcdb.dmnd")
Shell(f"""
apptainer exec {diamond_sif} diamond makedb \
    ...
""")
```

Possible solution

```
tcdb_fasta = LIB/"transporter_classification_db/tcdb.faa"
tcdb_db = Path("./diamond/tcdb.dmnd")
Shell(f"""
mkdir -p {tcdb_db.parent} # parent folder of given path
apptainer exec {diamond_sif} diamond makedb \
    --threads {CPUS} \
    --in {tcdb_fasta} \
    --db {tcdb_db}
""")
```

Now we can search our predicted ORFs for transporters catalogued in TCDB.

```
Shell(f"apptainer exec {diamond_sif} diamond blastp")
```

That produced a lot of output, but these are the essential arguments:

- `--query` input query file
- `--db` database file
- `--out` output file

These may be frequently useful:

- `--fast` enable fast mode
- `--sensitive` enable sensitive mode
- `--ultra-sensitive` enable ultra sensitive mode
- `--threads` number of CPU threads
- `--outfmt` output format
- `--evalue` maximum e-value to report alignments (default=0.001)

I find that this works well for general cases.

```
columns = "qseqid sseqid score pident evalue"
diamond_out = Path("./diamond/orfs.tsv")
Shell(f"""
apptainer exec {diamond_sif} \
    diamond blastp \
        --query ./prodigal/orfs.faa \
        --db {tcdb_db} \
        --out {diamond_out} \
        --outfmt 6 {columns} \
        --threads {CPUS}
""")
```

In addition to e-evalue, we can calculate a blast score ratio (BSR) to assign confidence to the hits. BSR for a hit is defined as the ratio of its score to the score of the query aligned to itself (the best possible score). A BSR of less than 0.4 was found to be a good threshold [1].

1. Rost B. Twilight zone of protein sequence alignments. Protein Eng. 1999;12(2):85–94. https://doi.org/10.1093/protein/12.2.85

Perform a self blast. We can skip making a database since the number of sequences is small.

```
diamond_self = Path("./diamond/self.raw")
Shell(f"""
apptainer exec {diamond_sif} \
    diamond blastp \
        --query {prodigal_aa} \
        --db {prodigal_aa} \
        --out {diamond_self} \
        --outfmt 6 {columns} \
        --max-target-seqs 1 \
        --faster \
        --threads {CPUS}
""")
```

“string”.split() and “string”.join()

Strings can be split into lists using the `split` method.

```
"a b c".split(" ") # ["a", "b", "c"]
"a|b|c".split("|") # ["a", "b", "c"]
"c_001_3".split("_") # ["c", "001", "3"]
```

Lists can be joined into strings using the `join` method.

```
" ".join(["a", "b", "c"]) # "a b c"
"_".join(["c", "001"]) # "c_001"
```

We now remove hits with a BSR < 0.4. First, load in the self blast results.

```
# get self scores
df_self = pd.read_csv(diamond_self, sep="\t", header=None, names=columns.split(" "))
df_self = df_self[df_self["qseqid"] == df_self["sseqid"]] # use boolean mask to select
print(df_self.shape)
df_self.head(2)
```

As the code gets more complex, it may be helpful to examine individual components to increase understanding. For example, have a look at:

```
columns.split(" ")
# and
df_self["qseqid"] == df_self["sseqid"]
# or even
df_self["qseqid"]
```

Calculate BSR

```
df_raw = pd.read_csv(diamond_raw, sep="\t", header=None, names=columns.split(" "))
# merge in self scores by matching the qseqid
# "left" refers to a "left join" where
# each row in the left dataframe (df_raw) is found a match in the right (df_self)
# duplicate columns from the right df are suffixed with "_self", such as "score_self"
df_hits = df_raw.merge(df_self, on="qseqid", suffixes=("", "_self"), how="left")
# calculate bsr and place in new column
df_hits["bsr"] = df_hits["score"] / df_hits["score_self"] # divide each value in score
df_hits.head(2)
```

Filter out hits with BSR < 0.4

```
df_hits = df_hits[df_hits["bsr"] >= 0.4] # filter out rows with bsr < 0.4
df_hits = df_hits[columns.split(" ") + ["bsr"]] # keep only the columns we want
df_hits.head(2)
```

Take the best hit

```
# sort by bsr and group by qseqid, keeping the first row of each group
# this will keep the best hit for each query sequence
# groupby("qseqid") will set the index to qseqid, so we reset the index
df_hits = df_hits.sort_values("bsr", ascending=False).groupby("qseqid").first().reset_
```

```
df_hits = df_hits.sort_values("qseqid") # re-sort by qseqid
print(df_hits.shape)
df_hits.head(2)
```

Let's plot the results using a simple diverging bar chart for each MAG. First, let's create whitelists for the contigs in each MAG.

```
MAG1 = set()
for e in SeqIO.parse(LIB/"inputs/HMP2_MAG_00001-contigs.fa", "fasta"):
    MAG1.add(e.id)
MAG2 = set()
for e in SeqIO.parse(LIB/"inputs/HMP2_MAG_00002-contigs.fa", "fasta"):
    MAG2.add(e.id)
len(MAG1), len(MAG2)
```

Python `set`

Sets enable easy checking of membership using `in`. For example:

```
X = {1, 2, 3}
1 in X # True
0 in X # False
empty_set = set()
```

We can use the whitelists with additional parsing to get the family and MAG for each hit.

```
_fams = []
_mags = []
df_fam = pd.DataFrame(df_hits)
for _, row in df_fam.iterrows():
    orf = row["qseqid"]
    parts = orf.split("_")
    contig = "_".join(parts[:-1])
    if contig in MAG1:
        mag = 1
    elif contig in MAG2:
        mag = 2
```

```
    else:
        mag = 0 # not in either MAG
    _mags.append(mag)

    description = row["sseqid"]
    tcdb_id = description.split("|")[-1]
    parts = tcdb_id.split(".")
    family = ".".join(parts[:3])
    _fams.append(family)

df_fam["family"] = _fams
df_fam["mag"] = _mags
df_fam = df_fam[df_fam["mag"] != 0] # remove no-MAG ORFs
print(df_fam.shape)
df_fam.head(2)
```

Python `dict`

Dictionaries enable the lookup of values by their corresponding key.

```
d = {"a": 1, "b": 2, "c": 3}
d["a"] # 1
```

By using `enumerate` to provide the index of each element in an iterable (set, list, dataframe, etc.), dictionaries that map from values to their index can be created.

```
values = ["a", "b", "c"]
value2i = {value: i for i, value in enumerate(values)}
value2i["a"] # 0
```

To visualize the results, we will first organize the data into a matrix of counts for each transporter family and MAG.

```
mat = np.zeros(shape=(2, df_fam["family"].nunique())) # MAGs x families
mag2i = {mag: i for i, mag in enumerate([1, 2])}
fam2j = {fam: i for i, fam in enumerate(df_fam["family"].unique())}
for _, row in df_fam.groupby(["family", "mag"])[["qseqid"]].count().reset_index().iter
    i = mag2i[row["mag"]]
```

```
    j = fam2j[row["family"]]
    mat[i, j] = row["qseqid"]
order = np.argsort(-mat.sum(axis=0)) # produces a sorted list of indexes
```

Some utility functions are provided to make simple figures with `Ploty`. Let's use them to compare the abundance of transporter families in each MAG with a diverging bar plot.

```
from local.figures.template import BaseFigure, ApplyTemplate, go

family_labels = np.array([tcdb_families[k] for k in fam2j.keys()])
TOP_K = 10
fig = BaseFigure()
for mag in [1, 2]:
    i = mag2i[mag]
    sign = -1 if mag == 1 else 1 # -1 for left bar
    fig.add_trace(
        go.Bar(
            x = sign*mat[i, order[:TOP_K]],     # length of each bar
            y = family_labels[order[:TOP_K]],   # name of each row
            orientation='h',
            name = f"MAG{mag}",
        )
    )
    # break # this will only draw the bars of one MAG. Uncomment "break" to see what each trace

fig = ApplyTemplate(
    fig,
    layout=dict(
        legend_orientation ='h',
        barmode="relative",
        width=1000, height=400,
    ),
)
fig.show()
```

How to change the figure to show the top 3 instead?

```
TOP_K = 3
```

Can you put MAG1 on the right and MAG2 on the left?

```
    sign = -1 if mag == 2 else 1
```

3.4.2 Bakta

https://quay.io/repository/biocontainers/bakta?tab=tags

```
# docker://quay.io/biocontainers/bakta:1.9.4--pyhdfd78af_0
bakta_sif = LIB/"bakta.sif"
```

Bakta is an automated workflow for annotating both coding and non-coding features using a variety of tools [1]. Its core protocol can be reduced to ORF prediction by Prodigal, followed by sequence similary search by Diamond, as we have just done. Many additional tools and consolidated reference databases are used to search for non-coding features, provide information on ORFs with no good hits in any database, and perform QC.

1. Schwengers O, Jelonek L, Dieckmann MA, Beyvers S, Blom J, Goesmann A. Bakta: rapid and standardized annotation of bacterial genomes via alignment-free sequence identification. Microb Genomics. 2021;7(11):000685. https://doi.org/10.1099/mgen.0.000685

To run Bakta, we expect to first set up reference databases before it will: - accept contigs, for which it will first run Prodigal for us - or accept a list of regions + contigs, for which Prodigal will be skipped - and produce many files containing useful annotations

Why is a protein sequence fasta insufficient to full utilize Bakta?

Non-coding features would not be represented.

The database was pre-installed, but here is some code to guide you if you decide to revisit in the future.

```
Shell(f"apptainer exec {bakta_sif} bakta_db list")
```

```
# bakta_db = LIB/"bakta_db/db"        # 2 hours, ~70 GB
bakta_db = LIB/"bakta_db/db-light"  # 30 mins, ~3.5 GB
if not bakta_db.exists():
    if "light" in str(bakta_db):
        _type = "light"
    else:
        _type = "full"
    Shell(f"apptainer exec {bakta_sif} bakta_db download --output {bakta_db} --type {_type}")
```

There may be other annotation tools that we wish to run in addition to Bakta, which depend on predicted ORFs. Waiting for Bakta to finish before we recieve the predicted ORFs may be inconvenient, especially for large metagenomic datasets. Running Prodigal ourselves and passing the ORFs to Bakta frees us to run additional tools in parallel. Bakta does not accept ORFs that run off the edge of contigs, so we will need to filter these out.

```
gff = pd.read_csv(prodigal_gff, sep="\t", header=None, comment="#")
complete_orfs = [] # a boolean mask
for _, row in gff.iterrows():
    meta = tuple(row)[-1]
    if "partial=00" in meta:
        complete_orfs.append(True)
    else:
        complete_orfs.append(False)

gff_complete = "./prodigal/orfs.complete.gff"
gff[complete_orfs].to_csv(gff_complete, sep="\t", header=False, index=False)
```

Running Bakta, even on this reduced dataset, may take **15 minutes or more**. Precomputed outputs are available at `{LIB}/outputs/bakta`.

