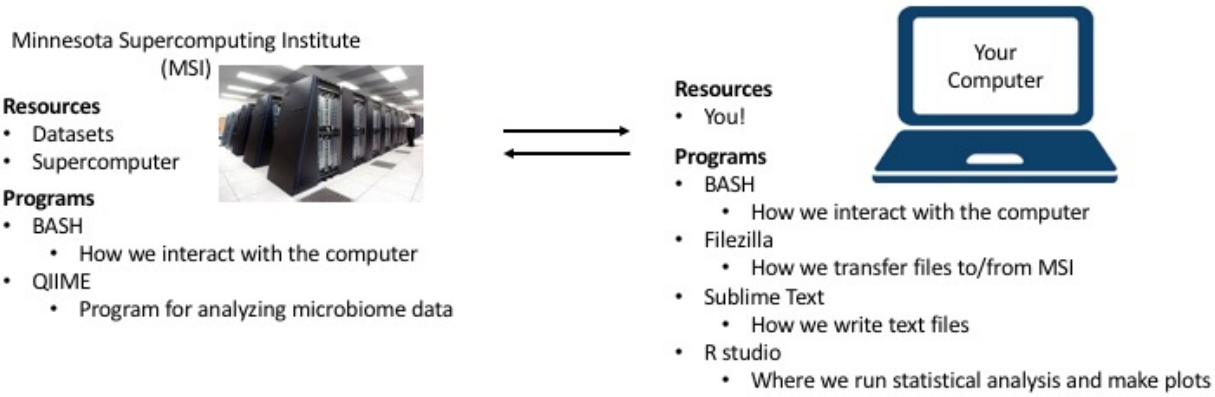
Computational Microbiology Lab Manual (Biology 2003/3004)

Tonya Ward Contributions by: Elise Morton, Gabe Al-Ghalith, Joseph Guhlin, Andrew Honsey, Jen Teshera-Levye, Rachel Soble, & Maxwell Kramer

Getting Started

Overview of Computational Microbiology Workflow

During the computational microbiology section of this course you will use your own computer, and the supercomputers at the Minnesota Supercomputing Institute (MSI) to analyze microbiome data. Below is a diagram explaining which resources and programs will be found on your computer and on MSI.



In order to analyze the microbiome datasets available, we must install some programs on your computer. These programs are listed in the table below. Links for these programs are also available on the course Moodle for your convenience.

```
## Warning: package 'knitr' was built under R version 3.3.2
```

Program	Use	Download	Cost
Putty	Connect to MSI (Windows only)	**Mac**: Not Needed **Windows**: http://www.putty.org	free
FileZilla	Transfer files to/from MSI	**Mac & Windows**:https:FileZilla-project.org	free
Sublime Text	Writing text files	**Mac & Windows**: www.sublimetext.com/3	free
RStudio	Analyze & plot data	**Mac and Windows**: First: [https://cran.r-project.org & www.rstudio.com/products/rstudio/download	free

PuTTY

PuTTY allows people with Windows to access the computers on MSI from the command line. It is a terminal application that we will use specifically for MSI. **Mac and Linux users do not need to this**, and will access MSI using their **terminal** instead.

FileZilla

FileZilla will let you connect your computer to the supercomputers at MSI https://FileZilla-project.org. This is a point and click program with a visual user interface. You can install it and we will set up the MSI-specifics later when we first log in to MSI.

Sublime Text

Sublime Text is a text editor https://www.sublimetext.com/3. Just like we might use Microsoft Word for writing our papers and assignments, we need a text editor to write the type of text files we use in computational microbiology. We need a special text editor because the computer will be reading in and processing our files directly, so they must be formatted properly. We cannot use something like Word, or Notepad because they embed special modifications into the files they create. For the ease of troubleshooting, everyone is required to use **Sublime Text 3**. This is a point and click (and type) program with a visual user interface. You can install and use it right away.

RStudio

R is the programming language we will use to analyze and plot our data. We can think of it like Microsoft Excel. **R** can do everything we would need to do in Excel and a *lot* more. To download R, we will need to download it for your particular operating system. Choose your appropriate link from here: http://cran.us.r-project.org/.

After this downloads, you will probably get two R icons which open a dialogue which looks similar to the terminal. We will never be using these because it is kind of a clunky way to use R. Instead we will be installing an Integrated Development Environment (IDE) for R called **RStudio**. It works as a wrapper for R to create somewhat of a personalized console with different panes containing different information about what you are working on. To download it go to this link and choose the right installer for your operating system: http://www.rstudio.com/products/rstudio/download/.

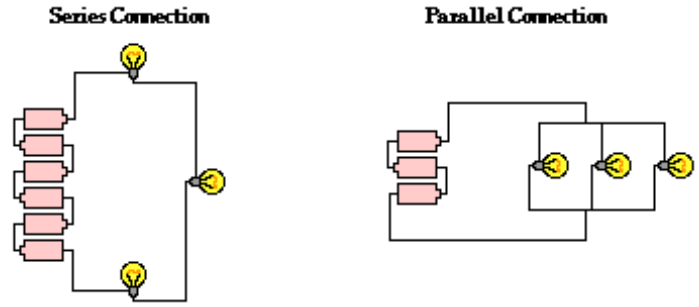
Accessing MSI

What is MSI?

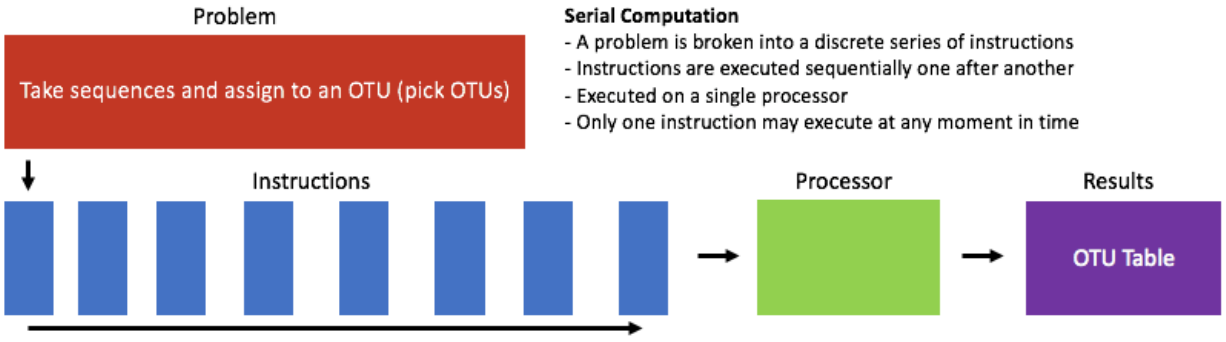
MSI stands for the Minnesota Supercomputing Institute. It is an academic unit of the University of Minnesota. They offer numerous services, such a batch high performance computing (HPC), interactive HPC and data storage. For the context of this course, you can think of MSI as a resource of many large computers that we can use to perform our computational tasks.

Serial versus Parallel Computing

When we use MSI we can complete our tasks faster if we run them in parallel. This is similar to thinking about electricity. Look at the example below with light bulbs. On the left the circuit is set up in series (bulbs one after another), on the right the circuit is set up in parallel (electricity reaching all bulbs at the same time).

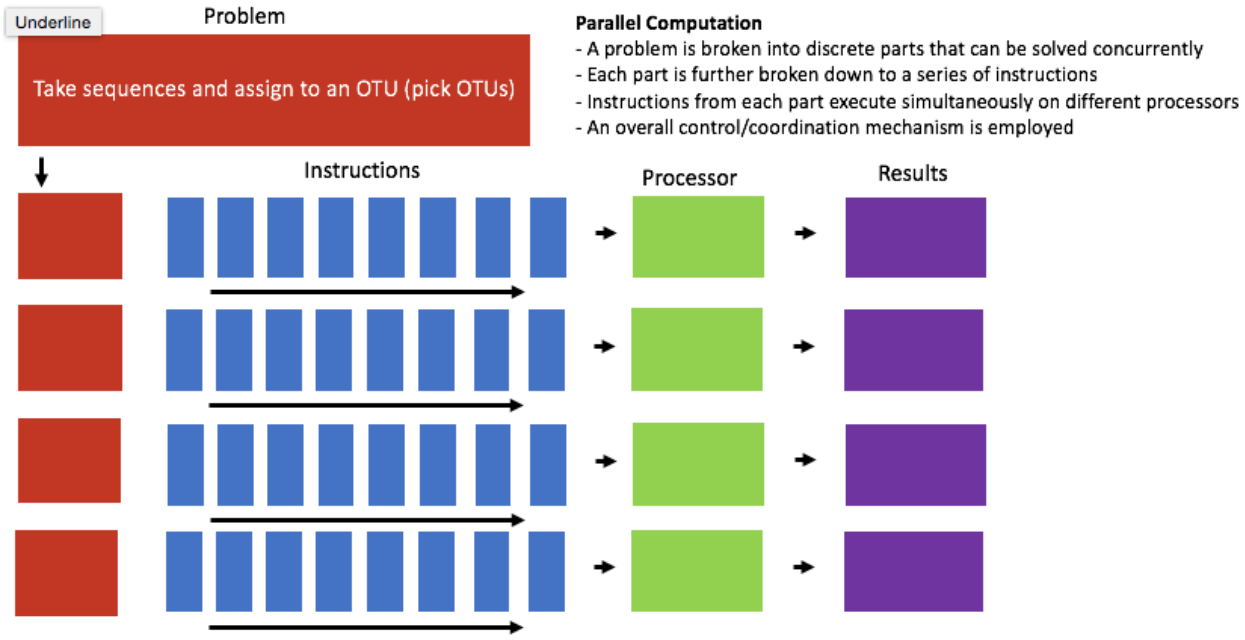


We can do the same type of thing with computers. Below is an example of serial computation. When run a command on MSI we are taking a big dataset and executing a bunch of instructions. The instructions have to be done in a certain order and only one can be done at a time. If we do this in series, it can take a long time. For example, if we want to align millions of DNA sequences to a reference, also known as OTU picking (we will talk about what this is later in the course), it takes a really long time if we want to do it accurately.



Below is an example of parallel computation. We can break up our problem into smaller chunks and run them at the same time on multiple processors. This way we can get to our results

faster. In the example we are using OTU picking, but many of our computational microbiology commands can be run in parallel.



So really, we use MSI so that we can quickly complete tasks that would take our own computers a really long time.

Accessing MSI

If you want to use your personal computer, you would physically interact with the keyboard and mouse attached to it. The computers at MSI are not located where we are, so we must access them remotely. To understand how, we should first learn some terminology:

term	Definition
Client	The computer where the user is sitting and where the connection is initiated from
Server	The remote computer that accepts the connection and provides the service
IP Address	Internet Protocol address. 4 8-bit number that all computers connected to the internet have, telling us where/who the computer is. MSI: @login.msi.umn.edu
SSH	Secure Shell. Enables command-line connection by creating an encrypted connection between your computer and the remote server

To access MSI you must be using the UMN secure wifi, eduroam, or you must be logged into the VPN

If you don't have the VPN, follow instructions here for on Moodle.

Logging In:

Mac/Linux:

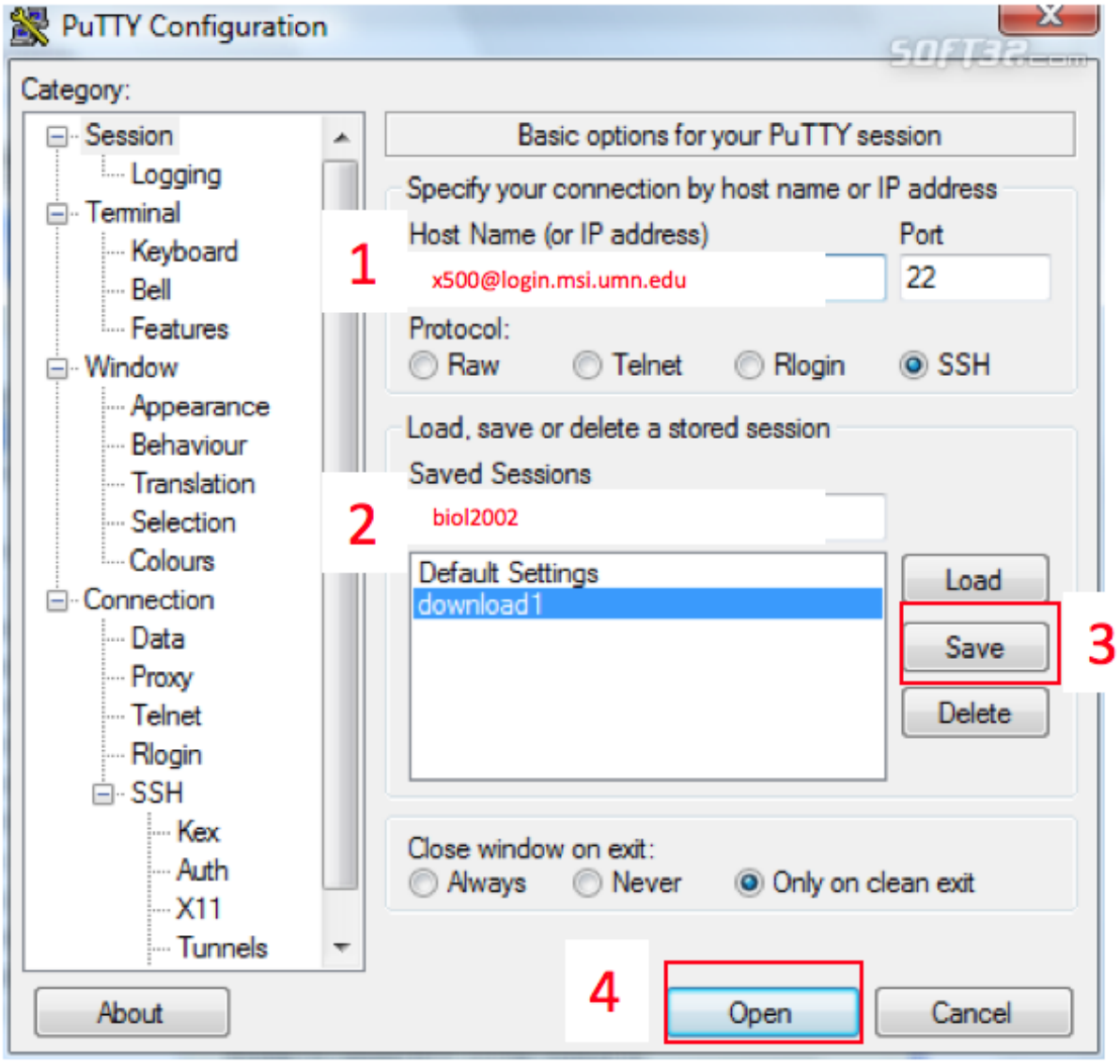
Open your terminal and type (test):

```
ssh x500@login.msi.umn.edu
```

It will then ask for your password, which is your x500 password. **It will not show up as you type it.**

Windows: open PuTTY:

First Time: Open PuTTY and do the following. Once you hit open, it will ask for your x500 password.



Every Other time: click on your biol1961 PuTTY shortcut.

Logout:

```
exit
```

Servers Available at MSI

When you first access MSI you are on the login server. This server can be used to look at which files are there and make simple text files. To perform any other tasks, you must move to a server that is capable of performing larger tasks. The server we will use is **'lab'**.

Move to Lab Server

```
ssh lab
```

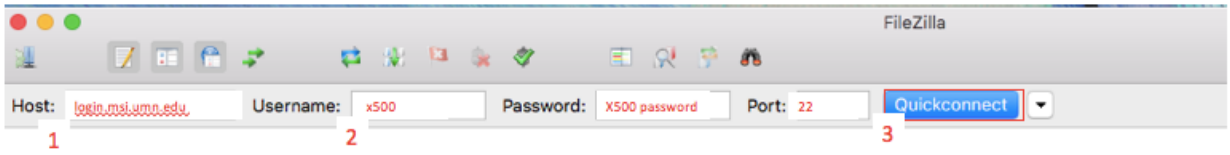
Directory Structure on MSI

We can see the exact directory structure of MSI using FileZilla. Generally it looks like this:

/home/
  biol1961/
    home_directory/
    shared/
    public/

Connecting to MSI with FileZilla

Type in the host IP for MSI: `login.msi.umn.edu`
Type in your username: `x500`
Type in your password
Click Quickconnect



Commenting Code

"Programs must be written for people to read, and only incidentally for machines to execute."
-Structure and Interpretation of Computer Programs, 1985

What is commenting?

When we write and execute code we are telling the computer to do something. The computer reads our code, interprets what to do and executes it. As people, we write comments in the code for other people to read. These comments include explanations as to what the code is doing, instructions, what the inputs and outputs should be and other important pieces of information. Comments are not read by the computer. We ensure they are not read by the computer by placing special symbols before the comment text that stops the computer from reading that text and trying to interpret it. There are different types of comment symbols depending on which language you are writing in. For this course, we will be using the following symbol: #

The pound symbol, or hashtag, tells the computer to not read any text that follows the symbol. The symbol is line specific, so once a new line is started the computer can again read the text.

Example:

```
# this line is a comment, not read by the computer
cd file_path/to/DNA_sequences
# the line above is NOT a comment and is read and executed by the computer
```

Summary: '#' = a comment for us, and not the computer.

Comments on Comments

Commenting is also important for later in the course when we begin to write our own code. You will be expected to comment any code you write so that it can be interpreted by your instructors and peers. For this class, please consider the following points:

1. **The value of a comment is proportional to the distance between the comment and the code.** Good comments stay as close as possible to the code they're referencing. As distance increases, the comment becomes misleading.

2. **Comments should be clear and concise**

3. **Comments should capture intent.** Because we are learning in this course, what we want our code to do and what it actually does might be two different things. Commenting your intent can help your instructors help you!

Adapted from https://blog.codinghorror.com/when-good-comments-go-bad/

The Command Line

Why Command Line?

We need to be able to interact with the Minnesota Supercomputing Institute's (MSI) supercomputers. Some of our computers work with the Windows or Mac iOS systems, while the supercomputers at MSI run UNIX. To work with UNIX, you have to learn how to enter commands from the terminal to tell the computer what to do. This will likely start out as trial and error for you, so don't get discouraged!

You should try these commands and combinations of these commands to get a better understanding. **Repetition** is essential to learning these commands and how they work. Much of working with the command-line is memory. So **copy-and-pasting commands is discouraged**. Typing them on the command-line yourself will accelerate your learning and the command line provides instant-feedback. If you make a mistake, you can use the **up arrow** on the keyboard to correct your command, or you can simply re-type it and try to correct your error.

When working from the command line, the case of the letter is important. Lowercase 'a' does not equal uppercase 'A'. So not only do you have to be careful of typos, you also have to be careful of the case of the letters!

Please note that in this manual command line test will be in a different font. Inputs (commands) that you can actually try out will be **bolded**. With the new font, you should be able to tell the difference between many common characters. Characters such as 0 and a capital O are distinct. As are lower-case l and the number 1. The pipe | and l and 1 are also different. This font is called Monaco.

Additional Help

The following website can help with learning UNIX. You can type in a command, with options, and it will provide a description of that command and any options you supplied. Try it with

```
ls -a
```

Help website: http://www.explainshell.com/

Commands

A command in UNIX/Linux can be as simple as ls. But can also be more advanced and have flags, affecting how the program is run, and arguments, telling the program where to work or which file to work with, depending on the command. Let's consider the following command:

```
ls -a ~/../sample/
```

This command lists the files and directories, including hidden ones, in the sample directory.

Notice the following:

- The command, flags, and argument are all separated by a **space**
- The first set of characters before any space occurs are the command (`ls`). If we forgot a space, and typed ls-a, the computer would try to run a command that is literally named 'ls-a' which does not exist, and you will get `command not found`.

Anatomy of a command (as it looks in the terminal):

```
tward@login02 [~] % ls -a test_directory
```

- `ls` is the command itself
  - `ls` stands for list. It tells the computer to list the contents of something
- `-a` is a flag to the command, affecting how the program functions
  - Flags modify the command. *-a* means all, and is telling the computer to list all files (even hidden ones)
- `test_directory` is the argument to the command
  - It is telling the computer what to list: List is all the files in the directory `~/Desktop/biol1961/file/`
- `tward@login02 [~] %` is the terminal output
  - It tells us where we are (logged in under *tward*, in the login server *login02*). What's inside the [] tells us which directory we are in (~ means "home"), and % tells you when the command starts. We will leave this part out for all future examples.

Commands may have a zero, one, or more flags and zero, one, or more arguments. For example, the `cp` command which copies a file will always have two arguments, but rarely uses flags.

```
cp test_directory/sample.fastq test_directory/my_sample.fastq
```

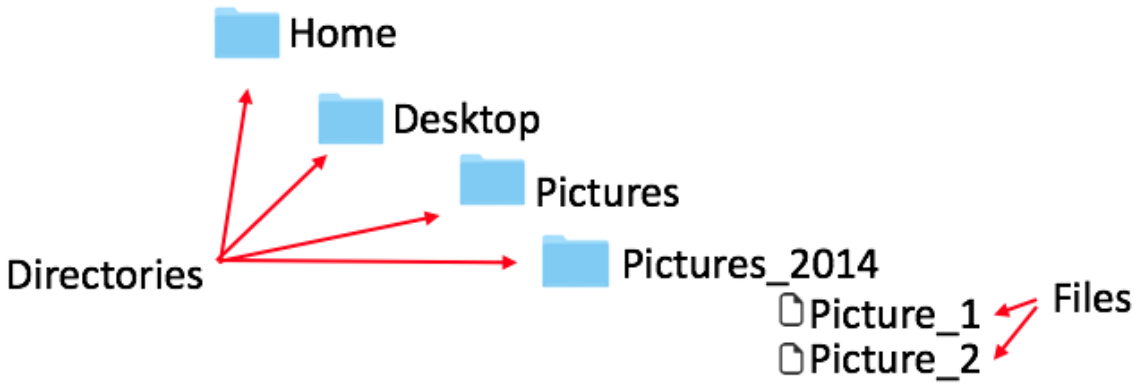
- `cp` is the command itself
- `test_directory/sample.fastq` is the first argument, the name and location of the file to be copied
- `test_directory/my_sample.fastq` is the second argument, the name and location to copy the file to

This will copy the `sample.fastq` file to the same directory, and the name of the new file will be `my_sample.fastq`.

Where Are We Working?

- When working from the command line, we first start in our home directory (a directory is another name for a folder)
- From our home directory we can reference other directories or paths to files
- Commands are performed in the directory you are in
- When you change directories (cd) you move from one location to another
- Commands default to working in your *current* directory, but you can tell them to work somewhere else using a specified path

Think of your own computer. When you open your "Finder" (Mac) or Windows Explorer (Windows), you start in your home directory. From there you have other directories you can enter. Inside those directories you might have even more subdirectories. Let's use an example of a pictures folder on your desktop. If you want to access Picture_1 from 2014, you would go to your Desktop, then into Pictures, then into Pictures_2014 and click picture_1.



Your Home Directory

Your home directory is your default starting place when you log in to MSI. On MSI, my home directory is '/home/biol1961/tward'. I can check this by opening my terminal. logging in to MSI, and typing pwd (print working directory).

```
pwd   /home/biol1961/tward
```

I can also test this by writing using `cd ~`. With this we move to our home directory (cd = change directory, ~ = home). We then check what directory we are in with `pwd` (print working directory).

```
cd ~
pwd
  /home/biol1961/tward
```

We can also check using `echo`, which will print the output of the ~ to the screen.

```
echo  ~
  /Users/tward
```

Changing Directories

- The command `cd` changes directories
- Use `cd ~` to change to your home directory
- Use `cd ..` to move into the parent directory (up one from where you are)
- Use `cd shared` to move into a directory called 'shared'

```
cd ~/../shared/
```

means to change directories to:
1. My home directory using ~ (which is *home/biol1961/tward*)
2. Go up to the parent directory using `../`
3. And the to the shared directory using `/shared`

If I want to move back to my home directory from here, I can type:

```
cd ~
```

which means to move to my home directory. note that `cd ~` will bring you home no mater where you are.

Try moving around on MSI from your home directory to the main biol1961 directory, to the shared directory, to the public directory...

Telling Commands Where Files Are

- Commands & programs work from where you currently are (see: `pwd`)
- When telling commands how to get somewhere, you can give a start position, such as in the following command:

```
cp ~/../shared/File.txt ~
```

This tells the `cp` command to start at your home directory (~), and then move into the parent directory (`..`), and into the shared folder and copy the `File.txt` file to your home directory (~).

Our Biology 1961 directory structure should look like this:

```
home/
    biol1961/
        x500/
        shared/
        public/
```

If we want to work from our home directory we can reference it, regardless of where we currently are, like this `/home/biol1961/x500`. Note: a slash in front means the absolute path from the root of the directory tree (home).

Relative path: `biol1961/x500`
Absolute path: `/home/biol1961/x500` or `~`

**remember ~ is short for your home directory.

Absolute versus Relative File Paths

Notice that in comparison to what we talk about above, here we are talking about referencing the path to a file, not the path itself. But the rules are still the same. An absolute file path is the file's address from the root of the computer's file system. For example:

```
/home/biol1961/x500/my-file.txt
```

is an **absolute** file path, and will work from any directory. A **relative** file path could be

```
my-file.txt
```

The command receiving this file path will only work if that file is in the same directory the command is running from. Another relative file path could be:

```
../my-file.txt
```

This example would only work if the file is backward one directory (parent directory) from your current directory. You'll get a "No such file or directory" error when you have not referenced the location of the file properly. **Relative** file paths are used to reduce typing and make things easier. **Absolute** file paths will almost always work. Remember, the way to tell a **relative** path from an **absolute** path is by a leading **forward slash /**.

Listing Files and Directories

- The command `ls` will allow you to list files and directories
- The default location it looks is your **present working directory**, which you can identify with the command **pwd**
- You can also specify other locations to examine; by using `ls ~/sample_directory/` you are asking to list files and directories found in the sample_directory folder
- You can list hidden files by using the flag `-a` and you can see file size, date, modification, and permissions by using `-l`, you can get file sizes in human readable format with `-h`
- You can combine these, and use `ls -lah ~/Desktop/` to find out this information for files in the shared folder

```
ls -lah /home/biol1961/tward/sample_directory
total 1.0G
```

```
drwxr-s---. 3 tward biol1961 4.0K Oct 3 13:04 .
drwxr-xr-x. 55 root biol1961 12K Sep 6 09:30 ..
-rw-r--r--. 1 tward biol1961 3.1K Sep 23 12:04 alpha_div.txt
-rw-r--r--. 1 tward biol1961 0 Sep 27 13:39 alpha_otu.txt
-rw-r--r--. 1 tward biol1961 0 Sep 27 13:39 alpha_taxa.txt
-rw-------. 1 tward biol1961 5.8K Oct 3 13:02 .bash_history
-rw-r--r--. 1 tward biol1961 403 Oct 3 13:04 .bashrc
-rw-r--r--. 1 tward biol1961 2.4M Sep 28 11:14 Gever_100.biom
```

Try listing all of the files in your home directory

Tab Complete

If you want to write a path or file path, you can use the tab button to help you. Let's say we want to type the following `/home/biol1961/tward/sample_directory/my-file.txt`. If we start typing the first couple of letters `/home/biol1961/twa`, we can then use tab to complete the rest for us. If there is only one option, tab will fill in the word. If there is more than one option, hitting tab once will not fill in the word, but hitting it twice will list all our options.

Try writing the path to your desktop by using the tab button.

Moving/Copying Files

To move files, use `mv`. To copy files, use `cp`. If we are in the `shared/` directory, this command moves a file called seq.txt from the shared directory to our home directory.

```
mv seq.txt ~
```

To copy the same file, we would have used cp instead and we would have a copy in both locations.

Try copying a file from named `copy_me.txt` *from the* `shared/` *directory into your home directory.*

Creating a Directory

To create a new directory, use `mkdir`.

In this example, we use `pwd` to show we are in a new directory on the desktop. We use `ls` to show that there are no files or directories in the current directory. We then create a new directory with `mkdir`, and can now see it with `ls`, and then we change into that directory with `cd`.

```
pwd
  Users/tward/Desktop/new_directory

ls
mkdir new_sub_directory
```

```
ls
  new_sub_directory

cd new_sub_directory/
```

Try making a new directory in your home directory. Trying copying a file from the `shared/` *folder (*`copy_me.txt`*) into the new directory you made.*

Counting Lines, words and characters

In computational microbiology we will be using multiple types of text files. Some of these files will include sequences of DNA (all the A,T,G,C's), tables of how many bacteria are in each sample, and other files with information about diversity. Sometimes it is handy to be able to pull snippets of information from these files. For example:

- we can count the number of lines, words, and characters in a file with the command `wc`
- "Words" are defined here by groups of characters (strings) separated by a space.

In this example, we get only the number of lines in the my_file.txt file using the `-l` flag with `wc`. Next, we get the number of lines, words, and number of characters in the file when we don't use the `-l` flag.

```
wc -l ~/my_file.txt
  552544 /home/biol1961/tward/my_file.txt

wc ~/Desktop/biol1961/my_file.fastq
  552544   552544 90114322 /home/biol1961/tward/my_file.txt
```

Try counting the number of lines in the `copy_me.txt` *file*

Getting lines from a file (beginning or end)

- To get lines from the beginning of a file or end of a file, we can use the commands `head` or `tail`
- By default, these commands will give us 10 lines, but we can change this with the flag `-n K` where K is the number of lines you would like to receive

Here is an example, searching a file called `sample.fastq`. This file contains all the sequences of DNA for a sample in a dataset. So, the top 2 lines from a file would be `head -n 2 sample.fastq`

```
head -n 2 ~/Desktop/biol1961/sample.fastq
@M00784_000000000-A8BPP:1:1101:14310:1364#0/2
CGACAACCATGCATCACCTGTCACTTCTGTCCCCGAAGGGAAAAATGCGATTAGGCATCGGTCAAAAGGATCTCACC
CTTCGCTCATCTTCTTCGCGTTGCTTCTAATTCCACCACATGCTCCCCTACTTCTCCGCCTCCCCCTCACTTCCTTT
```

```
GAGTTTCACTCTTGCGAGCGTACTTCCCAGGCGTAGTACTTAATGCTTTCGCTGCGCCACCGTCGCGCTTCCCCCCC
CACCCCTCCTTCCCATCTTTTCCTCCCTCCCCCTCCCGCGTCTCCCATCCCCCTCCCCTTCTCCCCCACC
```

Searching a File

- We can quickly search files with the `grep` command.
- To use `grep` you supply a string to search for, followed by a file to search in
- `grep` will send the result to standard output (typically the screen), which is the entire line that matches the search string you've provided.