```
Shell(f"""
apptainer exec {bakta_sif} \
    bakta --meta --threads {CPUS} --force --skip-plot \
        --db {bakta_db} \
        --keep-contig-headers \
        --output ./bakta/ \
        --regions {gff_complete} \
        {LIB}/inputs/assembly.fa
""")
```

3.4.3 Resistance Gene Identifier (RGI)

https://quay.io/repository/biocontainers/rgi?tab=tags

```
# docker://quay.io/biocontainers/rgi:6.0.4--pyh05cac1d_0
rgi_sif = LIB/"rgi.sif"
```

Sometimes, specific functions are of interest, such as antibiotic resistance. RGI is a tool for searching the Comprehensive Antibiotic Resistance Database (CARD) [1, 2] using tuned protocols for Diamond.

1. McArthur AG, Waglechner N, Nizam F, Yan A, Azad MA, Baylay AJ, et al. The Comprehensive Antibiotic Resistance Database. Antimicrob Agents Chemother. 2013;57(7):3348–57. https://doi.org/10.1128/aac.00419-13
2. Alcock BP, Huynh W, Chalil R, Smith KW, Raphenya AR, Wlodarski MA, et al. CARD 2023: expanded curation, support for machine learning, and resistome prediction at the Comprehensive Antibiotic Resistance Database. Nucleic Acids Res. 2023;51(D1):D690–9. https://doi.org/10.1093/nar/gkac920

```
Shell(f"apptainer exec {rgi_sif} rgi --help")
```

```
Shell(f"apptainer exec {rgi_sif} rgi main --help")
```

Errors are a common occurrence for bioinformatics tools. The following is a command which appears to be correct, but will nonetheless fail. Read the error messages and use the hints to fix the issue. Additional warnings will appear that do not crash the program. These can be ignored. RGI also provides no indication of progress or successful completion, so checking the output folder will be necessary.

```
rgi_out = Path("./rgi/chicken")
Shell(f"""
apptainer exec {rgi_sif} \
    rgi main \
        --num_threads {CPUS} \
        --clean \
        -i {prodigal_aa} \
        -o {rgi_out}
""")
```

`FileNotFoundError` hint

Is the path to the missing file related to any of the paths we gave it?

`FileNotFoundError` explanation

We indicated an output path within a new folder `rgi_out = Path("./rgi/chicken")`. It appears that RGI can't make this folder for us, so we will have to create it ourselves with `mkdir -p {rgi_out.parent}`. `parent` refers to the parent folder.

`invalid fasta` hint

Nucleotide?

`invalid fasta` explanation

We can use the argument `-t protein` to indicate that the input is a protein sequence.

solution

```
rgi_out = Path("./rgi/chicken")
Shell(f"""
mkdir -p {rgi_out.parent}
apptainer exec {rgi_sif} \
    rgi main \
        --num_threads {CPUS} \
        --clean \
        -t protein \
        -i {prodigal_aa} \
        -o {rgi_out}
""")
rgi_out = rgi_out.with_suffix(".txt") # RGI adds the .txt suffix
# or
# rgi_out = Path("./rgi/chicken.txt")
```

Let's take a quick peak at the results.

```
Shell(f"""
_df = pd.read_csv(rgi_out, sep="\t", header=0)
_df
""")
```

3.5 Machine learning

Alignment-based approaches that assign functional annotations based on sequence similarity rely on databases of known sequnces to transfer their known functions to the ORFs of a given sample. Machine learning may be able to discover more general patterns that can extend annotations to previously unseen sequences.

3.5.1 Kofamscan

https://quay.io/repository/hallamlab/external_kofamscan?tab=tags

```
# docker://quay.io/hallamlab/external_kofamscan:1.3.0
kofamscan_sif = LIB/"kofamscan.sif"
```

Kofamscan uses a machine learning technique called hidden markov models to learn the amino acid sequence patterns accociated with various functions, organized in the KEGG database [1, 2]. Kofamscan sits somewhere in the middle between alignment-based and machine learning-based annotation methods.

1. Aramaki T, Blanc-Mathieu R, Endo H, Ohkubo K, Kanehisa M, Goto S, et al. KofamKOALA: KEGG Ortholog assignment based on profile HMM and adaptive score threshold. Bioinformatics. 2020;36(7):2251–2. https://doi.org/10.1093/bioinformatics/btz859
2. Kanehisa M, Goto S. KEGG: Kyoto Encyclopedia of Genes and Genomes. Nucleic Acids Res. 2000;28(1):27–30. https://doi.org/10.1093/nar/28.1.27

```
Shell(f"apptainer exec {kofamscan_sif} kofamscan --help") # expect crash
```

It looks like we can't even get a help message with a naiive call to the tool. Let's check the documentation for some clues. Specifically, there is short section marked `## Usage` that will be helpful.

https://www.genome.jp/ftp/tools/kofam_scan/README.md

Ah, so the executable is `exec_annotation`.

```
Shell(f"apptainer exec {kofamscan_sif} exec_annotation --help")
```

Kofamscan needs pre-trained HMM models, which have been downloaded from these links.

- `ftp://ftp.genome.jp/pub/db/kofam/ko_list.gz`
- `ftp://ftp.genome.jp/pub/db/kofam/profiles.tar.gz`

Kofamscan will take a long time to run. You may want to try running it on the hypothetical ORFs from the bakta output at `LIB/"outputs/bakta.full/assembly.hypotheticals.faa` instead. Pre-computed outputs are available at `LIB/"outputs/kofamscan/kos.out"`

```
kofamscan_raw = Path("./kofamscan/kos.out")
Shell(f"""
mkdir -p ./kofamscan
apptainer exec {kofamscan_sif} \
    exec_annotation \
        --cpu={CPUS} --format=detail --no-report-unannotated \
        --profile={LIB}/kofamscan_db/profiles/prokaryote.hal \
        --ko-list={LIB}/kofamscan_db/ko_list \
        -o {kofamscan_raw} \
        {prodigal_aa}
""")
```

Kofamscan attempts to assigns KEGG orthology (KO) numbers to each ORF, linking them to rich KEGG databases which provide hierachical organization and metabolic context. We can use the provided utility to parse the results into a partial model of KEGG for local use.

```
from local.models.kegg_orthology import ParseKofamScanResults

model = ParseKofamScanResults(
    kofamscan_raw,
    LIB/"kofamscan_db/api_kegg.db",
    LIB/"kofamscan_db/brite.json",
)
```

```
model.Summary()
```

```
df = model.df
df = df[df.score > df.hmm_threshold] # retrieve only assignments that pass the model's adaptive c
print(df.shape)
df.head(2)
```

Let's take a peek at a KEGG database entry.

```
model.GetComponentModel("K06442")._raw
```

We can also look at where the assigned function sits in the BRITE functional hierarchy.

```
for lineage in model.GetLineage("K06442"):
    for k in lineage:
        print(model.GetNodeInfo(k))
```

`GetCategories` provides a shorthand to retrieve 2 useful levels from the BRITE heriarchy.

```
for l2, l1 in model.GetCategories("K06442"):
    for k in [l2, l1]:
        print(model.GetNodeInfo(k))
```

Just like what we did with transporters, let's visualize the abundance of functions aggregated by BRITE categories for each MAG. First, we will compile a count of each category by MAG.

```
rows = []
for _, row in df.iterrows():
    ko = row["ko"]
    orf = row["orf"]
    contig = "_".join(orf.split("_")[:-1])
    if contig in MAG1:
        mag = 1
    elif contig in MAG2:
        mag = 2
    else:
        mag = 0 # not in either MAG
    for category, _ in model.GetCategories(ko):
        rows.append((mag, category))
df_temp = pd.DataFrame(rows, columns=["mag", "category"])
```

Then, we can format it as a matrix for plotting.

```
# create some index lookup dictionaries for the matrix
c2i = {c: i for i, c in enumerate(df_temp["category"].unique())}    # category to inde
```

```
mag2i = {mag: i for i, mag in enumerate([1, 2])}                    # mag to index
mat = np.zeros(shape=(len(mag2i), len(c2i))) # MAG x categories
# iterate through the counts for each mag
# assigning them to the correct cell using the index lookup dictionaries
for _, row in df_temp.iterrows():
    mag = row["mag"]
    if mag == 0: continue
    i = mag2i[mag]
    j = c2i[row["category"]]
    mat[i, j] += 1

order = np.argsort(-mat.sum(axis=0)) # get the order of categories by size (largest -> smallest)
mat.sum(axis=1)
```

Plot the top 10 largest categories.

```
from local.figures.template import BaseFigure, ApplyTemplate, go

category_labels = np.array([model.GetNodeInfo(k)[1] for k in c2i.keys()])
K = 10
fig = BaseFigure()
for mag in [1, 2]:
    i = mag2i[mag]
    sign = -1 if mag == 1 else 1
    fig.add_trace(
        go.Bar(
            x = sign*mat[i, order[:K]],
            y = category_labels[order[:K]],
            orientation='h',
            name = f"MAG{mag}",
        )
    )

fig = ApplyTemplate(
    fig,
    layout=dict(
        legend_orientation ='h',
        barmode="relative",
        width=1000, height=400,
    ),
)
fig.show()
```

3.5.2 deepEC

https://quay.io/repository/hallamlab/external_deepec?tab=tags

```
# docker://quay.io/hallamlab/external_deepec:0.4.1
deepec_sif = LIB/"deepec.sif"
```

The DeepEC model uses convolutional neural networks, a deep learning architecture that is not far from HMM approach of Kofamscan. It too scans the protein sequence and attempts to predict function based on the "topology" or motif structure of the amino acid sequences. However, the model is able to make additional abstract decisions after the results of the initial scan in the form of additional model layers, further differentiating it from the sequence similarity approach [1]. As the name suggests, deepEC predicts enzyme commission (EC) numbers, which can be linked into the KEGG database, among others. It is straightforward to run.

1. Ryu JY, Kim HU, Lee SY. Deep learning enables high-quality and high-throughput prediction of enzyme commission numbers. Proc Natl Acad Sci. 2019;116(28):13996–4001. https://doi.org/10.1073/pnas.1821905116

```
Shell(f"""
apptainer exec {deepec_sif} \
    deepec \
        -p {CPUS} \
        -i {prodigal_aa} \
        -o ./deepec
""")
```

```
dfec = pd.read_csv("./deepec/DeepEC_Result.txt", sep="\t")
print(dfec.shape)
dfec.head(2)
```

How many ORFs did deepEC assign EC numbers to?

87 out of 7176

3.5.3 deepFRI

https://quay.io/repository/hallamlab/external_deepfri?tab=tags

```
# docker://quay.io/hallamlab/external_deepfri:1.0.1
deepfri_sif = LIB/"deepfri.sif"
```

So deep learning models are able to consider the protein sequence at a higher-level of abstraction, but what is it abstracting to? We often don't know the mental model of protein function that these approaches have constucted for themselves, but deepFRI provides and example that can ease us into the deep learning world. DeepFRI predicts function by first examining the potential structure of a protein sequence by predicting the pairwise distances between all amino acids in the form of a "contact map" [1]. The final output are Gene ontology (GO) numbers, which are an alternative to KEGG more common in medical settings.

1. Gligorijević V, Renfrew PD, Kosciolek T, Leman JK, Berenberg D, Vatanen T, et al. Structure-based protein function prediction using graph convolutional networks. Nat Commun. 2021;12(1):3168. https://doi.org/10.1038/s41467-021-23303-9

DeepFRI is quite tempermental. We must remove the * character indicating stop codons from the prodigal output.