Here are some examples, searching `sample.fastq`. This file contains all the sequences of DNA for a sample in a dataset.

```
grep GGGGGGATGAT ~/Desktop/biol1961/sample.fastq
CGACGGCCATGCAACACCTCCACAGGCGCCCCGAAGGGCCTCATCATCTCTGAAACATTCGCCTACAGTTCAAGCTC
CGGTAAGGTTCCTCGCGTATCATCCAATTAAACCCCCAGTTCCTCCGCTTTTGCCGGCCCCCGTCAATTCCTTTGAG
GTTCTACCCTGCCGGCGTACTCCCCCGGGGGGATGATTCATGCCTTCGCTTGGCCGCTTACGACAGACGCAACCAAC
GATCAACCATCATTTACGGCGTGCACTACACGGCTCACGATTCTCACTCCTCTCATCTATCACCACTCCC
CGACAACCATGCAGCACCTGTATCAGTATCCCCGAAGGGACTATGTAACTTTACAGGAATTACTGGAAGGCAAGACC
TGGGAAGGGTCCTCGCGTTGCTACGAAATAAAACAAAAGCTCCGCAGCCTGTGCGGGCCCCCGTCAATTACATTGAG
GTTCAAACTTGCGGCCGTACTCACCAGGGGGGATGATTAATGTGTTTACTTCGGAAAAGAAGGGGTCGATACCCAAT
ACACCTAGCAGCAATCGTTTACAGTGTGGACTACAAGGGTATCTAGTCACCTGTATCTTATACAAATCTG
CGACAACCATGCAGCACCTGCAAAGAGAGTACGAAGGAAGAGATAGTATTCAAAAGGGGCCACTGCAATTCAAGCAC
GGGGAAGGGTCCTCGGCGATCATTGAATTAAAACACATGGTCCTACGGTTGTGACGGGCCCCGTCAATTTCTTTGAG
GTTCACTGTTGCCGGAGTTATCCCCAGGGGGGATGATTAATGATTTTGCTGGGCCGCTCGAATGGTCTGGACAACAC
AGGGACTCGACATTATACGTTGAGGCGTGCCAGGGACACGAACACACGGTCATTTGTCATCAACACACCC
```

The lines returned are very long and are printed on multiple lines. However, they are the same length and you can see we find three lines that contain `GGGGGGATGAT`

Redirecting command output (to another command)

- We can use the output of one command as the input of another
- The pipe, `|` , is a key typically above the enter key. It is a vertical line
- The pipe redirects output from one command to input of another command
  For example, we can see how many lines in the `sample.fastq` file contains the sequence `'GATTACA'`. `grep` counts the number of lines 'GATTACA' appears in the file, and feeds it to `wc` to count the number of lines.

```
grep GATTACA ~/Desktop/biol1961/sample.fastq | wc -l 151
```

We can also see the last 2 lines that match `GATTACA`:

```
grep GATTACA ~/Desktop/biol1961/sample.fastq | tail -2
CGACGGCCATGCACCACCTCGGCCTCCGTCCGAAGAGCCACCCATCTCTGGGTGTTTCAGGCGCCGTTCGAGCCCGT
GTAAGGTTTCTTGCGTTTCATTGAATTTAACCACCTGTTTCTACGCCTGTTCGGGCCCCCCTCCAATTCCTTGAGGT
TTCACGCTTCCGATGTTCCTCCCAGGTGGATGTACTATTGCTGTCGCCTGGGCACCGACAGGGTTCCGCCGGCGGAC
```

```
ACCCATTATTCCTTGTTGAGTGGATTACATGGCAAGCTAATCACCCGTCTGTGTCTCTTCACACTCGCTC
CGACGGCCATGCAACACATGTTTTCATGTCCCCGAAGGGAAAGCTCCATCTCTGGAGCGGTCAATCAATGTCAAGCC
TTGGTAAGGTTCTTCGCGTTGCGTCGAATTAAACCACATACTCCACCGCTTGTGCGGGCCCCCGTAAATTCCTTTGA
GGTTCATCCTTGCGGACGTACTCCCCAGGCGGGGTACTTATTGCGTTAACTCCGGCACAGAAGGGGTCGATACCTCC
TACACCGAGTACCCATCGTTTACGGCAAGGACTACCGGGGATTACAACTCCCTGTCGCCTCTACCAATCT
```

We can also string multiple `grep` commands together, for example we could search for lines containing 'ATG' that also contain 'TAG' that also contain 'GATTACA', and count the number of matches.

```
grep ATG ~/Desktop/biol1961/sample.fastq | grep TAG | grep GATTACA | wc -l
138
```

Redirecting command output (to a file)

- We can also redirect the output of a command to a file
- Output can be directed to a new file with > (this will replace existing content if something is already there!)
- You can add to the end of a file with >> (this will NOT replace existing content, but adds to it instead)

We can use the same example as above, where we used `grep` to search for lines containing 'ATG' that also contain 'TAG' that also contain 'GATTACA', and direct the output to a file called `'many_grep.txt'`.

```
grep ATG ~/Desktop/biol1961/sample.fastq | grep TAG | grep GATTACA | wc -l >
many_grep.txt
```

We can write direct text to a file using `echo`. For example, we can write "the number of lines that contain ATG, TAG and GATTACA:" to the end of the `many_grep.txt` file.

```
echo "This is the number of lines that contain ATG, TAG, GATTACA" >>
many_grep.txt
```

Try writing "I will master computational microbiology" to a new file called 'mantra.txt' on your desktop.

Exploring .txt files from terminal

- We can open and look at text files from the command line with the command `nano`
- `nano file.txt` opens the file
- Arrow keys move up and down
- Directions for quitting the file are located at the bottom of the screen

We can explore what the `sample.fastq` file looks like by typing the following the command, and then quitting by typing `x`.

```
nano sample.fastq
```

Try looking into your mantra.txt file on your Desktop.

For more commands you can use, please see the *Bash_commands* file on Moodle. Please use this file and update it with commands you find useful as you complete the computational microbiology section.

Modifying '.bashrc' on MSI

The following instructions are for modifying your unmask setting on MSI. This setting is listed within your .bashrc file. It controls who has access to the files you create on MSI. The default setting is 077, which means all files and directories are private. We want to set it to 027, so that people in our group (e.g., TAs and instructors) can have access to the files you create.

1. Log onto MSI

```
ssh x500@login.msi.umn.edu
```

PuTTY users: Just click on your MSI PuTTY shortcut

2. Copy the `.bashrc` file from the shared directory to your home directory

```
cp /home/biol1961/shared/.bashrc ~
```

3. Source your file to make it active

```
cd ~
source .bashrc
```

Job Submission on MSI

Why submit jobs

MSI uses job queues to efficiently and fairly manage when computations are executed. The queuing system that MSI uses is called **PBS**, which stands for **Portable Batch System**. We submit a **script** to the supercomputer to be run at a later time (when the resources are available). This is the alternative to waiting around for hours, potentially longer, for the resources you need to become available.

What is a script?

A script is a file containing commands to be run in order. The script itself can then be run like a command and will execute all the tasks outlined in the file. There are many types of scripts. For example, all the commands we will run in QIIME are scripts that execute many tasks for us.
A PBS job script is a type of script we use with the MSI supercomputers. It is a small plain text file containing information about what resources a job requires - including time, number of nodes and memory. The PBS script also contains the commands needed to begin the desired computation.
Below is an example of a PBS script that we will use repeatedly. For our purposes, the text in black will remain the same for all jobs submitted to MSI. The text in red will vary, depending on the user and the job to be submitted.

```
#! /bin/bash -l
#PBS -l nodes=1:ppn=16,mem=2Gb,walltime=3:00:00
#PBS -m abe
#PBS -M your_email
#PBS -o job_name_stdout
#PBS -e job_name_stderr
cd /home/biol1961/x500
module load name_of_software
command_X_Y_Z
```

`#!/bin/bash -l`
The first line in the PBS script defines which type of shell the script will be read with. We need the BASH shell and it will be read line by line (-l).

`#PBS -l nodes=1:ppn=16,mem=2Gb,walltime=3:00:00`
The second line contains the PBS resource request. The sample job will require 3 hours, 1 node, each with 1 processor core per node (ppn), and 2 gigabytes of RAM (mem). Note that commands for the PBS queuing system begin with #PBS. Other commands to be run do not have the '#PBS'.

```
#PBS -m abe
#PBS -M your_email
```

The third and fourth lines are both commands having to do with sending message emails to the user. The first of these lines instructs the PBS system to send a message email when the job aborts, begins, or ends. The second command specifies the email address to be used.

```
#PBS -o job_name_stdout
#PBS -e job_name_stderr
```

The fifth and sixth lines specify the names of the files to which the job's output and errors should be written, respectively. You can change the name (job_name) to specify the job that is run (e.g. OTU_picking_stdout).

```
cd home/biol1961/x500
```

A PBS script should also contain the appropriate change directory commands to get to the job execution location (in this case the computer will move to the user's home directory).

```
module load name_of_software
```

The script also needs to contain module load commands for any software modules that the calculation might need. 'Module load' is effectively the same as opening an application on your computer. This could be something like: `module load qiime/1.8.0`

```
command_X_Y_Z
```

The last lines of a PBS script contain commands used to execute the job. This could be something like: pick_otus.py -i sequences.fasta -o otus.

You must write your PBS scripts in your plain text editor (Sublime Text 3) and save them with the extension `.pbs`

Check a Job Status

there are two commands used to to check the progress of our job: **qstat** and **showq**. For our own jobs, we need to use the **-u username** flag to specify. If there is no flag for **qstat**, it will show all of the jobs currently running or waiting on the specified machine. **showq** will show more information about the jobs, including the starting time, expected finishing time, and usage of computational resources (processors, nodes).

```
qstat -u x500
showq -u x500
```

Kill a Job

When we check the job status, there is a Job ID starting with a number. This number can be used to kill the job. For example, if the Job ID is 4327.node1081.locald, type

```
qdel 4327
```

or

```
qdel 4327.node1081.locald
```

We can kill multiple jobs in a row by specifying the Job IDs, or use `all`, if all the jobs need to be killed.

QIIME

What is QIIME?

QIIME stands for Quantitative Insights Into Microbial Ecology. It is pronounced 'chime'. It is pipeline for preforming microbiome analysis from raw DNA sequences. Some of the things QIIME can do for us includes:

- Quality filtering
- OTU picking
- Assigning taxonomy
- Diversity analysis
- Visualizations
- Statistics

QIIME uses a mix of other existing softwares and algorithms to perform its tasks. Because of this we call it a 'wrapper'. That means it wraps up many other existing tools and algorithms in a package that works as one cohesive unit.

How Do We Use QIIME?

As mentioned above, QIIME is a wrapper for many different components. This means installing QIIME can be extremely challenging because it requires MANY dependencies (other programs and algorithms). For this reason (and the computer power of MSI), we use the 1.8.0 version of QIIME installed on MSI.
When we are on MSI and logged into the lab server, we can turn QIIME 'on' by typing:

```
module load qiime/1.8.0
```

`module load` means out of all the modules (programs installed on MSI), load the one we are going to specify. There are different versions of QIIME on MSI, so we must specify that we want version 1.8.0 with **`qiime/1.8.0`**. If you wanted to see all the different modules available on MSI you could do so with **`module avail`**. If you wanted to turn off a module you had loaded, you can type **`module unload`** followed by the name of the module.

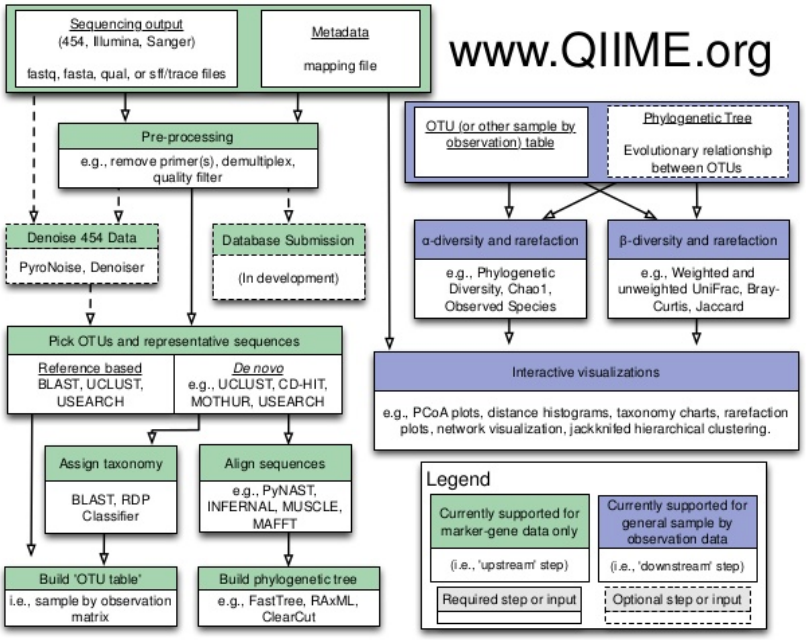
QIIME has many commands (files that contain ordered lists of commands to be run), and these commands can be found at the link below. This is how we complete different steps in our analysis.

QIIME Commands: http://qiime.org/scripts/

Try going to this page and click some of the commands links, for example click **`summarize_taxa.py`** and read what this command does, what the inputs are and the examples. Notice that all commands end in '**.py**'. This is because all the commands are written in a language called **Python**, and commands in Python end in '**.py**'.

QIIME Workflow

Here is a flowchart of the how QIIME works:



Notice the initial inputs in green:

Input	Definition
Sequencing Output	This is all of the raw DNA sequences from the sequencer (.fastq files)
Metadata	This is another word for **mapping file**. It's a tab-delimited text file, where the sample IDs are rows and the columns are different categories of data. For example, which primers were used for sequencing, which body site the sample is from, clinical data (like if the sample came from a person with the disease or a control), etc. Many times we actually make these files in Excel, and export them as .txt files.

The rest of the steps in this pipeline will be covered in detail throughout the rest of the course.

Running Commands

To determine how to run a command, we have to look up the documentation. We can either go to the command page (for QIIME) mentioned above and click on the command we want, or we can type the following:

```
qiime_command.py -h
```

By specifying `-h` we are saying 'HELP!'. The command will list the documentation associated with it. This output will not be as comprehensive as what is available online, but will at least tell us all the possible inputs and outputs.

Example

```
collapse_samples.py
```

Online:

Collapse samples in a BIOM table and mapping file. Values in the BIOM table are collapsed in one of several different ways; see the available options for –collapse_mode. Values in the mapping file are collapsed by grouping the values if they differ for the grouped samples, and by providing the single value if they don't differ for the grouped samples.

Usage: `collapse_samples.py [options]`

Input Arguments:

```
[REQUIRED]

-b, --input_biom_fp
    The biom table containing the samples to be collapsed

-m, --mapping_fp
    The sample metdata mapping file

--output_biom_fp
    Path where collapsed biom table should be written

--output_mapping_fp
    Path where collapsed mapping file should be written

--collapse_fields
    Comma-separated list of fields to collapse on

[OPTIONAL]

--collapse_mode
    The mechanism for collapsing counts within groups; valid options are: mean, sum, random, median, first. [default: sum]

--normalize
    Normalize observation counts to relative abundances, so the counts within each sample sum to 1.0. [default: False]
```

This tells us what the command does:

Uses the mapping file to collapse the OTU table.
The minimum we need to specify (required) is:

- The OTU table we care about in .biom format
- The mapping file we care about
- The output file path for our collapsed OTU table
- The output file path for our collapsed OTU table
- The field we would like to collapse by

Optionally, we can also determine: + The collapse mode + To normalize or not

Terminal:

```
collapse_samples.py -h
```

This will tell us (more or less) the same information, but prints it to the terminal screen.

Quality Control of Sequence Data

When we get our DNA sequences from the sequencer, there is some quality control that must be done. For example, the barcodes and primers used in the sequencing reaction should be removed. Also any sequences that are too low in quality should be discarded.

If we are analyzing 16S amplicon data, then we should also trim our sequences to the expected amplicon size. If we used paire-end sequencing we can also stitch our pairs together to make our sequences longer and of higher quality for alignment.

We can trim and filter our raw sequences using many different tools, but in this course we will be using **SHI7 (pronouned shizen)**. This program will do all the quality control we need and produce a final combined file sequence file that we will use as the input for OTU picking (to determine which bacteria are in our samples). The input that SHI7 requires is a directory of .fastq files, where each sample as its own fastq file.

What's a fastq file?

A `.fastq` file contains our DNA sequences as well as other information regarding the quality of the sequencing reaction. Each sequence within a `.fastq` has four lines of information:

- Line 1 begins with a '@' character and is followed by a sequence identifier and an optional description
- Line 2 is the raw sequence letters (A,T,G,C...)
- Line 3 begins with a '+' character and is optionally followed by the same sequence identifier
- Line 4 encodes the quality values for the sequence in Line 2, and contains the same number of symbols as letters in the sequence

What's a fasta file?

A `.fasta` or `.fna` file contains our DNA sequences only. Each sequence within a `.fasta` has two lines of information:

- Line 1 begins with a '>' character and is followed by a sequence identifier and an optional description
- Line 2 is the raw sequence letters (A,T,G,C...)

The `.fasta` or `.fna` format is the input format required for OTU picking. When we quality control our sequences we also convert them from `.fastq` to `.fna`.

How Do We Quality Control Sequences?

SHI7 is currently installed in the `shared` directory, and our `.bashrc` file contains information to have MSI use this program as if it were installed as a module. The full documentation for SHI7 is located here: https://github.com/knights-lab/shi7.

To run SHI7 picking, we will use the main SHI7 command with the following parameters:

```
shi7.py
-i directory_with_fastqs/
-o qc_reads_output
```

The input file path is to the sequences you want to process (-i). This should be a directory with one fastq per sample. The name of each fastq should be the sample ID followed by the `.fastq` file extension. The output directory is where you want your final clean sequence file to be (-o). There are some optional parameters to use depending on the dataset. The include:

```
-SE                 # This will use single-end mode.
                    # If you don't have paired reads, use -SE
-trim_q 32          # Trim sequences based on quality, default is 20,
                    # increase to 32 if sequencing run is old (before 2015)
--adaptor Nextera   # You can specifically take out the adapter that was used
                    # In most recent sequencing it's Nextera adapters
--strip_underscore T # You can process the file names to keep the first part
```

For the full list of options, you can type:

```
shi7.py -h
```

This is telling SHI7 to print the help page.

To run this command, we must submit a job file. Your job file should look like this:

```
#!/bin/bash -l
#PBS -l nodes=1:ppn=16,mem=2Gb,walltime=6:00:00
#PBS -m abe
#PBS -M x500@umn.edu
#PBS -o job_name_stout
#PBS -e job_name_stderr

cd /home/biol1961/x500

module load python

shi7.py
-i directory_with_fastqs/
-o qc_reads_output
```

Of course, you have to modify the file to specify YOUR file paths and the outputs to what you want. Things that **cannot** change include:

- The first 3 lines. The `nodes` and `ppn` must be 1 and 16 for the lab queue

- The `__module load python` (SHI7 requires python to run)

All the parameters for the actual command **must be all on one line in the job file**, with a space between the parameter letter and value. In the above example they are on separate lines so that you can read them easily. The files and paths must be specific to you.

As mentioned above, the script will run its own jobs after it has started. You will know quallity control is done when you have the following files in your output directory (-o):

shi7.log (all the information about the quality control)
combined_seqs.fna (your combined and cleaned sequences in .fna format)

You will also have some files that were generated by the jobs submitted by this script. They include:
job_name_stout (the standard out captured by the job submission)
job_name_stderr (the errors captured by the job submission)

Picking OTUs

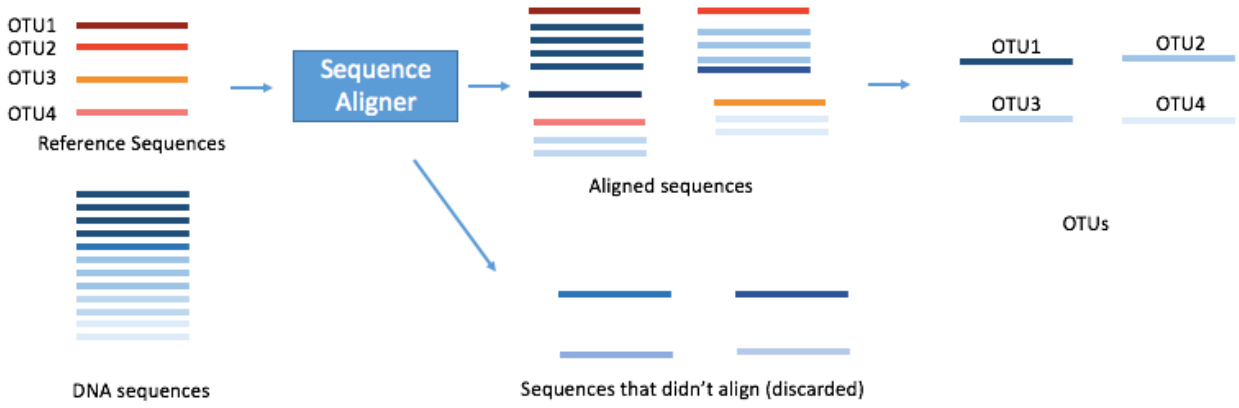
What is OTU picking?

OTU picking is how we take our 16S DNA sequences and assign them to an OTU identifier. An operational taxonomic unit (OTU) is a cluster of similar 16S sequence variants. Each cluster is meant to represent a taxonomic unit of bacteria (species, genus, phylum..) depending on the sequence similarity threshold. OTU clusters are usually defined by a 97% identity threshold of the 16S gene sequence variants. There are three main types of OTU picking that we can do.

De-novo

- Doesn't use a reference database
- Majority of the reads are clustered
- Very slow
- Erroneous reads get clustered
- Cannot assign taxonomy

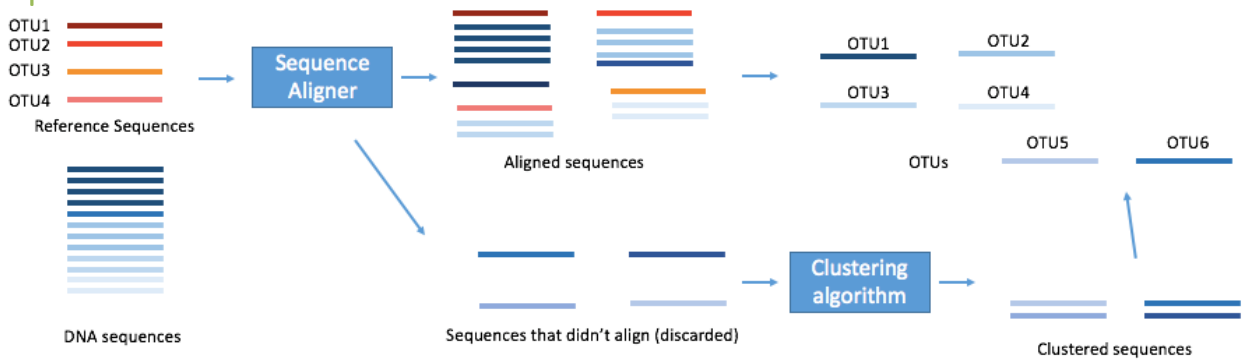
Closed Reference



- Reference database is quality filtered
- Faster because you can use parallel computation
- No new OTUs can be observed
- Reference database bias

- Uses the GreenGenes database of all known 16S
- Can assign taxonomy

Open Reference



- Combines the two approaches
- No data is thrown out
- De-novo clustered OTUs cannot be assigned taxonomy

How do we actually pick OTUs?

Depending on the OTU picker you choose to use, OTU picking can be *very* computionally heavy. This means it can require a lot of time and resources. Thanks to the development efforts of microbiome researchers, we have been able to speed up this process immensely. The OTU pickers within QIIME are currently not the gold standard, so we will use a different OTU picker that is installed seperately.

For this course we will used a closed reference OTU picker called **NINJA**, which stands for **NINJA Is Not Just Another aligner**. NINJA is currently installed in the `shared` directory, and our `.bashrc` file contains information to have MSI use this program as if it were installed as a module. The full documentation for NINJA is located here: https://github.com/GabeAl/NINJA-OPS.

To run OTU picking, we will use the main NINJA command with the following parameters:

```
ninja.py
-i combined_seqs.fna
-o ninja_otus
-m normal
-p 4
-z
-d 2
```

The input file path is to the sequences you want to align (-i). These should be the output of the quality control we did earlier. The output directory is where you want your final OTU table to be (-o). The -m parameter set to normal tells NINJA to run at medium sensitivity (to maximize the speed to accuracy ratio). We will use 4 threads (-p), and we will search both DNA strands (-z). We will also set denoising to 2 (-d), which means we will discard any sequences that appear less than 2 times.

To run this command, we must submit a job file. Your job file should look like this:

```
#!/bin/bash -l
#PBS -l nodes=1:ppn=16,mem=2Gb,walltime=6:00:00
#PBS -m abe
#PBS -M x500@umn.edu
#PBS -o job_name_stout
#PBS -e job_name_stderr

cd /home/biol1961/x500

module load python bowtie2

ninja.py
-i combined_seqs.fna
-o ninja_otus
-m normal
-p 4
-z
-d 2
```

Of course, you have to modify the file to specify YOUR file paths and the outputs to what you want. Things that **cannot** change include:

- The first 3 lines. The `nodes` and `ppn` must be 1 and 16 for the lab queue

- The `__module load python bowtie2` (NINJA requires python and bowtie2 to run)

All the parameters for the actual command **must be all on one line in the job file**, with a space between the parameter letter and value. In the above example they are on separate lines so that you can read them easily. The files and paths must be specific to you.

As mentioned above, the script will run its own jobs after it has started. You will know OTU picking is done when you have the following files in your output directory (-o):

ninja_log.txt (all the information about the alignment)
ninja_otutable.biom (your otu table)

You will also have some files that were generated by the jobs submitted by this script. They include:
job_name_stout (the standard out captured by the job submission)
job_name_stderr (the errors captured by the job submission)

If you want to pick otus again, you should delete these files prior to submitting another OTU picking job.

What's an OTU table?

The output of **ninja.py** is an OTU table in **.biom** format. When in .biom format the file is not in a form we can read easily. We must first convert the table to .txt file to view it.

Converting .biom files

A **.biom** file is a way to package a lot of information in a way that doesn't take up too much space. Because all the information is compact, it makes the file not human readable. If you call **head** on a **.biom** file, the output will look mostly like gibberish. What is important is that QIIME and other microbiome softwares use **.biom** files because they are smaller and fast to work with. If you want to put your OTU table in a human-readable format you have to convert it to a tab-delimited file. We will cover this later.

Biom Summaries

OTU Tables in .biom format

We store out OTU tables in two different formats, either as a tab-delimited text file (.txt) or as a compact, human non-readable biom format (.biom). When we store the table as a biom file, we cannot easily look in the file to see how many OTUs or samples there are, but we can access a summary of the file using some biom commands through QIIME.

Biom Summary

We can summarize our OTU table with the **biom summarize-table** command while using QIIME interactively:

```
ssh lab

cd /home/biol1961/x500

module load qiime/1.8.0

biom summarize-table -i file/path/to/otu_table.biom -o OTU_summary.txt
```

The input file would be the file path the YOUR OTU table and the output can be whatever you would like to name the summary file. This command will make a text file that contains a summary of your OTU table. We can look at the nonctents of the .txt file by using nano (the text editor on MSI).

```
nano OTU_summary.txt
```

```
Num samples: 104
Num observations: 9957
Total count: 1443869
Table density (fraction of non-zero values): 0.053
Table md5 (unzipped): c3ea77ea5cb942958ee7258abf7309ba

Counts/sample summary:
 Min: 4.0
 Max: 53844.0
 Median: 12121.000
 Mean: 13883.356
 Std. dev.: 8912.638
 Sample Metadata Categories: None provided
 Observation Metadata Categories: taxonomy

Counts/sample detail:
 700102569: 4.0
 700107977: 1933.0
 700105480: 2549.0
 700032671: 3716.0
 700097437: 4535.0
 700034196: 4645.0
 700097859: 4692.0
 700035264: 4802.0
 700107920: 4943.0
 700035697: 4973.0
 700097781: 5329.0
 700035184: 5464.0
 700102995: 5663.0
 700034228: 5968.0
 700097477: 6132.0
 700108145: 6264.0
 700114637: 6618.0
 700108866: 6781.0
 700097163: 6967.0
 700099539: 7051.0
```

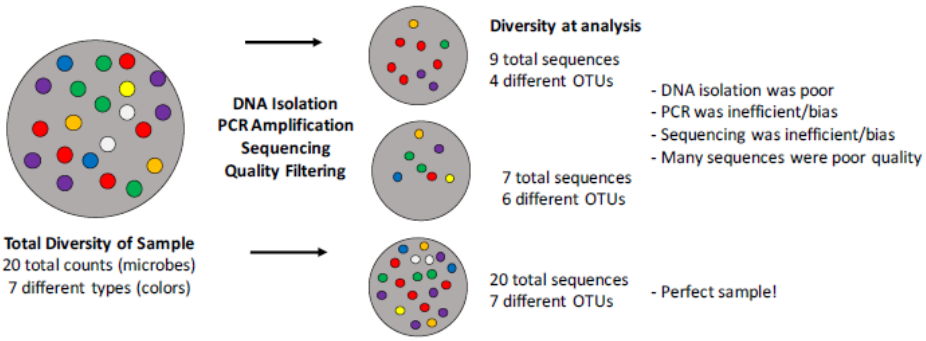
You can see an example of the summary file above. It tells us: * The number of samples * THe number of observations (OTUs) * The minimum, maximum, median, mode and standard deviations of of the number of counts per sample * The taxonomy is stored as the observation metadata * A list of how many counts are in each sample

Rarefaction

What is Rarefaction?