```
prodigal_aa_no_stop = Path("./prodigal/orfs.no_stop.faa")
with open(prodigal_aa_no_stop, "w") as f:
    for e in SeqIO.parse(prodigal_aa, "fasta"):
        if e.seq[-1] == "*":
            e.seq = e.seq[:-1]
        SeqIO.write(e, f, "fasta")
```

We must also create and run it within a dedicated output folder, while using absolute paths. As it runs, it will dump files into the current directory despite us telling it otherwise. Try with the reduced set of hypotheticals (`{LIB}/"outputs/bakta.full/assembly.hypotheticals.faa"`) or use the pre-computed results at (`{LIB}/"outputs/deepfri/"`).

```
prodigal_aa_no_stop = Path("./prodigal/orfs.no_stop.faa")
deepfri_out = Path("./deepfri")
Shell(f"""
mkdir -p {deepfri_out}
cp {prodigal_aa_no_stop} {deepfri_out}/
cd {deepfri_out}
apptainer exec -B ./:/ws -W /ws ../{deepfri_sif} \
```

```
    deepfri \
        --ont mf bp ec \
        --fasta_fn /ws/{prodigal_aa_no_stop.name} \
        --output_fn_prefix /ws/chicken
""")
```

EC:

```
df = pd.read_csv(LIB/"outputs/deepfri/chicken_EC_predictions.csv", comment="#")
print(df.shape)
df.head(2)
```

GO molecular function

```
df = pd.read_csv(LIB/"outputs/deepfri/chicken_MF_predictions.csv", comment="#")
print(df.shape)
df.head(2)
```

GO biological process

```
df = pd.read_csv(LIB/"outputs/deepfri/chicken_BP_predictions.csv", comment="#")
print(df.shape)
df.head(2)
```

3.5.4 proteinBERT

https://quay.io/repository/hallamlab/external_proteinbert?tab=tags

```
# docker://quay.io/hallamlab/external_proteinbert:2024.03.28
proteinbert_sif = LIB/"proteinbert.sif"
```

ProteinBERT "reads" protein sequences like reading text. The core of its architecture is an attention mechanism that is part of what's called a "transformer", often used in natural language processing [1, 2]. Residue context is considered and pairwise importance is predicted to enable the model to "focus" on what it thinks are important motifs. The pairwise attention could be spatial distance, as in deepFRI, but is not strictly limited to this interpretation, hence increasing the abstraction capabilities of the model. The core transformer architecture is used in current large language models like ChatGPT. The only difference is that ChatGPT is much, much larger.

1. Vaswani A, Shazeer N, Parmar N, Uszkoreit J, Jones L, Gomez AN, et al. Attention is all you need. Curran Associates Inc.; 2017. (NIPS'17).
2. Brandes N, Ofer D, Peleg Y, Rappoport N, Linial M. ProteinBERT: a universal deep-learning model of protein sequence and function. Bioinformatics. 2022;38(8):2102–10. https://doi.org/10.1093/bioinformatics/btac020

Since the model did not have friendly interface, a CLI was constructed courtesy of the Hallam Lab.

```
apptainer exec {proteinbert_sif} \
    pbert run \
        --threads {CPUS} \
        -i {prodigal_aa} \
        -o protein_bert
```

The output of proteinBERT is an array of numbers for each ORF, representing their predicted function. These numbers can be used as coordinates in proteinBERT's mental model. We say each ORF is embedded in proteinBERT's functional space. The euclidean distance between each ORF's embedding represents predicted functional similarity.

```
df_index = pd.read_csv("./protein_bert/orfs.csv")
print(df_index.shape)
df_index.head(2)
```

ProteinBERT has 6 transformer layers, each of which reads then paraphrases the aminoacid sequence for the next. This information is stored in a 512-dimensional vector for each ORF, so the total embedding matrix is `N_orfs x (6x512)`.

```
mats = []
for batch in sorted(df_index.batch.unique()):
    mats.append(np.load(f"./protein_bert/orfs.{batch}.embedding.npy"))
mat = np.vstack(mats)
mat.shape
```

Examining the standard deviation of the embeddings for each layers reveals that the last layer has the most to say, so we will use it to reduce needed compute.

```
fig = BaseFigure()
fig.add_trace(
```

```
    go.Scatter(
        x=np.arange(mat.shape[0]),
        y=mat.std(axis=0),
        mode="markers",
        marker=dict(size=2),
    )
)
fig = ApplyTemplate(fig, default_xaxis=dict(dtick=512))
fig.show()
```

```
mat_last = mat[:, -512:]
mat_last.shape
```

Let's load the kofamscan results to provide KEGG landmarks for these embeddings.

```
LIB = Path("./lib")
kofamscan_raw = Path("./kofamscan/kos.out")
```

```
from local.models.kegg_orthology import ParseKofamScanResults

ko_model = ParseKofamScanResults(
    kofamscan_raw,
    LIB/"kofamscan_db/api_kegg.db",
    LIB/"kofamscan_db/brite.json",
)
```

```
df_kofam = ko_model.df
df_kofam = df_kofam[df_kofam["score"]>df_kofam["hmm_threshold"]]
print(df_kofam.shape)
df_kofam.head(2)
```

```
categories = {}
for _, row in df_kofam.iterrows():
    ko = row["ko"]
    orf = row["orf"]
    for brite2, brite1 in ko_model.GetCategories(ko):
        categories[brite2] = categories.get(brite2, set()) | {orf}

```

```
categories = sorted(categories.items(), key=lambda x: len(x[1]), reverse=True)
len(categories)
```

This assigns each ORF a category based on the BRITE hierarchy.

```
orf2cat = {}
for brite2, orfs in categories:
    for orf in orfs:
        if orf in orf2cat: continue
        orf2cat[orf] = brite2
cat2i = {cat: i+1 for i, (cat, _) in enumerate(categories)}

y_categories = []
for _, row in df_index.iterrows():
    orf = row["id"]
    brite2 = orf2cat.get(orf, "None")
    i = cat2i.get(brite2, 0)
    y_categories.append(i)
```

We can use UMAP to reduce the 512 dimensional embeddings to 2 dimensions for visualization and use the BRITE categories as a guide.

```
from umap import UMAP

model = UMAP(
    n_components=2, # reduce to x, y
    spread=5,       # spread out the points
    min_dist=0.1,   # reduce the maximum density of points
    metric="cosine",
)

emb = model.fit_transform(mat_last, y=y_categories)
```

Let's have a look.

```
from local.figures.template import BaseFigure, ApplyTemplate, go
from local.figures.colors import Palettes, COLORS

fig = BaseFigure()

```

```
K = 10
seen = set()
todo = []
for brite2, _ in categories[:K]:
    _filter = [orf2cat.get(orf) == brite2 for orf in df_index["id"]]
    seen |= set(orfs)
    todo.append((brite2, _filter))
_palette = Palettes.PLOTLY
for i, (name, _filter) in enumerate(todo):
    fig.add_trace(
        go.Scatter(
            name=ko_model.GetNodeInfo(name)[1],
            x=emb[_filter, 0],
            y=emb[_filter, 1],
            text=df_index[_filter]["id"],
            mode="markers",
            marker=dict(
                size=10,
                color=COLORS.TRANSPARENT,
                line=dict(
                    width=1,
                    color=_palette[i%len(_palette)].color_value,
                ),
            ),
        )
    )

# plot rest as gray dots
_filter = ~df_index["id"].isin(seen)
fig.add_trace(
    go.Scatter(
        name="other",
        x=emb[_filter, 0],
        y=emb[_filter, 1],
        text=df_index[_filter]["id"],
        mode="markers",
        marker=dict(
            size=5, opacity=0.1,
            color=COLORS.GRAY,
        ),
    )
)
no_axis = dict(linecolor=COLORS.TRANSPARENT, ticks=None, showticklabels=False)
fig = ApplyTemplate(
```

```
    fig,
    default_xaxis=no_axis, default_yaxis=no_axis,
    layout=dict(
        width=1000,
        height=700,
    ),
)
fig.show()
```

What are the gray dots?

These are ORFs not in the top K categories, including ORFs of unknown function.

3.6 gapseq

https://quay.io/repository/biocontainers/gapseq?tab=tags

```
# docker://quay.io/biocontainers/gapseq:1.4.0--h9ee0642_1
gapseqt_sif = LIB/"gapseq.sif"
```

So far, we have used various tools to predict the locations of genes and their functions in our metagenomic assembly. A portion of these functions were enzymatic reactions which convert between chemical compounds called metabolites. Others facilitate the transport of metabolites across membranes. Imagine reactions and transporters to be the edges of a network, chained together by common metabolites. The combined network would model the metabolic capacity encoded in the examined genetic sequences, drawing out possible pathways by which compounds are chemically transformed. Models from single genomes can indicate viability by possesing all necessary pathways to produce the constituents of biomass (nucleotides, amino acids, cofactors, lipids, etc.). Prediction errors that cause reactions to be missing create gaps in pathways, removing metabolic capacity, and potentially rendering the model non-viable. `gapseq` is a tool for filling these gaps from reference metabolic models by weighing candidate reactions against sequence evidence [1]. We will use gapseq to generate a model for each of the 2 MAGs by assuming viability on M9 minimal media.

1. Zimmermann J, Kaleta C, Waschina S. gapseq: informed prediction of bacterial metabolic pathways and reconstruction of accurate metabolic models. Genome Biol. 2021;22(1):81. https://doi.org/10.1186/s13059-021-02295-1

Can we run gapseq on the entire metagenomic assembly?

This would be a crude model of the community's metabolic capacity and can be used to detect possible pathways distributed across different members. A major source of error would be the lack of compartmentalization. The fact that a metabolite would have to be transported out of the cytosol of one cell and into the cytosol of another cell is not considered.

First, we separate the ORFs by MAG.

```
mag1_orfs, mag2_orfs = [], []
for e in SeqIO.parse(prodigal_aa, "fasta"):
    contig = "_".join(e.id.split("_")[:-1])
    if contig in MAG1:
        mag1_orfs.append(e)
    elif contig in MAG2:
        mag2_orfs.append(e)

mag1_aa, mag2_aa = "./prodigal/mag1.faa", "./prodigal/mag2.faa"
with open(mag1_aa, "w") as f:
    SeqIO.write(mag1_orfs, f, "fasta")
with open(mag2_aa, "w") as f:
    SeqIO.write(mag2_orfs, f, "fasta")
```

We will have to repeat the gapseq protocol for each MAG.

```
gapseq_mag = "mag1"
# gapseq_mag = "mag2" # repeat with mag2
aa_fasta = f"./prodigal/{gapseq_mag}.faa"
out_dir = f"./gapseq/{gapseq_mag}"
```

Candidate reactions are searched for by gapseq *one pathway at a time*. This makes multiple (thousands) of redundant calls to the alignment tool which makes the process incredibly slow. This is because gapseq was intended to be an aid to manual curation and not a fully automated tool. Pre-computed outputs are available at `{LIB}/outputs/gapseq`. A quick trial with only a single pathway (`glycolysis`) is possible.

- `-p keywords such as pathways or subsystems (for example amino,nucl,cofactor,carbo,polyamine)`

- -l Select the pathway database (MetaCyc, KEGG, SEED, all; default: metacyc,custom)
- -K Number of threads for sequence alignments.
- -O Force offline mode
- -f Path to directory, where output files will be saved

```
gapseq_p = "glycolysis"
# gapseq_p = "all" # takes about 4 hours!!
Shell(f"""
mkdir -p {out_dir}
apptainer exec {gapseq_sif} \
    gapseq find -p {gapseq_p} -l KEGG -K {CPU} -O -f {out_dir} {aa_fasta} \
        > {out_dir}/find.out 2> {out_dir}/find.err
""")
```

> and 2>

> redirects the standard output and 2> redirects the standard error to a files for logging.

Transporters are faster, but still takes several minutes. Please use pre-computed results.