In microbiome research, diversity represents the number OTUs within in a data set. This number can be greatly impacted with different sequencing depths. For example, the deeper you sequence the more species you will find. This is a problem, especially if you sequence 50,000 reads from one sample and only 100 reads from another sample. You would likely find more species in the sample that is deeply sequenced (50,000 reads) in comparison to the one that was shallowly sequenced (100 reads).

Below is an example where we are going to sequence one sample three times. Each colored dot represents a microbe and each color represents a different species. Through the process of DNA isolation, 16S PCR amplification, sequencing, quality trimming and OTU picking we can lose information or sequences.



To prevent any bias we may see in our diversity analysis we can rarefy our data. A rarefaction is a random collection of sequences from a sample, with a specified number of sequences (depth). For example, a rarefaction with a depth of 1000 reads per sample is a simulation of what your sequencing results would look like if you sequenced exactly 1000 reads from each sample. By rarefying our OTU table we can fairly measure alpha diversity across samples.

Exploring Rarefied Data with Alpha Diversity

In QIIME, we can first explore our data by looking at alpha diversity across multiple different sequencing depths. This task is performed using the `alpha_rarefaction.py` command that takes your OTU table and makes a directory full of many OTU tables, all of which are repeats of rarefactions at specfifc depths. The output of this command allows us to visualize how measurements in alpha diversity will change across a range of sequence depths per sample. Once we know how the diversity changes with depth, we can create one final rarefied table to use for our alpha diversity and beta diversity calculations.

The command is run using these parameters:

```
alpha_rarefaction.py
-i input_file_path
-o output_directory
-t tree_file_path
-m metadata_file_path
```

The input file path is to the otu table in .biom format that you want to rarefy (-i). The output directory is where you want your final results to be (-o). The tree file path is the location of the GreenGenes 97 percent phylogeny tree (-r).

The `alpha_rarefaction.py` command will do multiple things:

(1) Create multiple rarefied OTU tables at ten step increments (+ ten sequences each time) starting at a minimum level of ten sequences and stopping at the median number of sequences per sample

(2) Run `alpha_diversity.py` on each of the rarefied OTU tables using the chao1, observed_species, and phylogenetic distance metrics

(3) Collates the results for each metric at the various depths into one table per metric, within the `alpha_div_collated/` subdirectory

(4) Plots the different metrics for each category in the metadata file and places those within the `alpha_rarefaction_plots/` subdirectory

(5) Deletes all of intermediate the OTU tables it had generated to do the analysis

(6) Creates a log file and overall rarefaction plot within the main output directory

To run this QIIME command, we must submit a job. Your job file should look like this:

```
#!/bin/bash -l
#PBS -l nodes=1:ppn=16,mem=2Gb,walltime=6:00:00
#PBS -m abe
#PBS -M x500@umn.edu
#PBS –o job_name_stout
#PBS -e job_name_stderr

cd /home/bioltrm1/x500

module load qiime/1.8.0

alpha_rarefaction.py
-i file/path/to/otu_table.biom
-o file/path/to/whatever_you_want
-t /home/biol1961/shared/97_otus.tre
-m /home/biol1961/shared/map.txt
```

Of course, you have to modify the file to specify YOUR file paths and the outputs to what you want. Things that cannot change include:
* The first 3 lines. The nodes and ppn must be 1 and 16 for the lab queue * The module load qiime/1.8.0 We must always load QIIME and it must be qiime/1.8.0 * The –t must be to the 97 percent GreenGenes tree

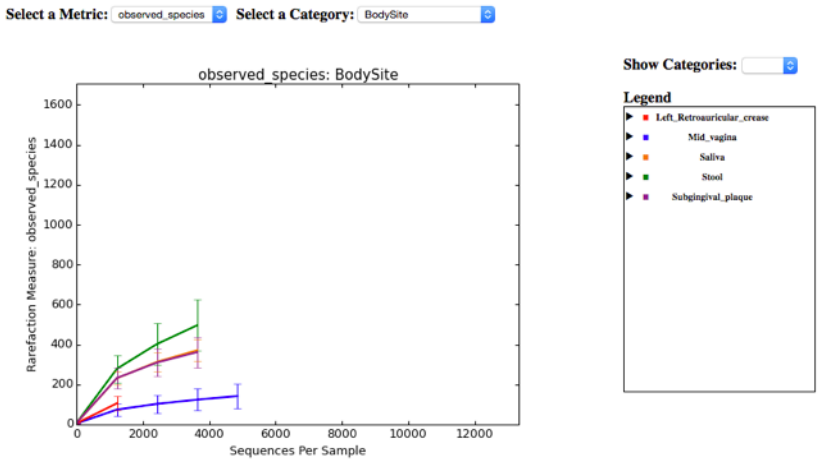
All the parameters for the actual command must be all on one line in the job file, with a space between the parameter letter and value. In the above example they are on separate lines so that you can read them easily. As mentioned above, the script will run its own jobs after it has started. You will know `alpha_rarefaction.py` is done when you have the following files in your output directory (-o):

`alpha_div_collated/` (one table per metric in here)
`alpha_rarefaction_plots/` (plots per metadata column)
`log_##.txt` (log file)
`rarefaction_plots.html` (overall plot)

You will also have some files that were generated by the jobs submitted by this script. They include: **job_name_stout** (the standard out captured by the job submission)
job_name_stderr (the errors captured by the job submission)

If you want to run the command again, you should delete these files prior to submitting another job.

To look at your plots, you must transfer the **entire** `alpha_rarefaction.py` output folder from MSI to your computer. The `rarefaction_plots.html` file needs other information supplied within the subfolders. You can move through all of the plots by selecting different categories and diversity metrics.



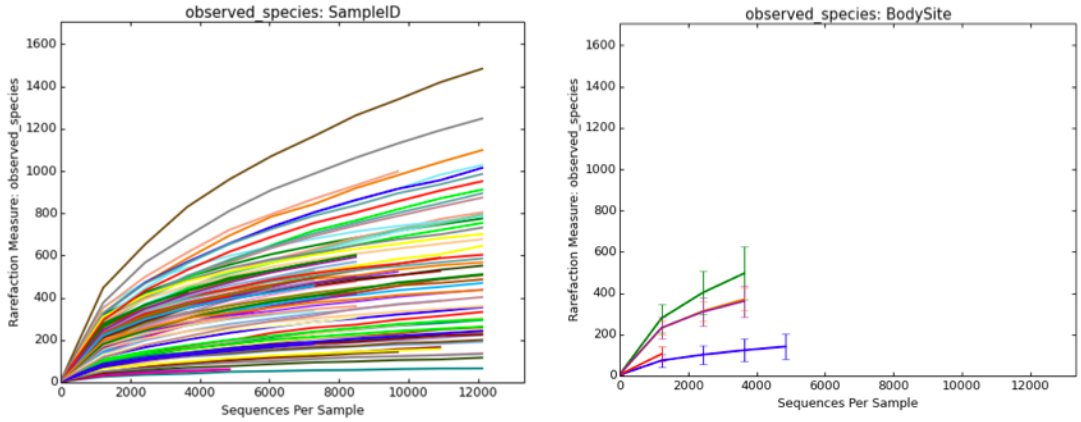
Creating a Rarefied OTU Table

How do you pick a depth?

You want to pick a depth that:

(1) Keeps as many samples as possible (isn't too high)

(2) Isn't so low that samples aren't representative of the total diversity



In the above examples the left plot shows all samples and the right plot shows the mean number of observed species according to body site. If we chose a depth of 2000 sequences, we would loose some the orange samples on the right (saliva). We know this because on the group plots, the line will stop where at least one sample in that group no longer has that many sequences. The depth you choose is entirely up to you! But you should be able to justify why you pick it. Normally we try to pick a depth where the rarefaction curves begin to level off. This means for each increase in the number of sequences, we are not detecting any (or very few) new OTUs. In the above example we would probably rarefy at no lower than 1,500.

Creating your Rarefied OTU Table

Once you have picked a depth based on the `alpha_rarefaction.py outputs`, you are ready to create a rarefied OTU table. To do this, we use the `single_rarefaction.py` command with the following parameters:

```
single_rarefaction.py
-i input_file_path
-o output_file_path
-d number_of_sequences
```

The input file path is to the OTU table in .biom format that you want to rarefy (-i). The output file path is where you want your final rarefied OTU table to be (-o, make sure the name is different from the original!). Both the input and output will be .biom files. The last parameter, (-d) is the depth you have chosen based on the plots generated by alpha_rarefaction.py.

Remember, all the parameters for the actual command must be all on one line in the job file, with a space between the parameter letter and value and must be specific to you.

The output of this command will be the OTU table you use for alpha and beta diversity calculations.

Filtering OTU Tables

Why do we Filter Samples From an OTU Table?

Filtering low depth samples from an OTU table can be used as an alternative for rarefying an OTU table. Rarefying results in taking only a small fraction of the original data. It causes an increase in two types of error:

Type I

* "Decreased specificity" or an increased likelihood in saying two groups are different when they aren't * Caused by rarefied samples remaining over-dispersed (a small number of sequences come from a variety of sources)

Type II

* "Loss of power" or "decreased sensitivity" to detect real differences between groups * Caused by valuable information about diversity being thrown out

Let's discuss rarefaction by looking at the biom summary for an original OTU table and the rarefied version.

```
Num samples: 104
Num observations: 9957
Total count: 1443869
Table density (fraction of non-zero values): 0.053
Table md5 (unzipped): c3ea77ea5cb942958ee7258abf7309ba

Counts/sample summary:
 Min: 4.0
 Max: 53844.0
 Median: 12121.000
 Mean: 13883.356
 Std. dev.: 8912.638
 Sample Metadata Categories: None provided
 Observation Metadata Categories: taxonomy

Counts/sample detail:
 700102569: 4.0
 700107977: 1933.0
 700105480: 2549.0
 700032671: 3716.0
 700097437: 4535.0
 700034196: 4645.0
 700097859: 4692.0
 700035264: 4802.0
 700107920: 4943.0
 700035697: 4973.0
 700097781: 5329.0
 700035184: 5464.0
 700102995: 5663.0
 700034228: 5968.0
 700097477: 6132.0
 700108145: 6264.0
 700114637: 6618.0
 700108866: 6781.0
 700097163: 6967.0
 700099539: 7051.0
```

```
Num samples: 102
Num observations: 5574
Total count: 204000
Table density (fraction of non-zero values): 0.042
Table md5 (unzipped): d7d478170e6d85ccd79924fb41f2960e

Counts/sample summary:
 Min: 2000.0
 Max: 2000.0
 Median: 2000.000
 Mean: 2000.000
 Std. dev.: 0.000
 Sample Metadata Categories: None provided
 Observation Metadata Categories: taxonomy

Counts/sample detail:
 700111532: 2000.0
 700034196: 2000.0
 700111841: 2000.0
 700099408: 2000.0
 700110689: 2000.0
 700107806: 2000.0
 700101335: 2000.0
 700114832: 2000.0
 700103653: 2000.0
 700108086: 2000.0
 700114444: 2000.0
 700106637: 2000.0
 700109608: 2000.0
 700037714: 2000.0
 700023313: 2000.0
 700108145: 2000.0
 700097437: 2000.0
 700035697: 2000.0
 700014994: 2000.0
 700038978: 2000.0
```

Biom Summary of Original OTU table | Biom summary of rarefied OTU table (2000 seqs)

Notice how in the above example, the first samples in the original OTU table are low in sequence number. One sample has only 4 sequences. The last sample listed has over 7,000 sequences, and that is still low compared to the rest of the sequences in the dataset (the median is 12,121 sequences). The table on the right is the same OTU table rarefied to 2,000 sequences. Notice that the samples below 2000 sequences are thrown out, and that the remaining samples are subsampled to an even depth of 2,000. That means most samples have lost about 10,000 sequences! That's is a lot of information to throw out!

We should note, that **rarefying our data is still the gold standard when measuring alpha and beta diversity**. When looking for specific taxa, however, we can filter low-depth samples from our OTU table and keep the full depth of sequences for the rest of the samples. We can then account for differences in sequencing depth by transforming the data later. We accomplish the filtering of low-depth samples through the `filter_samples_from_otu_table.py` command in QIIME. Normally, we keep samples that have >1,000 sequences and throw out the others. You can choose to go higher or lower depending on the sequencing results.

How do We Filter an OTU Table?

In QIIME, this task is performed on your OTU table. The QIIME command `filter_samples_from_otu_table.py` takes your OTU table and makes a new version of both based on the filtering parameters you set.

The command is run using these parameters:

```
filter_samples_from_otu_table.py
-i input_file_path
-o output_file_path
-n number_of_sequences
```

The input file path is to the original OTU table in .biom format that you want to filter (`-i`). The output file path is where you want your filtered OTU table to be (`-o`). The minimum number of sequences a sample must have to remain in the OTU table is set with the last parameter (`-n`).

There are other options you can use to filer your OTU table, such as:

```
-s valid_states
--sample_id_fp path_to_text_file
--negate_sample_id_fp path_to_text_file
-m max_sequence_count
```

The valid states let you specify a mapping column and values in that column that a sample must be associated with to remain in the OTU table (`-s`).

For example, if we sampled people from different locations in the Twin Cities and the collection location was under a header called 'Location' in the mapping file. If we wanted to keep only samples collected from Uptown and Downtown and not St Paul, you could use: `-s Location:Uptown,Downtown`. You could also use a sample ID text file, with one sample ID per line, to list which samples to keep (`--sample_id_fp`), or which to be remove from the OTU table (`--negate_sample_id_fp`) . You can also filter using the maximum number of sequences a sample can have to remain in the OTU (`-m`).

For the full list of parameters and how to use them, you can look at the command page on the QIIME webpage: http://qiime.org/scripts/filter_samples_from_otu_table.html

The `filter_samples_from_otu_table.py` command will create a new OTU table in biom format containing only the samples that meet the filtering criteria.