```
Shell(f"""
mkdir -p {out_dir}
apptainer exec {gapseq_sif} \
    gapseq find-transport -K {CPU} -f {out_dir} {aa_fasta} \
        > {out_dir}/findtr.out 2> {out_dir}/findtr.err
""")
```

Construct a draft model using candidate reactions, transporters, and pathways.

```
Shell(f"""
apptainer exec {gapseq_sif} \
    gapseq draft \
        -r {out_dir}/{gapseq_mag}-{gapseq_p}-Reactions.tbl \
        -p {out_dir}/{gapseq_mag}-{gapseq_p}-Pathways.tbl \
        -t {out_dir}/{gapseq_mag}-Transporter.tbl \
        -c {aa_fasta} \
        -f {out_dir} \
        > {out_dir}/draft.out 2> {out_dir}/draft.err
""")
```

Add additional reactions to enable growth on M9 minimal media.

```
Shell(f"""
apptainer exec {gapseq_sif} \
    gapseq fill --quick.gf \
        -m {out_dir}/{gapseq_mag}-draft.RDS \
        -c {out_dir}/{gapseq_mag}-rxnWeights.RDS \
        -g {out_dir}/{gapseq_mag}-rxnXgenes.RDS \
        -n {LIB}/gapseq/m9.csv \
        --output.dir {out_dir} \
        > {out_dir}/fill.out 2> {out_dir}/fill.err
""")
```

We can take a peek at the observed completion of each pathway. The model will be examined further in the next lab.

```
Shell(f"""
mag = 1
dfp = pd.read_csv(f"./gapseq/mag{mag}/mag{mag}-all-Pathways.tbl", sep="\t", comment="#"
dfp.sort_values("Completeness", ascending=False, inplace=True)
dfp
""")
```

The following code will generate a link to the KEGG pathway viewer with found reactions coloured. Pick a few and have a look.

```
from urllib.parse import quote
# https://www.genome.jp/kegg/webapp/color_url.html
pathway = "map00970"
color = "#00cc96"

reactions = dfp[dfp["ID"] == pathway]["ReactionsFound"].tolist()
reactions = [row.strip().split(" ") for row in reactions]
reactions = [f"{r} {color}" for row in reactions for r in row]
q = "\n".join(reactions)
url = f"https://www.kegg.jp/kegg-bin/show_pathway?map={pathway}&multi_query={quote(q)}"
url
```

This concludes the lab!

Chapter 4

Module 4 Lab: Visualization and Finding Functional Significance

In this lab, we will work through typical transcriptiomic analyses to find significantly expressed genes and gene sets. Metagenomic reads will be used in place of transcriptomic data, so we will be examining abundance instead of expression, but the analyses shown will remain unchanged. For the sake of consistency, the abundance data will be referred to "expression" for the rest of the lab.

4.1 Setting up the workspace

The following procedure is near identical to that of lab3. Please refer to the setup sectiom of lab3 for justifications and context. We will need to perform the following steps:

1. Open an additional port via SSH for the new jupyter notebook
2. Create a new workspace directory for this lab
3. Unpack and start the container
4. Open the jupyter notebook in your browser

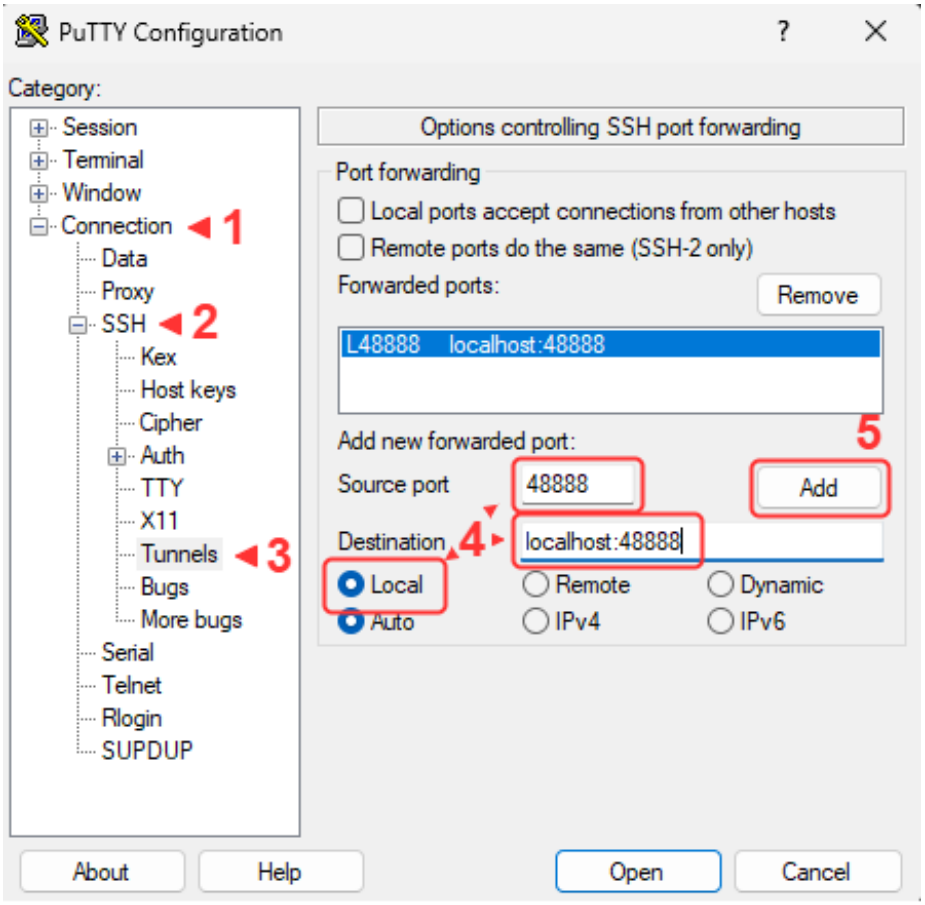
4.1.1 SSH port forwarding

4.1.1.1 Terminal (Unix/MacOS)

Add `-L 48888:localhost:48888` to the SSH command you use to connect to the server. For example:

```
ssh -L 48888:localhost:48888 ubuntu@${your_number}.uhn-hpc.ca -i ${path_to_your_key}
```

4.1.1.2 Putty (Windows)



4.1.2 Workspace

Create a new workspace in your home directory.

```
cd ~
mkdir amb_lab4
cd amb_lab4
```

Link the resource folder for easy access.

```
ln -s ~/CourseData/MIC_data/amb_module4/ ./lib
```

4.1.3 Start container

Unpack the main container for this lab.

```
apptainer exec ./lib/amb_lab4.sif unpack
```

The following will dedicate the current terminal to running the juptyer notebook on port 48888. You may want to connect another terminal to maintain terminal access.

```
./start_lab
```

4.1.4 Jupyter

JupyterLab should now be accessible at http://localhost:48888/lab. Create a new python notebook and import dependencies.

```
import os
import pandas as pd
from pathlib import Path

LIB = Path("./lib")
CPUS = 4 # your workshop instance has been alloted 4 cores
```

All software tools are packaged in the main container, so we can use `os.system("...")` instead of `Shell`.

```
os.system(f"""
pwd -P
echo {LIB}
""")
```

4.2 Overview

To understand the objective of this lab, let's first have a look at the metadata.

```
df_meta = pd.read_csv(LIB/"inputs/metagenome_metadata.txt", sep="\t")
print(df_meta.shape)
df_meta.head(2)
```

How many samples and how many categories?

We have 33 samples divided into "Bacteroides" and "Non-Bacteroides" groups.

We also have a set of predicted genes.

```
from Bio import SeqIO

count = 0
for i, sequence in enumerate(SeqIO.parse(GENES, "fasta")):
    count += 1
count
```

The objective of this lab is to identify and visualize the pathways who's gene expression patterns correlate with the Bacteroides vs Non-Bacteroides grouping. To do this we will:

1. Use salmon to obtain a count matrix representing the expression level of each gene in each sample.
2. Use deseq2 to identify significantly expressed genes and generate a null distribution for gene set enrichment analysis.
3. Perform Gene set enrichment analysis (GSEA) to identify significantly enriched pathways.
4. Visualize the metabolic network in cytoscape with expression information.

4.3 Salmon

The alignment of transcriptomic reads to genes is often ambiguous. A single read can map to multiple genes and the length of genes compared to reads means that multiple reads will map to different regions of the same gene. Salmon uses probabilistic and statistical models to quantify the expression of each gene while also addressing potential biases [1]. We will use salmon to obtain a count matrix for deseq2.

1. Patro R, Duggal G, Love MI, Irizarry RA, Kingsford C. Salmon provides fast and bias-aware quantification of transcript expression. Nat Methods. 2017;14(4):417–9. https://doi.org/10.1038/nmeth.4197

Let's have a look at the help message.

```
os.system(f"salmon --help")
```

```
os.system(f"salmon index --help")
```

```
os.system(f"salmon quant --help")
```

We will need to index the reference genes before performing quantification.

```
salmon_dir = Path("./salmon")
GENES = LIB/"inputs/lab3_orfs.fna"
salmon_index = salmon_dir/"index"
os.system(f"salmon index -t {GENES} -i {salmon_index}")
```

Quantification will take about 6 minutes. `tqdm` is a used to display a progress bar. Precomputed outputs are at `./lib/outputs/salmon/`.

```
from tqdm import tqdm

reads_dir = LIB/"inputs/reads"
samples = list(reads_dir.iterdir())
for read_file in tqdm(samples, total=len(samples)):
    srr = read_file.name.split("_")[0]
    out = salmon_dir/f"{srr}"
    os.system(f"""
        mkdir -p {out}
        salmon quant \
            -l A \
            -p {CPUS} \
            -i {salmon_index} \
            -r {read_file} \
            -o {out} \
            >{out}/salmon.log 2>{out}/salmon.err
    """)
```

Salmon requires information about the reads' library type, such as stranded, unstranded, facing, paired-end, etc. `-l A` will tell Salmon to infer the type from the first 1000 reads. More information is here.

Let's have a look at one of the results. The file we want is `quant.sf`.

```
for srr in salmon_dir.iterdir():
    if not srr.name.startswith("SRR"): continue
    df = pd.read_csv(srr/"quant.sf", sep="\t")
    break
df
```

The `TPM` (transcipts per million) column is useful on its own, but deseq2 prefers raw counts, so we will use the `NumReads` column. Let's create the count matrix of genes by samples. First, create an index of genes.

```
gene2i = {}
for i, sequence in enumerate(SeqIO.parse(GENES, "fasta")):
    gene2i[sequence.id] = i
len(gene2i)
```

Now, we will load the counts for each sample into the matrix by matching the gene name using the index.

What are the expected dimensions of the count matrix?

Genes x Samples, so 3298 x 33.

```
mtx = np.zeros(shape=(len(gene2i), len(samples)), dtype=np.int32) # genes x samples
for i, srr in tqdm(enumerate(df_meta["sample_name"]), total=len(df_meta)):
    df = pd.read_csv(salmon_dir/f"{srr}/quant.sf", sep="\t")

    gene_indexes = [gene2i[row["Name"]] for _, row in df.iterrows()]
    counts = df["NumReads"]
    mtx[gene_indexes, i] = counts
mtx.shape
```

Finally, we will label the rows and columns of the matrix with gene IDs and sample names before saving it to a file.

```
gene_ids = list(gene2i.keys())
sample_ids = list(df_meta["sample_name"])

df = pd.DataFrame(mtx, columns=sample_ids)
df["gene_id"] = gene_ids
df = df.set_index("gene_id")
print(df.shape)
df.to_csv(salmon_dir/"counts.mtx")
df.head(2)
```

We are now ready for deseq2.

4.4 DESeq2

DESeq2 is a statistical package for R that is used to analyze count data from RNA-Seq experiments [1]. It uses a negative binomial distribution to model count data, taking into account large biological diversity and technical noise (overdispersion). We will use the statistical models and tools provided by DESeq2 to calculate fold changes and estimate significance. The main purpose of DESeq2 in this lab is to generate the needed data for GSEA, which includes estimating the correlation between gene expression and the experimental groups: Bacteroides and Non-Bacteroides. We will also generate a null distribution for GSEA by permuting the sample labels.

1. Patro R, Duggal G, Love MI, Irizarry RA, Kingsford C. Salmon provides fast and bias-aware quantification of transcript expression. Nat Methods. 2017;14(4):417–9. https://doi.org/10.1038/nmeth.4197

Transistion to an R notebook. http://localhost:48888/lab

```
library(DESeq2)
library(readr)
```

load the count matrix

```
cts <- as.matrix(read.csv(
    "./salmon/counts.mtx",
    row.names = "gene_id"
))
```

load the sample metadata

```
coldata <- read.csv(
    "./lib/inputs/metagenome_metadata.txt",
    sep = "\t",
)
coldata$grouping
```

Create a Deseq dataset object.

```
dds <- DESeqDataSetFromMatrix(
  countData = cts,
  colData = coldata,
  design = ~ grouping
)
dds
```

This is how we would run DESeq2, but don't run it just yet.

```
dds <- DESeq(dds)
```

We can cache the results to an "R data serialized" (RDS) to prevent having to re-run this analysis when restarting the notebook.

```
deseq_save <- "./deseq2/dds.rds"
if (!file.exists(deseq_save)) {
  dds <- DESeq(dds)
  saveRDS(dds, deseq_save)
} else {
  dds <- readRDS(deseq_save)
}
```

Extract the results and save to a dataframe. This will contain log2 fold changes and p-values for each gene.

```
res <- results(dds, contrast = c("grouping", "Bacteroides", "Non-Bacteroides"))
results_table <- as.data.frame(res)
write.csv(results_table, "./deseq2/results.csv", row.names = TRUE)
```

Next, we will compute the correlation between gene expression and the experimental groups and generate a null distribution for GSEA by permuting the sample labels. This effectively randomizes the assignemnt of samples to each experimental condition, but preserves any relationships between genes within a single sample. When the number of samples is low, genes can be randomly assigned to gene sets instead, but this fails to take into account gene-gene relationships.

What is the minimum number of samples to simulate a null distribution with a precision of at least 0.05?

4 samples ($4! = 24$, $1/24$ is about 0.04) This will be explored further in the next section on GSEA.

We will use the suggested variance stablized transform (VST) to obtain normalized counts. VST removes the dependence of variance on the mean, which limits the high variance of the logarithm of counts when the mean is low. (more info here).

```
vst <- vst(dds, blind = FALSE)
vst_counts <- assay(vst)
```

The human-readable class labels need to be converted to more machine-friendly ordinal labels.

```
coldata$grouping
class_labels_ordinal <- 1 - (as.numeric(factor(coldata$grouping)) - 1)
class_labels_ordinal
```

We will use the point biserial correlation, a special case of the Pearson correlation designed to compare a continuous variable with a binary variable. Alternatively the biserial correlation achieves nearly the same, but assumes that the binary variable is the result of thresholding a continuous variable. For example, it would be appropriate to use the point biserial correlation for a coin toss, while the biserial correlation would be more appropriate for pass/fails from percentage grades. Practically, the biserial correlation also requires the relative propotions of each class. While there likely is a genetic spectrum of bacteria spanning from Bacteroides to Non-Bacteroides, we don't have any information on this spectrum or the genetic threshold at which an organism stops being a Bacteroides.

```r
point_biserial <- function(expression, classes) {
  if (sd(expression) == 0) return(NA)
  return(cor(expression, classes, method = "pearson"))
}
```

Estimate the observed correlation.

```r
gene_correlations <- apply(vst_counts, 1, point_biserial, classes = class_labels_ordina
gene_correlations <- as.data.frame(gene_correlations)
write.csv(gene_correlations, "./deseq2/gene_correlations.csv", row.names = TRUE)
```

To generate the null distribution, we will be taking many samples (200). The following libraries will enable us to parallelize the computation.

```r
library(pbapply)
library(parallel)
```

Since this is computationally expensive, we will use the caching technique shown earlier.

```r
null_save <- "./deseq2/null_distr.csv"

if (!file.exists(null_save)) {

  tryCatch(             # if anything goes wrong, we must run the "finally" block
    {
      cl <- makeCluster(4) # make a cluster with 4 workers
      # make these variables available to each worker
      clusterExport(cl, c("vst_counts", "point_biserial", "class_labels_ordinal"))

      n_permutations <- 200
      permutation_results <- pbsapply(
        1:n_permutations,   # iterate 200 times
        function(i) {       # and run this function each time
          # Shuffle labels
          n <- length(class_labels_ordinal)
          shuffled_labels <- sample(class_labels_ordinal, size = n, replace = FALSE)

          # Calculate correlations for all genes
          return(apply(vst_counts, 1, point_biserial, classes = shuffled_labels))
```

```
        },
        cl = cl # use workers to compute in parallel
      )
    },
    finally = {         # no matter what happens
      stopCluster(cl)   # remember to stop the cluster
    }
  )

  # cache results as a dataframe
  null_distr <- data.frame(
    sample = permutation_results,
    stringsAsFactors = FALSE
  )
  write.csv(
    null_distr,
    null_save,
    row.names = TRUE
  )
} else {
  null_distr <- read.csv(null_save)
}

dim(null_distr)
```

4.5 Gene set enrichment analysis (GSEA)

Back to python.

GSEA ranks genes by their correlation with experimental groups, then examines the overrepresentation of gene sets in highly ranked genes by calculating an enrichment score (ES) [1]. The observed ES compared to the null distribution of enrichment scores is then used to estimate the significance of the observed ES. To apply GSEA, we need:

- N samples where M continuous features are measured (eg. N transcriptomic samples for M genes, N metagenomes of M members, N pictures of M pixels, etc.)
- Groupings (overlap permitted) of these M features to test

Despite GSEA being widely applicable, implementations are overly specific to human transcriptomics, rendering them difficult to adapt to even adjacent domains. Fortunately, the technique itself is rather straightforward to implement.

1. Patro R, Duggal G, Love MI, Irizarry RA, Kingsford C. Salmon provides fast and bias-aware quantification of transcript expression. Nat Methods. 2017;14(4):417–9. https://doi.org/10.1038/nmeth.4197

We will use KEGG pathways as our gene sets. These were predicted using Kofamscan from the previous lab.

```
from local.models.kegg_orthology import ParseKofamScanResults

kegg_model = ParseKofamScanResults(
    LIB/"inputs/kofamscan/kos.out",
    LIB/"kofamscan_db/api_kegg.db",
    LIB/"kofamscan_db/brite.json",
)
```

We will find the pathways for each gene by tracing: `gene -> function -> reaction -> pathway`.

```
function2genes = {}
for _, r in kegg_model.df.iterrows():
    ko = r["ko"]
    orf = r["orf"]
    function2genes[ko] = function2genes.get(ko, set())|{orf}

reactions2functions = {}
for ko in function2genes:
    meta = kegg_model.Functions[ko]
    for r in meta.reactions:
        reactions2functions[r] = reactions2functions.get(r, set())|{ko}

pathway2reactions = {}
for r in reactions2functions:
    meta = kegg_model.Reactions[r]
    for p in meta.pathways:
        pathway2reactions[p] = pathway2reactions.get(p, set())|{r}

gene_sets = {} # collect the genes for each pathway
for p, reactions in pathway2reactions.items():
    genes = set()
    for r in reactions:
        functions = reactions2functions[r]
        for f in functions:
            genes |= function2genes[f]
    gene_sets[p] = genes
```

Load correlations from DESeq2.

```
df_null = pd.read_csv(deseq_dir/"null_distr.csv", index_col=0)
print(df_null.shape)
df_null.head(2)
```

```
df_corr = pd.read_csv(deseq_dir/"gene_correlations.csv", index_col=0)
print(df_corr.shape)
df_corr.head(2)
```

Next we will calculate the ES and p-value for an example pathway `rn00983`.

```
gene_set = gene_sets["rn00983"]
kegg_model.Pathways["rn00983"].name
```

First, create a matrix of permutations x genes, only taking those that are in gene sets.

```
enzymes = {x for g in gene_sets.values() for x in g}
def _only_enzymes(df: pd.DataFrame):
    return df.loc[df.index.isin(enzymes)]

corr = _only_enzymes(df_corr).to_numpy()
permuted = _only_enzymes(df_null).to_numpy()
mat = np.hstack([corr, permuted]) # true correlations is the first permutation
mat.shape
```

Rank order the genes in each permutation by their correlation.

```
mat_index = np.argsort(-mat.T, stable=True)
mat_sorted = np.zeros_like(mat)
for i, order in enumerate(mat_index):
    mat_sorted[:, i] = mat[:, i][order]
mat_sorted.shape
```

To calculate the ES, a running sum is calculated by scanning down the ranked list of genes, increasing the score for genes that are in the gene set and decreasing it otherwise. Genes in the gene set are called “hits” and those not in the gene set are called “misses”. The formulation for the running sum of hits (P_hit) at

position i for gene set S is given in the paper. Don't look at all of it at once, we will break it down.

First, note that the summations only consider genes in the gene set S.

$$P_{\text{hit}}(S,i) = \sum_{\substack{g_j \in S \\ j \leq i}} \frac{|r_j|^p}{N_R}, \quad \text{where } N_R = \sum_{g_j \in S} |r_j|^p$$

Let's create a mask to select only genes in S.

```
labels = _only_enzymes(df_corr).index.to_numpy()
S = np.array([k in gene_set for k in self.labels], dtype=int) # g in S
S = S[mat_index.T] # same order as mat
```

Next, have a look at NR. rj is simply the correlation values of each gene. p is a hyperparamter set to 1. Let's calculate NR.

```
p = 1
corr = np.pow(np.abs(self.mat_sorted), p)
N_R = np.sum(corr*S, axis=0)
```

We now need to combine the repeated |rj|p term in the numerator with NR in the denominator and cumulatively sum up to each gene. `np.cumsum()` does this for us.