To run this QIIME command, we can use QIIME interactively.
``` { r, eval=F} ssh lab
cd /home/biol1961/x500 module load qiime/1.8.0

filter_samples_from_otu_table.py -i file/path/to/otu_table.biom -o file/path/to/whatever_you_want.biom -m home/biol1961/shared/map.txt --output_mapping_fp file/path/to/whatever_you_want.txt -n number_of_sequences ```

Of course, you have to modify the file to specify YOUR file paths and the outputs to what you want. For the HMP dataset in the examples, the file path for the map would remain the same. All the parameters for the actual command must be all on one line in the job file, with a space between the parameter letter and value. In the above example they are on separate lines so that you can read them easily. **We should use a depth no lower than 1000 sequences.**
```

Converting Table Types

##.biom to .txt

We can convert .biom files to.txt using `biom convert` on MSI. To do so, **QIIME should be loaded for interactive use.** This command will convert the `.biom` file to a tab-delimited `.txt` file. It will take in the OTU table (in `.biom` format, -i) and will output a new `.txt` file (-o). Specifying `-b` means we are going from `.biom` to `.txt` and `--header-key specifies` it has taxonomy. Like other biom commands, this command must be run when QIIME has been loaded in interactive mode. Again, this command should be run all in one line, with the parameters separated by one space.

```
ssh lab

cd /home/biol1961/x500

module load qiime/1.8.0

biom convert
-i table.biom
-o table_from_biom_w_taxonomy.txt
-b
--header-key taxonomy
```

This text file of your OTU table will look something like this. The columns in the OTU table are the samples. The rows are the OTU IDs. The column header for the OTU IDs is always `#OTU ID`. In the text file, each column is separated with a 'tab'. When we open this tab-delimited text file in Excel, Excel knows to read each tab as a new column. The values in the OTU table are the counts for that OTU ID in each sample. For example, if we look at the first OTU (189503) we can see it occurs 34 times in sample A, 19 times in sample B, and so on. Notice that the last column is **not** a sample, it is the taxonomy.

```
# Constructed from biom file
#OTU ID sampleA sampleB sampleC sampleD sampleE sampleF sampleG sampleH taxonomy
189503  34      19      11      69      71      14      79      88      k__Bacteria; p__Firmicutes; c__Clostridia; o__Clostridiales; f__Clostridiaceae; g__; s__
35786   0       0       0       0       0       0       4       3       k__Bacteria; p__Firmicutes; c__Clostridia; o__Clostridiales; f__Ruminococcaceae; g__; s__
389067  0       0       1       0       0       0       0       0       k__Bacteria; p__Firmicutes; c__Bacilli; o__Lactobacillales; f__Lactobacillaceae; g__Lactobacillus; s__
782984  0       1       0       0       1       0       1       1       k__Bacteria; p__Firmicutes; c__Clostridia; o__Clostridiales; f__; g__; s__
189116  1       0       0       6       16      1       10      0       k__Bacteria; p__Firmicutes; c__Clostridia; o__Clostridiales; f__Ruminococcaceae; g__Ruminococcus; s__
181342  153     0       21      2       0       3       0       1       k__Bacteria; p__Firmicutes; c__Clostridia; o__Clostridiales; f__Clostridiaceae; g__02d06; s__
3589405 1       0       0       0       0       0       0       0       k__Bacteria; p__Firmicutes; c__Clostridia; o__Clostridiales; f__Ruminococcaceae; g__; s__
1026778 0       0       0       0       0       1       0       0       k__Bacteria; p__Proteobacteria; c__Gammaproteobacteria; o__Pasteurellales; f__Pasteurellaceae; g__Actinobacillus; s__parahaemolyticus
1974538 0       0       0       0       0       2       0       0       k__Bacteria; p__Proteobacteria; c__Betaproteobacteria; o__Burkholderiales; f__Alcaligenaceae; g__Sutterella; s__
4172237 0       0       0       1       0       0       0       0       k__Bacteria; p__Bacteroidetes; c__Bacteroidia; o__Bacteroidales; f__Bacteroidaceae; g__Bacteroides; s__
342844  449     0       90      0       0       228     60      0       k__Bacteria; p__Firmicutes; c__Clostridia; o__Clostridiales; f__; g__; s__
48899   0       0       0       0       0       0       0       1       k__Bacteria; p__Firmicutes; c__Clostridia; o__Clostridiales; f__; g__; s__
304239  0       0       0       1       0       0       0       0       k__Bacteria; p__Firmicutes; c__Clostridia; o__Clostridiales; f__Ruminococcaceae; g__; s__
537098  2       0       0       0       0       0       0       1       k__Bacteria; p__Firmicutes; c__Clostridia; o__Clostridiales; f__Lachnospiraceae; g__; s__
180563  0       0       1       0       0       0       0       0       k__Bacteria; p__Firmicutes; c__Clostridia; o__Clostridiales; f__Ruminococcaceae; g__; s__
4478898 0       0       1       0       0       0       0       0       k__Bacteria; p__Firmicutes; c__Clostridia; o__Clostridiales; f__Clostridiaceae; g__Clostridium; s__
915327  2       6       6       0       0       0       1       0       k__Bacteria; p__Firmicutes; c__Bacilli; o__Lactobacillales; f__Carnobacteriaceae; g__Granulicatella; s__
301064  0       0       0       0       1       0       0       0       k__Bacteria; p__Firmicutes; c__Clostridia; o__Clostridiales; f__Ruminococcaceae; g__Ruminococcus; s__flavefaciens
288810  0       0       1       0       0       0       0       0       k__Bacteria; p__Firmicutes; c__Clostridia; o__Clostridiales; f__Ruminococcaceae; g__; s__
348806  0       1       0       0       0       0       1       0       k__Bacteria; p__Firmicutes; c__Clostridia; o__Clostridiales; f__[Tissierellaceae]; g__ph2; s__
4375336 0       1       0       0       0       0       0       0       k__Bacteria; p__Firmicutes; c__Clostridia; o__Clostridiales; f__; g__; s__
```

.txt to .biom

We can convert out OTU tables from `.txt` to `.biom` using `biom convert` on MSI. This might be important to do after we filter and normalize our tables (if we want to use them with QIIME again). To do so, **QIIME should be loaded for interactive use.**

```
ssh lab

cd /home/biol1961/x500

module load qiime/1.8.0

biom convert
-i normalized_table.txt
-o normalized_table.biom
--table-type "OTU table"
--header-key taxonomy
```

(This should be written all on one line!)

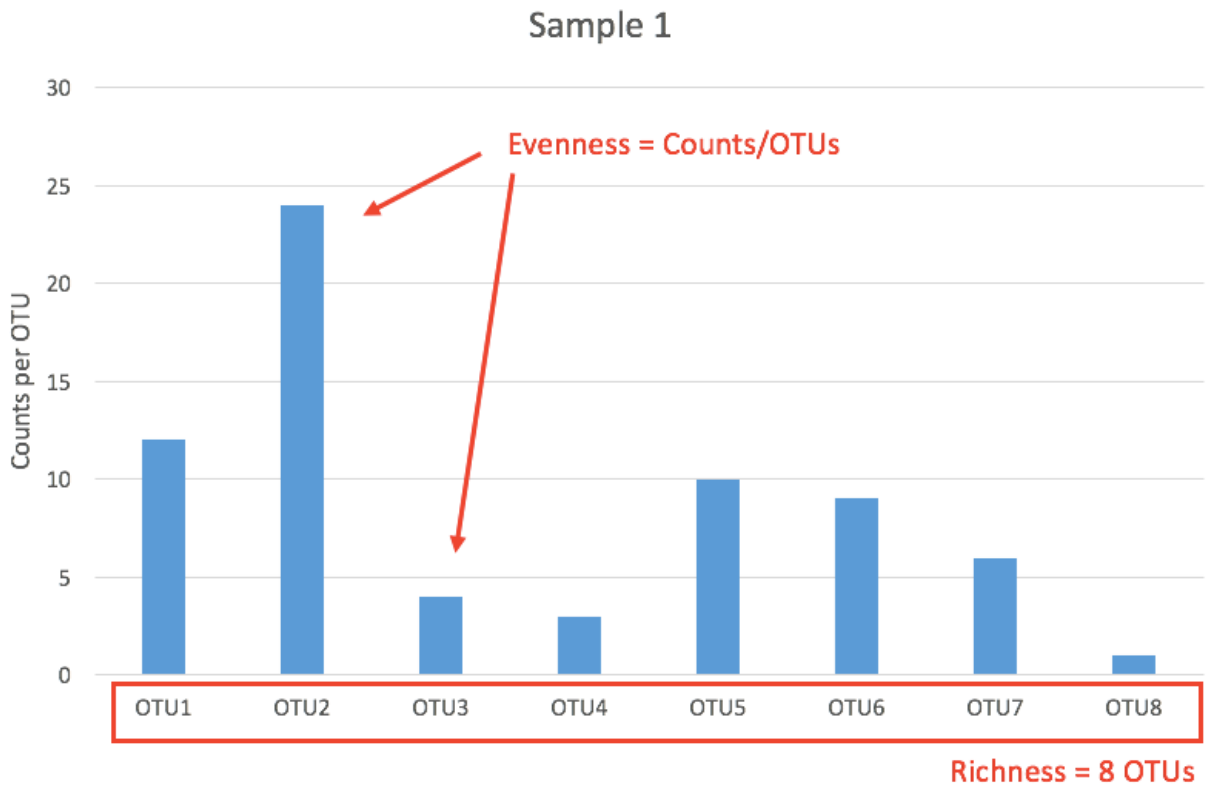
Alpha Diveristy

What is alpha diversity?

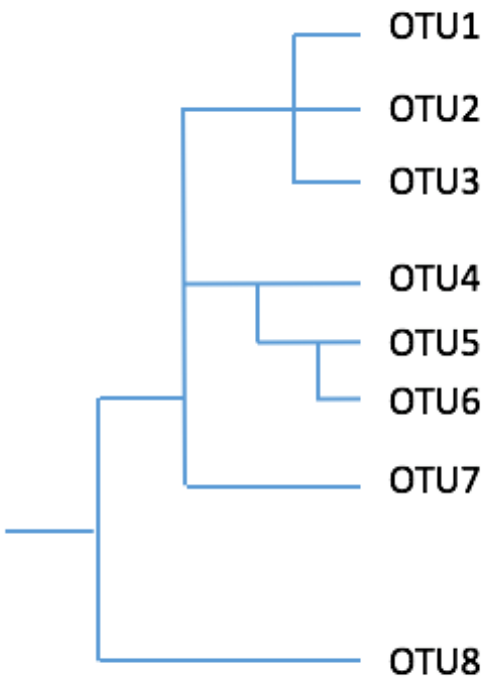
Alpha diversity measures how many different things are within a particular area or ecosystem, and is usually expressed by the number of species (i.e., species richness) in that ecosystem. In our case, the ecosystem in question is the sample type we are analyzing (stool, soil, skin.). It's important to remember that alpha diversity is **within a sample**, which is what makes it different than beta diversity, which we will talk about later in the course. The amount of diversity in any community is extremely important in determining ecological dynamics (e.g. community productivity, stability, and resilience). For humans, there is a great deal of data demonstrating that the ancestral human gut microbiome is more diverse than the modern one, and that this lower diversity is highly correlated with a number of important diseases. Therefore, alpha diversity is an important phenotype in microbiome research.

Different metrics have been developed to calculate alpha diversity. Some of these include:

Richness: A measure of the number of OTUs present in a sample
Evenness: How many of each OTU is present in a sample
Phylogenetic relationship: Accounts for taxonomy and phylogenetic relationships



Richness and evenness for one sample in a microbiome study.



Example of a phylogenetic tree.

What are Diveristy Metrics?

Below are some common alpha diversity metrics use in microbiome research. There are numerous other metrics available in QIIME, but we don't need to cover all of them for Biology 2002.

Observed Species

The simplest diversity index; it is just the number of OTUs.

Chao1 estimator

$$S_1 = S_{obs} + \frac{F_1^2}{2F_2}$$

This is commonly used, and is based upon the number of rare OTUs found in a sample The problem with this metric is that if a sample contains many singletons (OTUs that happen just once, usually by sequencing error) the Chao 1 index will estimate greater species richness than it would for a sample without rare OTUs. This problem is avoided if we first filter the rare OTUs from out OTU table. In the equation Sobs is the number of species in the sample, F1 is the

number of singletons (i.e., the number of species with only a single occurrence in the sample) and F2 is the number of doubletons, which is the number of species with exactly two occurrences in the sample (Colwell, et al. 1994).

Shannon index

$$\text{Shannon Index (H)} = -\sum_{i=1}^{s} p_i \ln p_i$$

This index accounts for both abundance and evenness of the species present. It assumes all species are represented in a sample. In the Shannon index, p is the proportion (n/N) of individuals of one particular species found (n) divided by the total number of individuals found (N), ln is the natural log, ?? is the sum of the calculations, and s is the number of species (CISN. 2010).

Simpson index

$$\text{Simpson Index (D)} = \frac{1}{\sum_{i=1}^{s} p_i^{\,2}}$$

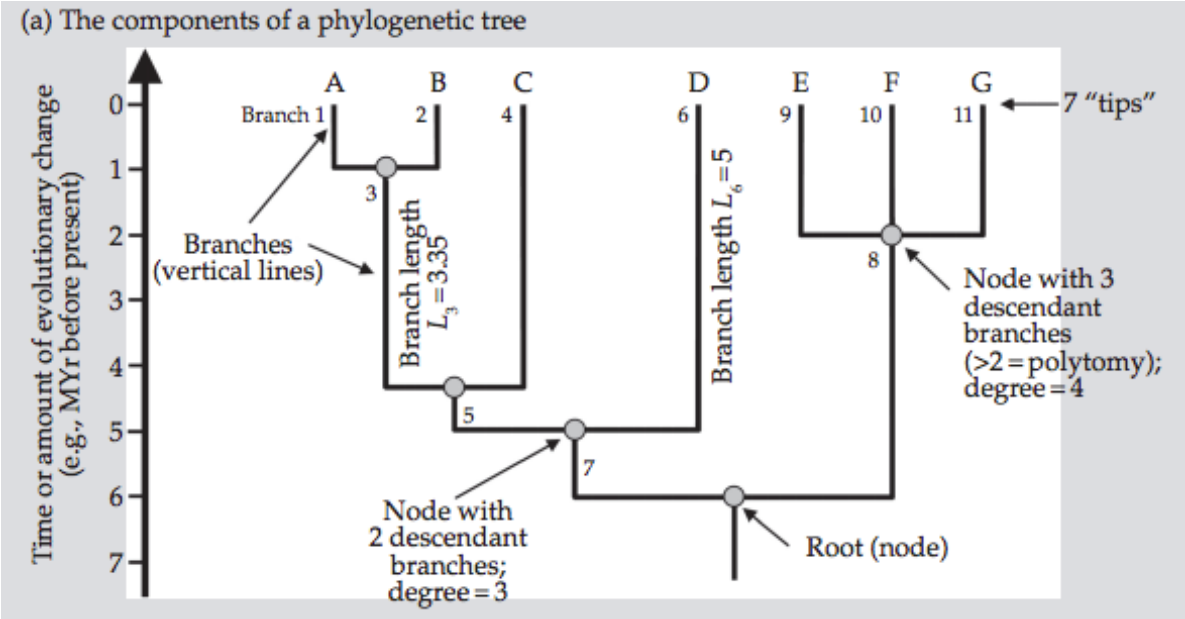
The Simpson index is actually a similarity index, so the higher the value the lower diversity in the sample. It gives more weight to common or dominant species. In the Simpson index, p is the proportion (n/N) of individuals of one particular species found (n) divided by the total number of individuals found (N), ?? is still the sum of the calculations, and s is the number of species (CISN,. 2010).

Phylogenetic Distance (PD Whole Tree)

$$\text{PD} = \sum_{i=1}^{S} \text{ED}_i$$

$$ED(T,i) = \sum_{e \in s(T,i,r)} \left(\lambda_e \cdot \frac{1}{S_e}\right)$$

The phylogenetic distance metric used most often is PD whole tree. It is the sum of all phylogenetic branches connecting OTUs together within a community. PD is the sum of ED for each species (i) in the sample. ED is the evolutionary distinctiveness. It is calculated by the second equation where for species i in tree (T), ED is the sum of edge of length ??? in the set s(T,i,r) connecting species i to the root (r) and Se is the number of species that descend from edge e (Cadotte, et al. 2010). Below is a figure showing the components of a phylogenetic tree for reference.



The components of a phylogenetic tree (Vellend, et al. 2011.)