```
P_hit = corr/N_R
P_hit = P_hit*S # g_j in S
P_hit = np.cumsum(P_hit, axis=0)
```

Have a look.

```
from local.figures.template import BaseFigure, ApplyTemplate, go
fig = BaseFigure()
fig.add_trace(go.Scatter(
    x = np.arange(len(P_hit[:, 0])),
    y = P_hit[:, 0],
))
fig = ApplyTemplate(fig)
fig.show()
```

For P_miss, we cumulatively sum the fraction of misses.

$$P_{\text{miss}}(S, i) = \sum_{\substack{g_j \notin S \\ j \leq i}} \frac{1}{(N - N_H)}.$$

```
P_miss = 1/(len(labels) - len(gene_set))
_mask = 1-S.T # invert hits for misses
P_miss = np.ones_like(labels)*_mask*P_miss
P_miss = np.cumsum(P_miss.T, axis=0)
```

That's it, have a look.

```
fig = BaseFigure()
fig.add_trace(go.Scatter(
    x = np.arange(len(P_miss[:, 0])),
    y = -P_miss[:, 0],
))
fig = ApplyTemplate(fig)
fig.show()
```

The running statistic is the difference between P_hit and P_miss. Finally, the ES is the maximum magnitude of the running statistic.

```
S_distr = (P_hit - P_miss).T # running statistic

ES_pos = S_distr.max(axis=1)
ES_neg = S_distr.min(axis=1)
ES_candidates = np.vstack([ES_pos, ES_neg]).T       # candidate is either the positive or negati
to_take = np.argmax(np.abs(ES_candidates), axis=1)  # we must select for each permutation
ES = (ES_neg*to_take) + (ES_pos*(1-to_take))
```

Have a look.

```
fig = BaseFigure()
fig.add_trace(go.Scatter(
    x = np.arange(len(S_distr[0])),
    y = S_distr[0],
))
fig = ApplyTemplate(fig)
fig.show()
```

We can now split the score and null distribution and estimate the p-value.

```
score, null = ES[0], ES[1:]
pvalue = np.sum(null >= score) / null.shape[0]
if pvalue == 0:
    pvalue = 1 / null.shape[0]
pvalue
```

Let's clean things up a bit. We will return a `dataclass`, which is just a convient way to group many variables together. The procedure above is also attached to the class object `Gsea`, but otherwise unchanged.

```
from dataclasses import dataclass

@dataclass
class GseaResult:
    set_name: str
    set_size: int
    score: float
    pvalue: float
    hits: np.ndarray
    running_sum_statistic: np.ndarray

class Gsea:
    def __init__(self, labels: np.ndarray, corr: np.ndarray, permuted: np.ndarray) -> N
        self.labels = labels
        mat = np.hstack([corr, permuted])
        self.mat_index = np.argsort(-mat.T, stable=True)
        self.mat_sorted = np.zeros_like(mat)
        for i, order in enumerate(self.mat_index):
            self.mat_sorted[:, i] = mat[:, i][order]

    def run(self, name: str, gene_set: set, p=1):
        S = np.array([k in gene_set for k in self.labels], dtype=int)
```

```
        if S.sum() == 0: return
        S = S[self.mat_index.T]

        corr = np.pow(np.abs(self.mat_sorted), p)
        N_R = np.sum(corr*S, axis=0)

        P_hit = corr/N_R
        P_hit = P_hit*S # g_j in S
        P_hit = np.cumsum(P_hit, axis=0)

        P_miss = 1/(len(self.labels) - len(gene_set))
        _mask = 1-S.T # invert hits for misses
        P_miss = np.ones_like(self.labels)*_mask*P_miss
        P_miss = np.cumsum(P_miss.T, axis=0)

        S_distr = (P_hit - P_miss).T

        ES_pos = S_distr.max(axis=1)
        ES_neg = S_distr.min(axis=1)
        ES_candidates = np.vstack([ES_pos, ES_neg]).T
        to_take = np.argmax(np.abs(ES_candidates), axis=1)
        ES = (ES_neg*to_take) + (ES_pos*(1-to_take))
        score, null = ES[0], ES[1:]
        pvalue = np.sum(null >= score) / null.shape[0]
        if pvalue == 0:
            pvalue = 1 / null.shape[0]

        return GseaResult(
            set_name=name,
            set_size=len(gene_set),
            score=score,
            pvalue=pvalue,
            hits=S.T[0],
            running_sum_statistic=S_distr[0],
        )

def _only_enzymes(df: pd.DataFrame):
    return df.loc[df.index.isin(enzymes)]
_df_corr_e = _only_enzymes(df_corr)
gsea = Gsea(
    labels=_df_corr_e.index.to_numpy(),
    corr=_df_corr_e.to_numpy(),
    permuted=_only_enzymes(df_null).to_numpy(),
)
```

We can now run GSEA for all pathways.

```
results: list[GseaResult] = []
for k, s in tqdm(gene_sets.items(), total=len(gene_sets)):
    res = gsea.run(k, s)
    results.append(res)
results_map = {r.set_name: r for r in results}
```

And print out some relavent results.

```
results = sorted(results, key=lambda x: x.pvalue)
for r in results:
    if r.pvalue > 0.05: continue
    meta = kegg_model.Pathways[r.set_name]
    print(f"{r.pvalue:.2} {r.score:.2} {r.hits.sum()} {r.set_name}:{meta.name}")
```

Let's save the results to a table.

```
rows = []
for res in results:
    rows.append((
        res.set_name,
        res.pvalue,
        res.score,
    ))

df_gsea = pd.DataFrame(rows, columns=["set_name", "pvalue", "score"])
Path("./gsea").mkdir(exist_ok=True, parents=True)
df_gsea.to_csv("./gsea/results.csv", index=False)
```

The nice GSEA plot is composed of 3 elements. First, we indicate the rank of genes within the selected gene set.

```
res = results_map["rn00983"]

fig = BaseFigure()
_f = res.hits.astype(bool)
fig.add_trace(
    go.Scatter(
        x = np.arange(gsea.labels.shape[0])[_f],
        y = np.zeros_like(gsea.labels)[_f],
```

```
        mode = "markers",
        marker=dict(
            size=25,
            color=COLORS.TRANSPARENT,
            symbol="line-ns",
            line=dict(width=1),
        ),
        showlegend=False,
    ),
)
fig = ApplyTemplate(fig)
fig.show()
```

Next, we draw out the magnitudes of the correlations.

```
fig = BaseFigure()
fig.add_trace(
    go.Bar(
        x = np.arange(gsea.labels.shape[0]),
        y = gsea.mat_sorted.T[0],
        showlegend=False,
        marker=dict(
            color=COLORS.BLACK,
            line=dict(width=0),
        ),
    ),
)
fig = ApplyTemplate(fig)
fig.show()
```

And draw out the running sum statistic for the ES.

```
fig = BaseFigure()
fig.add_trace(
    go.Scatter(
        x = np.arange(gsea.labels.shape[0]),
        y = res.running_sum_statistic,
        mode = "lines",
        showlegend=False,
        line=dict(
            color = COLORS.BLACK,
        ),
```

```
    ),
)
fig = ApplyTemplate(fig)
fig.show()
```

Finally, we stack the 3 together to replicate the figures used in the GSEA paper.

```
def draw_gsea(res: GseaResult, gsea: Gsea):
    k = res.set_name
    fig = BaseFigure(
        shape=(1, 3),                    # 3 subplots, vertically stacked
        row_heights=[0.2, 0.2, 0.6],     # spaced like so
        vertical_spacing=0.05,
    )

    _f = res.hits.astype(bool)
    fig.add_trace(
        go.Scatter(
            x = np.arange(gsea.labels.shape[0])[_f],
            y = np.zeros_like(gsea.labels)[_f],
            mode = "markers",
            marker=dict(
                size=25,
                color=COLORS.TRANSPARENT,
                symbol="line-ns",
                line=dict(width=1),
            ),
            showlegend=False,
        ),
        col=1, row=1,
    )

    fig.add_trace(
        go.Bar(
            x = np.arange(gsea.labels.shape[0]),
            y = gsea.mat_sorted.T[0],
            showlegend=False,
            marker=dict(
                color=COLORS.BLACK,
                line=dict(width=0),
            ),
        ),
        col=1, row=2,
```

```
    )

    fig.add_trace(
        go.Scatter(
            x = np.arange(gsea.labels.shape[0]),
            y = res.running_sum_statistic,
            mode = "lines",
            showlegend=False,
            line=dict(
                color = COLORS.BLACK,
            ),
        ),
        col=1, row=3,
    )

    hidden = dict(ticks=None, linecolor=COLORS.TRANSPARENT, showticklabels=False)
    fig = ApplyTemplate(
        fig,
        default_xaxis=hidden,
        default_yaxis=hidden,
        layout=dict(
            width=600, height=400,
            margin=dict(l=15, r=15, t=50, b=15),
            title=dict(text=f"(p={res.pvalue}) {k}:{kegg_model.Pathways[k].name}", font_size=18,
        ),
    )
    fig.show()

draw_gsea(results_map["rn00983"], gsea)
```

If you have time, use the `draw_gsea` function to explore other pathways.

4.6 Cytoscape

Cytoscape is a powerful visualization environment for biological networks [1, 2]. We will combine the previously generated gapseq metabolic model with the gene expression data from this lab for visualization.

1. Shannon P, Markiel A, Ozier O, Baliga NS, Wang JT, Ramage D, et al. Cytoscape: A Software Environment for Integrated Models of Biomolecular Interaction Networks. Genome Res. 2003;13(11):2498–504. https://doi.org/10.1101/gr.1239303

2. Cline MS, Smoot M, Cerami E, Kuchinsky A, Landys N, Workman C, et al. Integration of biological networks and gene expression data using Cytoscape. Nat Protoc. 2007;2(10):2366–82. https://doi.org/10.1038/nprot.2007.324

The gapseq model is saved in systems biology markup language (SBML) format, which is a standard for representing biochemical networks. We will load the model and convert it to a python dictionary for easy usage.

```
import xmltodict

MAG = "mag1"
# MAG = "mag2"  # Uncomment to use the second MAG
sbml = LIB/f"inputs/gapseq/{MAG}/{MAG}.xml"
with open(sbml, "r") as f:
    gs_model = xmltodict.parse(f.read())
```

It will be helpful to examine the raw file itself for orientation. Much of the information we need is burried in a matryoshka doll-like structure of nested objects. Our first objective is to extract the metabolic network consisting of reactions connected to compounds. We will use both reactions and compounds as nodes in the network, with edges connecting reactions to compounds. As we traverse the network, we will alternate between reactions and compounds.

```
# open the matryoshka data structure and find the information we need
def find(d, k):
    if k in d:  # found it
        return [d[k]]
    candidates = []
    for v in d.values(): # we didn't find it, so go deeper
        if isinstance(v, dict):
            candidates += find(v, k)
        elif isinstance(v, list):
            for x in v:
                candidates += find(x, k)
    return candidates

# retrieve list of reactions
raw_reactions = gs_model["sbml"]["model"]["listOfReactions"]["reaction"]
# store some useful information
names = {}      # node id to reaction or compound names
r2genes = {}    # reactions to genes
c2r = {}        # compound to reactions
```

```
r2c = {}        # reactions to compounds

# collect reactions and their associated genes and compounds
for r in raw_reactions:
    rxn_id = r["@id"]
    names[rxn_id] = r.get("@name", rxn_id)
    for gene_id in find(r, "@fbc:geneProduct"):
        r2genes[rxn_id] = r2genes.get(rxn_id, [])+[gene_id]
    compounds = find(r, "@species")
    for cpd_id in compounds:
        c2r[cpd_id] = c2r.get(cpd_id, [])+[rxn_id]
        r2c[rxn_id] = r2c.get(rxn_id, [])+[cpd_id]

# collect compound names
raw_compounds = gs_model["sbml"]["model"]["listOfSpecies"]["species"]
for c in raw_compounds:
    cpd_id = c["@id"]
    names[cpd_id] = c.get("@name", cpd_id)

len(r2c), len(c2r)
```

Certain compounds, such as H+, ATP, NADH, are ubiquitous and reactions connected by these compounds may not be very related. We can generate weights for edges to filter out low-information connections using the "promiscuity" of each compound.

```
compound_promiscuity = {k: len(v) for k, v in c2r.items()}

names["M_cpd00011_c0"], compound_promiscuity["M_cpd00011_c0"]
```

The positions of network nodes on the 2D canvas of Cytoscape is called a graph projection. It would be useful if similar or biochemically adjacent reactions were positioned close to each other in this projection. We can achieve this by using UMAP with pairwise distances calculated from the shared compounds between 2 reactions, weighted by the promiscuity of each compound. In other words, we will use weighted jaccard similarity to obtaina graph projection.

```
import numpy as np
from tqdm import tqdm

# helper function ot get connected reactions that share compounds
def adjacent_reactions(rxn):
```

```
    found_reactions = []
    for c in r2c[rxn]:
        for r in c2r[c]:
            if r == rxn: continue
            found_reactions.append(r)
    return found_reactions

# we scale the promiscuity values to be between 1 and 0 using a sigmoid function
# the more promiscuous a compound is, the less weight it has in the similarity calcula
_promiscuity_values = np.array([compound_promiscuity[c] for c in c2r.keys()])
b = 5
_scaled_promiscuity = 0.5*np.e/(1+np.exp((_cproms-b-1)/b)) # sigmoid, dist=0.5 at 8, n
compound_scaled_weights = {c: w for c, w in zip(c2r.keys(), _scaled_promiscuity)}

# calculate the weighted jaccard similarity between reactions
# jaccard = shared / total
reaction_similarity = {}
for r1 in tqdm(r2c, total=len(r2c)):
    for r2 in adjacent_reactions(r1):
        k = (r1, r2) if r1 < r2 else (r2, r1)
        if k in reaction_similarity: continue
        c1 = set(r2c[r1])
        c2 = set(r2c[r2])
        cintersect = c1&c2
        cunion = c1|c2
        total_weight = sum(compound_scaled_weights[c] for c in cunion)
        intersect_weight = sum(compound_scaled_weights[c] for c in cintersect)
        weighted_jaccard_sim = intersect_weight/total_weight
        if weighted_jaccard_sim < 1e-6: continue
        reaction_similarity[k] = weighted_jaccard_sim

min(reaction_similarity.values()), max(reaction_similarity.values()), len(reaction_sim
```

Many reactions connected by promiscuous compounds are given near 0 weights.

```
from local.figures.template import BaseFigure, ApplyTemplate, go
fig = BaseFigure()
fig.add_trace(
    go.Histogram(
        x=list(reaction_similarity.values()),
        nbinsx=100,
    )
)
```

```
fig = ApplyTemplate(fig, layout=dict(height=200))
fig.show()
```

Run UMAP to obtain the network projection.

```
from umap import UMAP

pdist = np.ones((len(r2c), len(r2c)))*1e6
r2i = {r: i for i, r in enumerate(r2c)}
for (r1, r2), w in edges.items():
    i, j = r2i[r1], r2i[r2]
    w = 1/w-1
    pdist[i, j] = w
    pdist[j, i] = w

model = UMAP(
    n_components=2, metric='precomputed',
)
rpos = model.fit_transform(pdist)
rpos.shape
```

We can get the positions of compounds in the projection by averaging the positions of associated reactions. A jitter is added to prevent compounds from overlapping reactions.

```
def polar2cartesian(rho, phi):
    x = rho * np.cos(phi)
    y = rho * np.sin(phi)
    return np.array([x, y])

cpos = np.zeros(shape=(len(c2r), 2))
c2i = {c: i for i, c in enumerate(c2r)}
for c in c2r:
    _f = list({r2i[r] for r in c2r[c]}) # mask of associated reactions
    _pos = np.mean(rpos[_f], axis=0)
    if rpos[_f].std(axis=0).max() < 1e-1: # if the reactions are too close, add some jitter
        jitter = polar2cartesian(1/50, 2*np.pi*np.random.rand())
        _pos += jitter
    i = c2i[c]
    cpos[i] = _pos
cpos.shape
```

For network edges, we will use the inverse of the compound promiscuity.

```
eweights = []
compound_has_edge = set()
for r in r2c:
    for c in r2c[r]:
        eweights.append((r, c, 1/compound_promiscuity[c]))
        compound_has_edge.add(c)
e2i = {e: i for i, e in enumerate(eweights)}
```

```
fig = BaseFigure()
fig.add_trace(
    go.Histogram(
        x=[w for _, _, w in eweights],
    )
)
fig = ApplyTemplate(fig, layout=dict(height=200))
fig.show()
```

For each reaction node, we will include additional information, starting with their parent pathway and GSEA results.

```
df_gsea = pd.read_csv("./gsea/results.csv")
print(df_gsea.shape)
df_gsea.head(2)
```

We will create a lookup table for when we are adding this information to the nodes. Since a reaction may belong to multiple pathways, we will also create a ranking and take the most relavent as indicated by the GSEA results.

```
df_gsea = df_gsea.sort_values(["pvalue", "score"], ascending=[True, False])
gsea_lookup = {r["set_name"]:r for _, r in df_gsea.iterrows()}
pathway2rank = {p:i for i, p in enumerate(df_gsea["set_name"])}
df_gsea.head(2)
```

We can also add the differential abundance results from DESeq2.

```
df_ds2 = pd.read_csv("./deseq2/results.csv", index_col=0)
df_ds2.index.name = "gene"
```

```
df_ds2 = df_ds2.reset_index()
print(df_ds2.shape)
df_ds2.head(2)
```

Again, we create a lookup table for when this information is needed. Since each reaction may be the result of multiple genes, we will also create a ranking to determine the most relavent gene as indicated by the DESeq2 results.

```
df_ds2 = df_ds2.sort_values(["pvalue", "log2FoldChange"], ascending=[True, False])
ds2_lookup = {r["gene"]:r for _, r in df_ds2.iterrows()}
gene2rank = {g:i for i, g in enumerate(df_ds2["gene"])}
df_ds2.head(2)
```

We will use the metabolic model to link reactions to pathways. Since the names are not present in the SBML, we will use the local KEGG model.

```
groups = gs_model["sbml"]["model"]["groups:listOfGroups"]["groups:group"]
r2pathways = {}
pathway_names = {}
for g in groups:
    g_id = g["@groups:name"]
    pathway_id = g_id.replace("map", "rn") # convert the index prefix
    meta = kegg_model.Pathways.get(pathway_id)
    name = meta.name if meta else g_id
    pathway_names[pathway_id] = name
    for r in find(g, "@groups:idRef"):
        r2pathways[r] = r2pathways.get(r, [])+[pathway_id]

len(r2pathways)
```

Let's start to construct the cytoscape save file, starting with the nodes.

```
i = 0       # a counter to assign unique ids to nodes. increment for each node
nodes = []  # we will store the nodes here
n2i = {}    # this lookup will convert
for _n, _pos, _type in [    # add reactions and compounds
    (r2c, rpos, "reaction"),
    (c2r, cpos, "compound"),
]:
    for k, (x, y) in zip(_n, _pos):
        n2i[k] = i
```

```
10
11          if _type == "reaction":
12              pathways = r2pathways.get(k, [])
13              pathways = sorted(pathways, key=lambda p: pathway2rank.get(p, 1e6))
14              relavent_pathway = pathways[0] if pathways else None
15              pathway_name = pathway_names.get(relavent_pathway, relavent_pathway)
16
17              gsea_meta = gsea_lookup.get(relavent_pathway, None)
18              gsea_score = gsea_meta["score"] if gsea_meta is not None else None
19              gsea_pvalue = gsea_meta["pvalue"] if gsea_meta is not None else None
20
21              genes = r2genes.get(k, [])
22              genes = [g[2:] for g in genes]
23              genes = sorted(genes, key=lambda g: gene2rank.get(g, 1e6))
24              relavent_gene = genes[0] if genes else None
25
26              ds_meta = ds2_lookup.get(relavent_gene, None)
27              ds_lfc = ds_meta["log2FoldChange"] if ds_meta is not None else None
28              ds_pvalue = ds_meta["pvalue"] if ds_meta is not None else None
29
30          else: # for compounds, we can leave these fields blank
31              relavent_pathway = None
32              pathway_name = None
33              gsea_score = None
34              gsea_pvalue = None
35              relavent_gene = None
36              ds_lfc = None
37              ds_pvalue = None
38
39          nodes.append(dict(
40              id=i, x=float(x)*1000, y=float(y)*1000, # numeric id and posiiton
41              v=dict( # node attributes
42                  name=names.get(k, k),
43                  type=_type,
44                  gene=relavent_gene,
45                  gene_lfc=ds_lfc,
46                  gene_pvalue=ds_pvalue,
47                  pathway=relavent_pathway,
48                  pathway_name=pathway_name,
49                  pathway_score=gsea_score,
50                  pathway_pvalue=gsea_pvalue,
51              )
52          ))
53          i += 1
```

Create a collection of edges, indicating the links for nodes. The only attribute added is the edge weight.

```
edges = []
for j, (r, c, w) in enumerate(eweights):
    edges.append(dict(
        id=j, s=n2i[r], t=n2i[c],
        v=dict(w=float(w)),
    ))
```

Additional parameters are obtained from creating a simple network with Cytoscape and saving it. The only modification is registering the node and edge attributes we created above.