So, as you can see, each one of the diversity metrics is slightly different, each with it's advantages and disadvantages. In terms of measuring richness and evenness, each metric is summarized below.

Summary of Diversity Metrics

Metric	Measurement
Observed Species	Richness
Chao1	Richness & Evenness
Shannon	Richness & Evenness
Simpson	Richness & Evenness
PD Whole Tree	Phylogeny

How do we calculate alpha diversity in QIIME?

In QIIME, we can use our rarefied or filtered OTU table to calculate alpha diversity. Earlier we used alpha diversity metrics to determine a reasonable rarefaction depth or a reasonable sequencing depth as filtering cutoff. Now we will use the `alpha_diversity.py` command in QIIME to make a final alpha diversity calculation for each sample. The command is run using these parameters:

```
alpha_diversity.py
-i file/path/to/otu_table.biom
-o file/path/to/alpha_diversity.txt
```

```
-m metrics,to,use
-t file/path/to/tree
```

The input file path is to the filtered or rarefied otu table in `.biom` format (`-i`). The output file path is where you want your alpha diversity table to be (`-o`). The metrics are what you would like to use as an estimate of diversity, and should be a comma separated list with **no** spaces (`-m`). The tree file path is to the GreenGenes 97% OTU tree (`-t`). For a full list of the metrics available and how to spell them, you can type:

```
alpha_diversity.py -s
```

The output is:

```
Known metrics are: ace, berger_parker_d, brillouin_d, chao1, chao1_ci,
dominance, doubles, enspie, equitability, esty_ci, fisher_alpha, gini_index,
goods_coverage, heip_e, kempton_taylor_q, margalef, mcintosh_d, mcintosh_e,
menhinick, michaelis_menten_fit, observed_otus, observed_species, osd,
simpson_reciprocal, robbins, shannon, simpson, simpson_e, singles, strong,
PD_whole_tree
```

For the full list of parameters and how to use them, you can look at the command page on the QIIME webpage: http://qiime.org/scripts/alpha_diversity.html

How do we run aplha_diversity.py?

To run this QIIME command, we can use QIIME interactively. **Note: You should be using your rarefied OTU table!**

```
ssh lab
cd /home/biol1961/x500
module load qiime/1.8.0
 alpha_diversity.py
-i file/path/to/otu_table.biom
-o file/path/to/alpha_diversity.txt
-m shannon,simpson,choa1,PD_whole_tree
-t /home/biol1961/shared/97_otus.tree
```

All the parameters for the `alpha_diveristy.py` command must be all on one line, with a space between the parameter letter and value. In the above example they are on separate lines so that you can read them easily.

What does alpha_diversity.py give us?

The output of the alpha_diversity.py command is a table, where the columns are the different diversity metrics and the rows are samples. We can then use this table to make alpha diversity plots, to visual our findings. We can also test to see if the alpha diversity is significantly different between and across different sample types.

References

Cadotte MW, et al. 2010. Ecology Letters.13: 96-105.

Colwell, R.K. and Coddington, J.A. 1994. Philosophical Transactions of the Royal Society: Biological Sciences. 345:101-118.

Community Invasive Species Network (CISN). 2010. How to Calculate Biodiversity. http://www.protectingusnow.org

Vellend M, et al. 2011. Biological diversity: frontiers in measurement and assessment. Oxford University Press.

Beta Diversity

What is Beta Diversity?

In his 1972 publication in Taxon, "Evolution and Measurement of Species Diversity", R. H. Whittaker laid out the terms and concepts for how we think about and define biodiversity. His idea was that the total species diversity in a landscape (γ or gamma-diversity) (e.g. ALL human gastrointestinal (GI) tracts) is determined by two different things:

| --------|---------- **1) Alpha diversity** | the mean species diversity at the habitat level | α | *e.g.* one person's GI tract **2) Beta diversity** | the differentiation among habitats | β | *e.g.* different people's GI tracts

The total diversity, gamma, is alpha multiplied by beta: $\gamma = \alpha * \beta$

We have already discussed alpha diversity and have compared the average alpha diversity of samples across body sites. We found that indeed, there are significant differences in alpha diversity between body sites. Now, we are interested in looking at the difference (the ecological distance) in the community members between samples (e.g. individuals) and groups of samples (e.g. body sites).
For example, let's say you are comparing the biological communities of a 20m2 patch of the Great Barrier Reef (right) and a 20m^2 of the Amazon rainforest (left).

Both of these habitats have very high alpha (α) diversity. However, despite similarly high alpha diversity, if you were to compare the composition these two communities at the macroscopic level, they are almost completely non-overlapping. Therefore, they would also have a very high beta diversity (β). This however, is a very extreme example.

Let's say that instead of comparing a single patch of coral reef and a single patch of rainforest, you compare multiple patches of 5 different coral reefs to each other and multiple patches of 5 different rainforests to each other. You might find that the average alpha diversity is about the same for coral reefs and rainforests, but beta diversity is significantly higher for rainforests than

for coral reefs. That would mean that different rainforests have species that differ from each other.

In another example, you sequence the GI tract microbiota of 100 healthy adults. Fifty individuals have been taking regular low doses of aspirin for the past 30 days. The other half of study subjects have been taking a placebo. You find that the alpha diversity for the treatment group is not significantly different from the control group. However, the beta diversity for the treatment group is significantly higher. What would that mean?

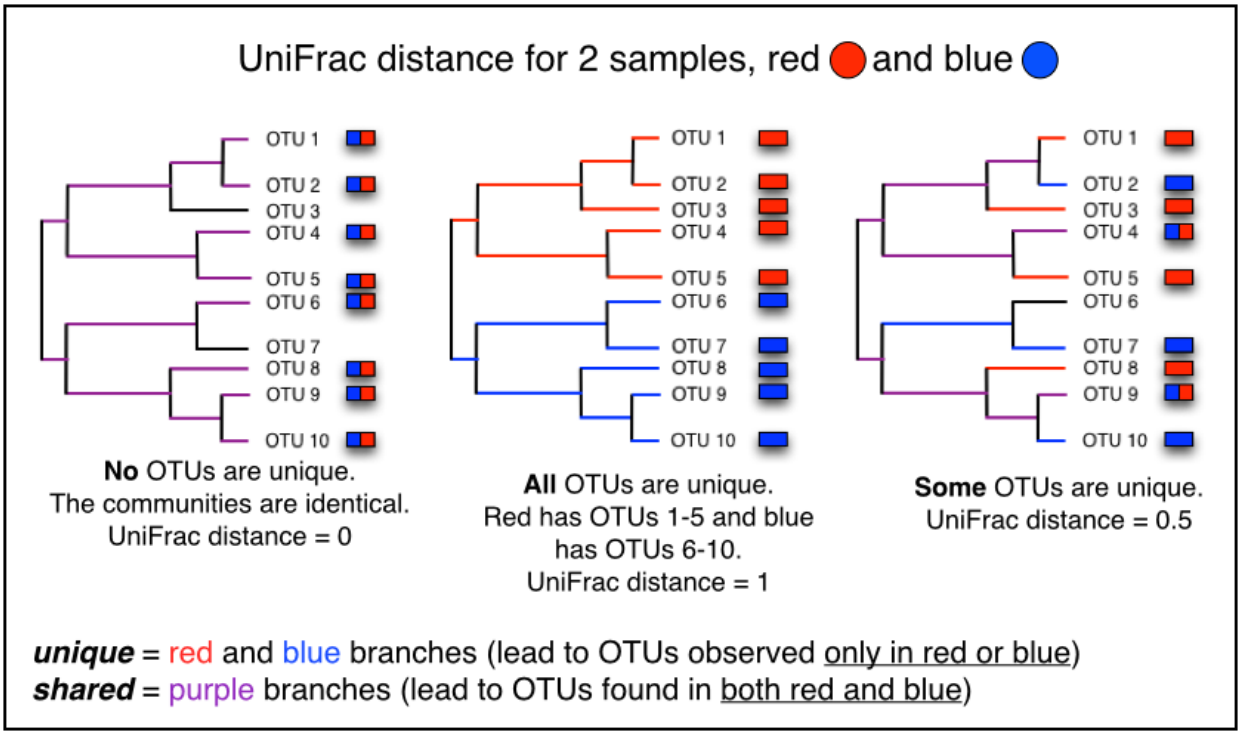
What are Beta Diversity Metrics?

If you remember there were multiple diversity metrics that we used for alpha diversity. Similarly, there are multiple beta diversity metrics. Below we will cover the most widely used distance metrics for beta diversity.

UniFrac Distance

$$U_{AB} = \frac{unique}{observed}$$

This is the most widely used index. The unique fraction metric, or UniFrac, measures the phylogenetic distance between sets of taxa in a phylogenetic tree. It counts the branch lengths of the tree that lead to taxa from either one environment or the other, but not both (Lozupone 2005). In the equation, UAB is the UniFrac distance between sample A and B, where unique = total unique branch length (cumulative branch lengths that lead to OTUs observed only in sample A or sample B) and observed = total branch length (cumulative branch lengths that leads to all OTUs in samples A or B). This metric is sensitive but also has emphasis on minor differences in the tree (Fukuyama, 2012).



Weighted Unifrac Distance

In the above example, the relative abundances of taxa is not taken into consideration (referred to as unweighted UniFrac distance). There is a second metric known as weighted UniFrac distance, that weights each OTU based on it's relative abundance. Both metrics are criticized for giving either too much (unweighted) or too little (weighted) value to rare taxa, but both have value in showing different aspects of community diversity.

Bray-Curis Dissimilarity

$$BC_d = \frac{\sum \left| x_i - x_j \right|}{\sum \left(x_i + x_j \right)}$$

Bray-Curtis takes the sum of the differences in OTU abundances over the sum of the total OTU abundances between samples. In the equation xi is the abundance of OTU x in sample i, and xj is the abundance of OTU x in sample j. If an OTU is absent then its abundance should be recorded as zero. The Bray-Curtis metric ranges from 0 to 1, where 0 means the two samples have the same composition and 1 means the two samples do not share any OTUs (Gardener, 2016). This metric does not take relatedness of the otus into consideration (phylogeny).

So as you can see, each one of the diversity metrics is slightly different, each with it's advantages and disadvantages. In terms of measuring abundance and phylogenetic differences, each metric is summarized below.

Summary of Beta Diversity Metrics

Metric | **Phylogeny?** | **Abundance** ---------- | ---------- | ---------- Unweighted UniFrac | yes | no Weighted UniFrac | yes | yes Bray-Curtis | no | yes

How Do We Calculate Beta Diversity in QIIME?

We will use the `beta_diversity.py` command in QIIME to calculate beta diversity metrics between samples and groups. This command will return a matrix of the distances of all samples to all other samples. This can be visualized as a graph of points, a network, or any other creative method you can come up with. We should note that sequencing depth can have an effect on beta diversity analysis, just as it does on alpha diversity.

```
beta_diversity_through_plots.py
-i file/path/to/otu_table.biom
-o file/path/to/beta_diversity
-m file/path/to/mapping_file.txt
-t file/path/to/tree
-p file/path/to/parameters_file.txt
-a run_parallel
-O job_to_run
-e sequences_per_sample
```

The input file path is to the filtered or rarefied OTU table in .biom format (`-i`). The output file path is where you want your beta diversity output to be (`-o`). The tree file path is to the GreenGenes 97% OTU tree (`-t`). The metrics are what you would like to use as an estimate of beta diversity are supplied in the parameters file (`-p`). To run this in parallel (`-a`), we must specify the number of jobs (`-O`, the lab queue max is 6). If we didn't rarefy our OTU table, but want an even depth for all the samples we could also specify the depth (`-e`). For the full list of parameters and how to use them, you can look at the script page on the QIIME webpage: http://qiime.org/scripts/beta_diversity_through_plots.html

The `beta_diversity_through_plots.py` command will do multiple things:

1. Create jobs within the `jobs/` folder it creates, as well as output (`.o##`) and error files (`.e##`), and a `pbs_nodefile.txt` file (just like the otu picking script)
2. Randomly subsample `otu_table.biom` to even number of sequences per sample (specified with `-e`)
3. Run `beta_diversity.py` for the diversity metrics wanted (specified with the parameters file via `-p`) and create distance matrices in the main output directory (`metric_dm.txt`)
4. Perform a principal coordinates analysis on the result of Step 3 in the main output directory (`metric_pc.txt`)

5. Generate a 2D and 3D plots for all mapping fields in the `metric_emperor_pcoa_plot/` subdirectories
6. Deletes all of intermediate generated to do the analysis
7. Creates a log file and overall rarefaction plot within the main output directory

To run this QIIME command, we can use QIIME interactively on MSI:

```
ssh lab
module load qiime/1.8.0


beta_diversity_through_plots.py
-i file/path/to/otu_table.biom
-o file/path/to/whatever_you_want
-t /home/biol1961/shared/97_otus.tree
-m /home/biol1961/shared/map.txt
-p /home/biol1961/shared/parameters.txt
```

Of course, you have to modify the file to specify YOUR file paths and the outputs to what you want and it should all be on one line.

All the parameters for the actual command must be all on one line in the job file, with a space between the parameter letter and value. In the above example they are on separate lines so that you can read them easily.

As mentioned above, the command will run its own jobs after it has started. You will know `beta_diversity_through_plots.py` is done when you have the following files in your output directory (`-o`):

`metric_pc.txt` (one table per metric, 3 total)
`metric_dm.txt` (one table per metric, 3 total)
`metric_emperor_pcoa_plot` / (one per metric, 3 total)
`log_##.txt` (log file)

To look at your plots, you must transfer the entire plot folder from MSI to your computer. The plot file needs other information supplied within the subfolders.

Manipulating 3D Plots

Once you have moved the entire plot folder for your metric of choice to your computer, you can click on the `.html` file with the directory to load the plot. There are many parameters about the plot you can change.

| ---------- | ---------- **Colors** | Change what covariate to color by **Visibility** | Make some samples more transparent **Scaling** | Make some samples larger or smaller **Labels** | Add sample labels **Axes** | Change which principal coordinates are plotted **Options** | Change backgroung/axes colors and save as image

References

Fukuyama J, et al. 2012. Pacific Symposium on Biocomputing. 2012:213-24. Gardener M. 2016. DataAnalytics.org.uk. (http://www.dataanalytics.org.uk/Publications/Writers%20Bloc/Distance%20metrics.htm). Lozupone C and Knight R. 2005. Applied and Environmental Microbiology. 71:8228-8235.

Ordination

When we want to look at high-demension data, one way to easily visualize similarities and differences is ordination. The type of ordination plots we will learn about and generate are Principal Component (PCA) and Coordinate Analyses (PCoA).

What is PCA and PCoA?

It is a way of identifying patterns in data and expressing data in such a way as to highlight their similarities and differences. Since our data can be of high dimensions, finding the patterns can be hard and this is where PCA and PCoA are powerful tools for analyzing data. The other main advantage of PCA/PCoA is that once you have found these patterns in the data you can compress the data by reducing the number of dimensions and visualize it.

This concept of dimension reduction can be very tricky to grasp. Understanding all the math behind PCA and PCoA is out of the scope of this class. We will, however, try to understand how the data is reduced, what we are actually plotting, and how to to accomplish this in R.

Below are some tutorials to help us understand **PCA**.