```
META = [
    {
        "CXVersion": "2.0",
        "hasFragments": False
    },
    {
        "networkAttributes": [
            {
                "name": "gen",
                "description": "gen from script"
            }
        ]
    },
    {
        "attributeDeclarations": [
            {
                "networkAttributes": {
                    "name": {
                        "d": "string"
                    },
                    "description": {
                        "d": "string"
                    },
                },
                "nodes": {k: {"d": t} for k, t in [
                    ("name", "string"),
                    ("type", "string"),
                    ("gene", "string"),
                    ("gene_lfc", "double"),
                    ("gene_pvalue", "double"),
```

```
                    ("pathway", "string"),
                    ("pathway_name", "string"),
                    ("pathway_score", "double"),
                    ("pathway_pvalue", "double"),
                ]},
                "edges": {
                    "w": {
                        "d": "double"
                    }
                }
            }
        ]
    },
]

VISUAL = [
    {
        "visualProperties": [
            {
                "default": {
                    "network": {
                        "NETWORK_BACKGROUND_COLOR": "#FFFFFF"
                    },
                    "edge": {
                        "EDGE_LABEL": "",
                        "EDGE_LABEL_COLOR": "#000000",
                        "EDGE_LABEL_FONT_FACE": {
                            "FONT_FAMILY": "serif",
                            "FONT_STYLE": "normal",
                            "FONT_WEIGHT": "normal"
                        },
                        "EDGE_LABEL_FONT_SIZE": 12,
                        "EDGE_LABEL_OPACITY": 1,
                        "EDGE_LABEL_ROTATION": 0,
                        "EDGE_LABEL_MAX_WIDTH": 100,
                        "EDGE_LINE_STYLE": "solid",
                        "EDGE_OPACITY": 1,
                        "EDGE_SELECTED_PAINT": "red",
                        "EDGE_SOURCE_ARROW_COLOR": "#000000",
                        "EDGE_SOURCE_ARROW_SHAPE": "none",
                        "EDGE_LINE_COLOR": "#000000",
                        "EDGE_TARGET_ARROW_COLOR": "#000000",
                        "EDGE_TARGET_ARROW_SHAPE": "none",
                        "EDGE_VISIBILITY": "element",
```

```
                        "EDGE_WIDTH": 1,
                        "EDGE_Z_LOCATION": 0
                    },
                    "node": {
                        "NODE_BORDER_COLOR": "#000000",
                        "NODE_BORDER_STYLE": "solid",
                        "NODE_BORDER_OPACITY": 1,
                        "NODE_BORDER_WIDTH": 1,
                        "NODE_BACKGROUND_COLOR": "#FFFFFF",
                        "NODE_HEIGHT": 40,
                        "NODE_LABEL": "",
                        "NODE_LABEL_COLOR": "#000000",
                        "NODE_LABEL_FONT_FACE": {
                            "FONT_FAMILY": "serif",
                            "FONT_STYLE": "normal",
                            "FONT_WEIGHT": "normal"
                        },
                        "NODE_LABEL_FONT_SIZE": 12,
                        "NODE_LABEL_OPACITY": 1,
                        "NODE_LABEL_POSITION": {
                            "HORIZONTAL_ALIGN": "center",
                            "VERTICAL_ALIGN": "center",
                            "HORIZONTAL_ANCHOR": "center",
                            "VERTICAL_ANCHOR": "center",
                            "JUSTIFICATION": "center",
                            "MARGIN_X": 0,
                            "MARGIN_Y": 0
                        },
                        "NODE_LABEL_ROTATION": 0,
                        "NODE_LABEL_MAX_WIDTH": 100,
                        "NODE_BACKGROUND_OPACITY": 1,
                        "NODE_SELECTED_PAINT": "yellow",
                        "NODE_SHAPE": "ellipse",
                        "NODE_VISIBILITY": "element",
                        "NODE_WIDTH": 40,
                        "NODE_Z_LOCATION": 0
                    }
                },
                "nodeMapping": {},
                "edgeMapping": {}
            }
        ]
    },
    {
```

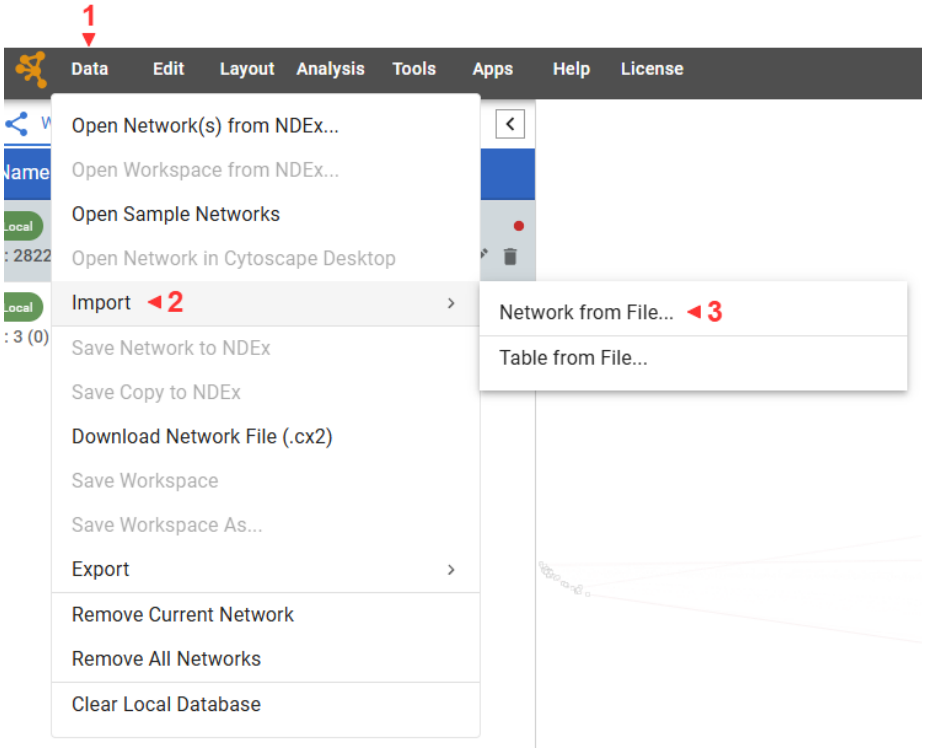
```
        "nodeBypasses": []
    },
    {
        "edgeBypasses": []
    },
    {
        "visualEditorProperties": [
            {
                "properties": {
                    "nodeSizeLocked": False,
                    "arrowColorMatchesEdge": False
                }
            }
        ]
    },
]
```

We can now create the Cytoscape save file and load it into Cytoscape.

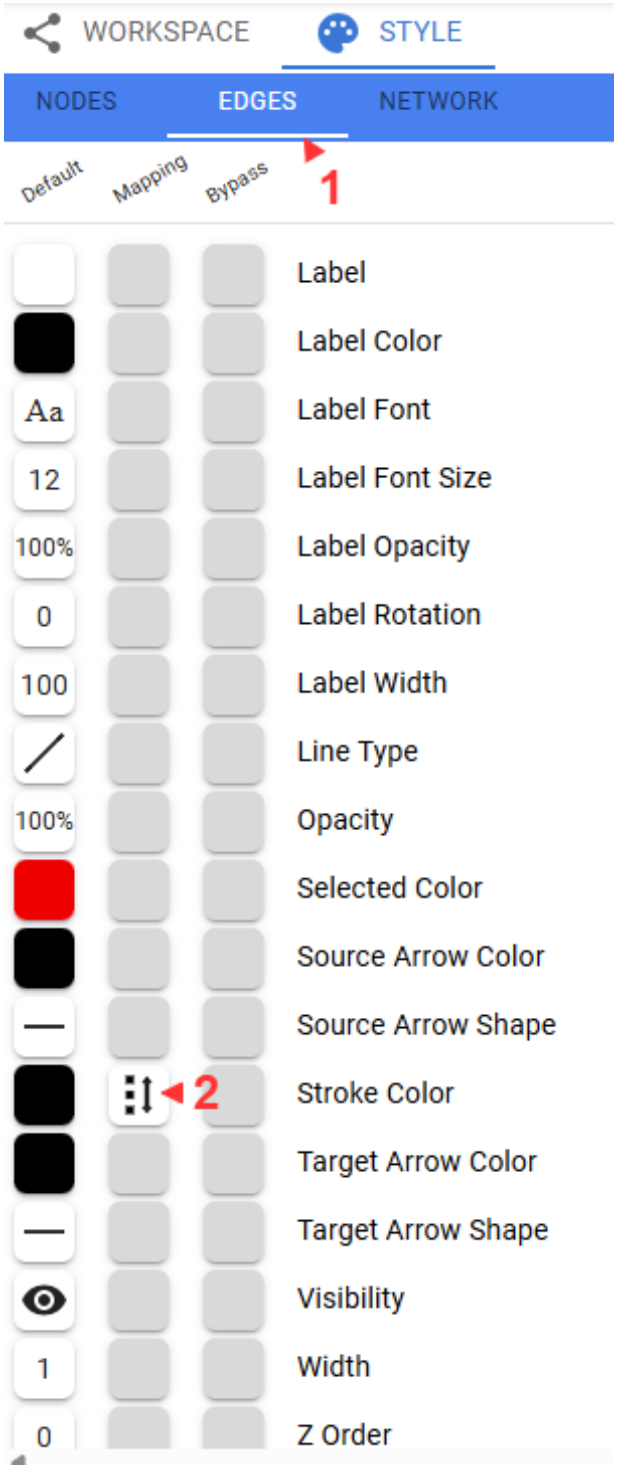
https://web.cytoscape.org

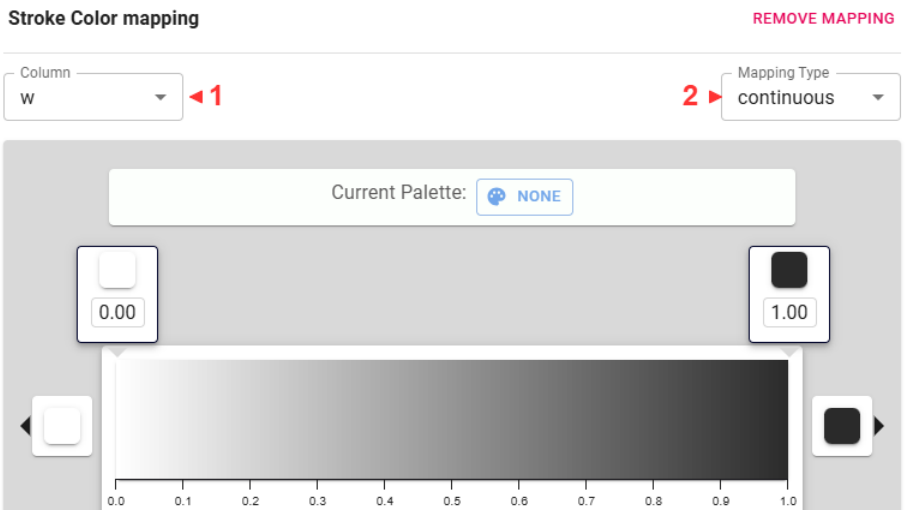
```
import json

Path("./cytoscape").mkdir(exist_ok=True, parents=True)
with open(f"./cytoscape/{MAG}.cx2", "w") as f:
    json.dump(HEADER+[dict(nodes=nodes), dict(edges=edges)]+VISUAL, f, indent=4)
```



Let's use the edge weights to fade the less important edges.


WORKSPACE
STYLE
NODES
EDGES
NETWORK
Default
Mapping
Bypass
1
Label
Label Color
Label Font
Label Font Size
Label Opacity
Label Rotation
Label Width
Line Type
Opacity
Selected Color
Source Arrow Color
Source Arrow Shape
Stroke Color
2
Target Arrow Color
Target Arrow Shape
Visibility
Width
Z Order
Aa
12
100%
0
100
100%
1
0



Some useful navigational tips:

- Scroll to zoom.
- Click and drag to pan.
- Nodes can be dragged around
- Multiple nodes can be selected by pressing shift, clicking, and dragging a box.
- Node properties are shown in a table at the bottom

Play with mapping other values to different visual aspects of the network. Save your network with `Data -> Download Network File (.cx2)`

This concludes the lab!