1. Please read the following website:
https://georgemdallas.wordpress.com/2013/10/30/principal-component-analysis-4-dummies-eigenvectors-eigenvalues-and-dimension-reduction

2. Please read the following website:
http://setosa.io/ev/principal-component-analysis

PCA vs PCoA

From the websites listed above we have learned about PCA. So what's PCoA? PCoA is similar PCA, however, PCoA can handle distances generated from any similarity or dissimilarity measure, such as Bray–Curtis and both weighted and unweighted UniFrac metrics. PCoA can also handle quantitative, semi-quantitative, qualitative, and mixed variables.

Similar to PCA, PCoA produces a set of uncorrelated axes to summarize the variability in the data set. Each axis has an eigenvalue whose magnitude indicates the amount of variation captured in that axis. The proportion of a given eigenvalue to the sum of all eigenvalues reveals the relative 'importance' of each axis. A successful PCoA will generate a few (2-3) axes with relatively large eigenvalues, capturing most of the variation in the input data, with all other axes having small eigenvalues.

Interpretation of a PCoA plot is straightforward: objects closer to one another are more similar than those further away. Similarity or dissimilarity is defined by the measure used in the construction of the (dis)similarity matrix used as input.

PCoA can handle a wide range of data, but the original variables cannot be recovered. This is because PCoA takes a matrix derived from the original data as input and not the original variables themselves.

The beta_diversity_through_plots.py command gives us PCoA plots of our data. Later in the course we will also learn how to generate these plots ourselves in R.

Summarizing Taxa

What are Taxa Summaries?

Summarizing taxa is a way to visualize which taxa are found in our samples. When we summarize taxa we can use the various levels of taxonomy. The following levels are those denoted by GreenGenes for taxonomy.

Level | Taxonomy | Example ------ | ---------- | ---------- 1 | Kingdom | Bacteria
2 | Phylum | Actinobacteria
3 | Class | Actinobacteria
4 | Order | Actinomycetales
5 | Family | Streptomycetacaea
6 | Genus | Streptomyces
7 | Species | mirabilis

In QIIME, we can use levels 2-6 to summarize taxa. We can't use level 1 because that would result in no summary (all of our OTUs are bacteria). We also cannot use level 7, because using 97% identity of a 16S gene cannot resolve species from one another (for the most part). We summarize taxa with the `summarize_taxa_through_plots.py` command in QIIME.

How do we actually summarize taxa?

In QIIME, this task is performed on your rarefied or filtered OTU table. It must be a COUNT table, not relative abundance. The QIIME command `summarize_taxa_through_plots.py` takes your OTU table and collapses the table into the various taxonomic levels. It will then plot the taxa summarize for us.

The command is run using these parameters:

```
summarize_taxa_through_plots.py
-i file/path/to/otu_table.biom
-o file/path/to/summary_output
-m file/path/to/mapping_file.txt
-c category_to_use
```

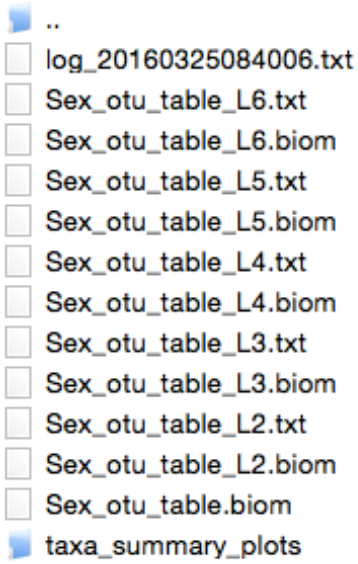
The input file path is to the filtered or rarefied OTU table in `.biom` format (`-i`). The output file path is where you want your taxa summary directory to be (`-o`). The mapping file path is the location of the mapping file (`-m`). The category you would like to use for the summarize must be a column header in the mapping file (`-c`). If you leave this parameter out, QIIME will make the summaries and plots using the entire OTU table.

For the full list of parameters and how to use them, you can look at the command page on the QIIME webpage: http://qiime.org/scripts/summarize_taxa.html

The `summarize_taxa_through_plots.py` command will:

1. Create an output directory named whatever you specified for `-o`
2. Create OTU tables collapsed at each taxonomic level (2-6) with samples grouped according to your `-c` parameter
3. Create taxa summary plots in a subdirectory called `taxa_summary_plots`

Below is an example of the contents of the output directory. The taxa summary was produced using the category 'sex' from the mapping file. If you want to look at the taxa summary plots, you must move that entire subdirectory to your personal computer to view the html plots.

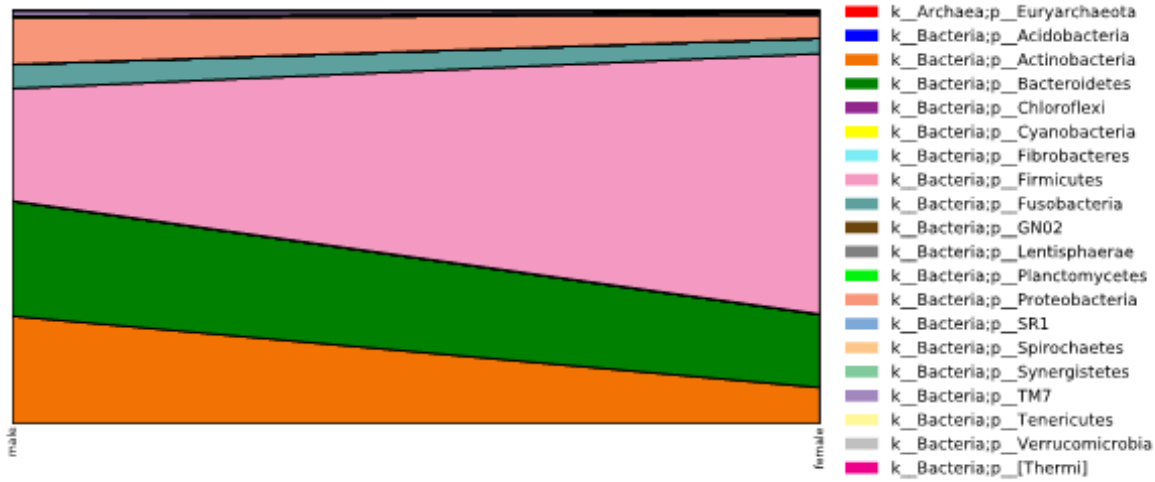


To run this QIIME command, we must submit a job file. Your job file should look like this:

```
#!/bin/bash -l
#PBS -l nodes=1:ppn=16,mem=2Gb,walltime=6:00:00
#PBS -m abe
#PBS -M x500@umn.edu
#PBS -o job_name_stout
#PBS -e job_name_stderr
cd /home/biol1961/x500
module load qiime/1.8.0
summarize_taxa_through_plots.py
-i file/path/to/otu_table.biom
-o file/path/to/whatever_you_want
-m file/path/to/mapping_file.txt
-c category_to_use
```

All the parameters for the actual command must be all on one line in the job file, with a space between the parameter letter and value. In the above example they are on separate lines so that you can read them easily.

Below is an example of what the taxa summary plots from QIIME look like. This example is using the category 'sex' from the mapping file. The plots produced by QIIME are not pretty or easy to read. To make easier to interpret taxa summary plots, we can use R.



Example taxa summary from QIIME

Plotting and Statistics

"There is no statistical tool that is as powerful as a well chosen graph"
-Chanmbers et al. 1983

When we begin to analyze our data it is important to be able to visualize our observations. Why is this?

- Plots are more effective in creating interest and in appealing the attention of others
- Visual relationships are more easily grasped and remembered
- Plots save time, since trends and differences can be visualized at a glance
- Plots can bring out hidden trends and relationships and aid in analytical thinking

Variables and Data Types

Before we try and visual our data we need to identify our variable and data types. This list is not exhaustive, but includes the main characteristics we need to think about. When we talk about our data, usually the dependent variable will be a measurement of the OTUs or taxa (for example, alpha diversity measurements) and the independent variable will be something from our mapping file. Sometimes we will call the independent variable in the mapping file a **covariate**, which is a variable that might be predictive of the outcome of the study.

Variables

Dependent Variables

are what we measured in the experiment and what were affected during the experiment. The dependent variable responds to the independent variable. You cannot have a dependent variable without an independent variable. On a graph, this is the y variable.

Independent Variables

are the variables we have control over, what we can choose and manipulate. They are usually what we think will affect the dependent variable. In some cases, we may not be able to manipulate the independent variable. It may be something observational that is already there and is fixed (sex, disease status, color). On a graph, this is the x variable.

Data Types

Qualitative

data is descriptive, it is observed and not measured. It is often categorical (color, smell, taste). Quantitative data is numeric and can be counted or measured (length, height, volume, weight).

Discrete

data can only take on a finite number of values, and is counted. All qualitative variables are discrete. Some quantitative variables are discrete, such as disease score rated as 1,2,3,4, or day sampled if people were only sampled on a specific finite number of days (day 1 and 15 only).

Continuous

data can take on any value in a certain range. No measured variable is truly continuous, however, discrete variables measured with enough precision can often be considered continuous for practical purpose (like age measured per day, or weight).

Types of Plots

There are numerous types of plots we can use. Here are a few very common types of plots, and a brief explanation as to what type of data they use, what they display, and when we should use them.

Pie Chart

Pie charts are used with discrete independent variables. Pie charts are best to use when you are trying to compare parts of a whole (percentage or proportional data). Pie charts should be used for displaying data with no more than 6 categories. They do not show changes over time. They are not used often in scientific research.

Bar Chart

Bar graphs are used with discrete independent variables. Bar graphs can be horizontal (x axis on side) or vertical (x axis is on the bottom). The height of each bar (dependent variable, y variable) are scaled according to their values and the bars can be compared to each other. Bar graphs have a space between each bar. Stacked bar charts can be used to compare overall quantities across items while illustrating the contribution of each category to the total.

Histogram

Histograms are used with continuous independent variables. Histograms can be horizontal (x axis on side) or vertical (x axis is on the bottom). The height of each bar (dependent variable, y variable) are scaled according to their values and the bars can be compared to each other. Histograms do not have a space between each bar.

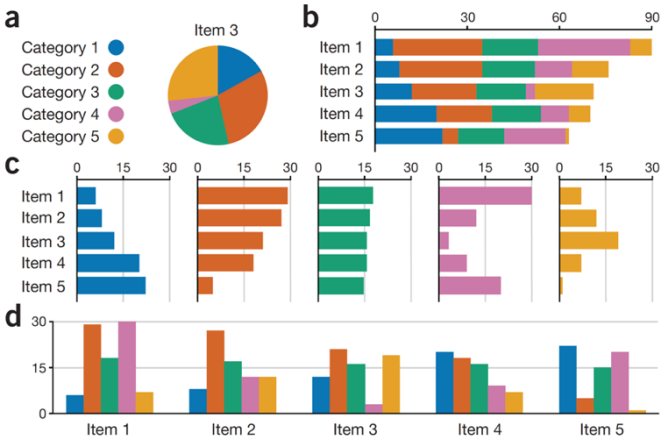


Figure 1. Examples of bar charts and a pie chart encoding the same data. *(a) Values in different categories are difficult to compare in pie charts. (b) Stacked bar charts enable*

comparison of overall values across items. (c) Layered bar charts support comparison of values within categories. (d) Grouped histograms allow comparison of values across categories (Streit and Gehlenborg, 2014).

Boxplot

Box plots are used with discrete independent variables. Box plots can be horizontal or vertical. Box plots show the full range of variation (from min to max), the likely range of variation (the interquartile range, IQR), a typical value (the median) and outliers (values 3 times the IQR). They provide more information than a bar chart.

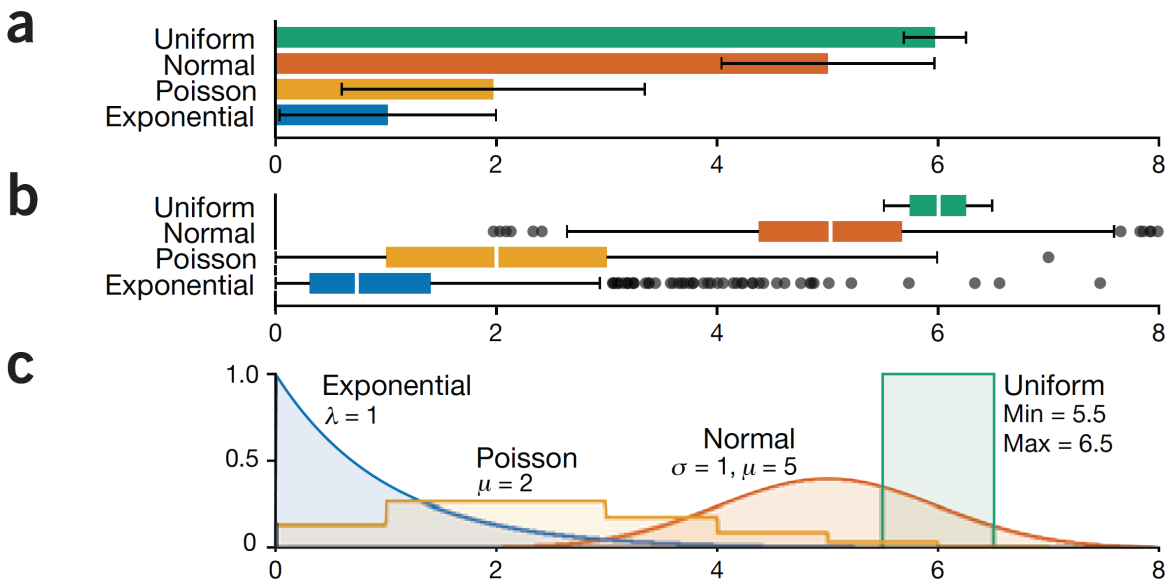


Figure 2. A comparison of bar graphs and box plots. *(a) Bar chart showing sample means with standard-deviation error bars. (b) Box plot with whiskers extending to ? 1.5 times the interquartile range. (c) Distributions of the different data sets. (Streit and Gehlenborg, 2014).*

Scatter Plot

Scatter plots are used when both the independent and dependent variables are quantitative. They show how much one variable is affected by another, also called their correlation. The closer the data points come when plotted to making a straight line, the higher the correlation between the two variables, or the stronger the relationship. Scatter plots can also help us see data that cluster together in certain areas of the scatter plot.

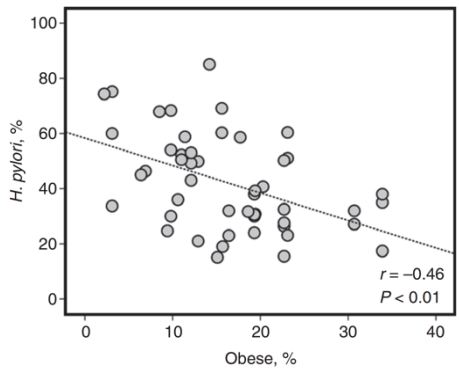


Figure 3. An example scatter plot of the percent of the microbiome that is H. Pylori and obesity *(Lender et al, 2014).*

Line Graph

Line graphs are used when both the independent and dependent variables are quantitative. Line graphs are like scatter plots except a line is created connecting each data point together. This emphasizes local changes from one point to the next. Unlike scatter plots, line graphs do not usually help us detect correlations, as the line emphasizes point-to-point changes.

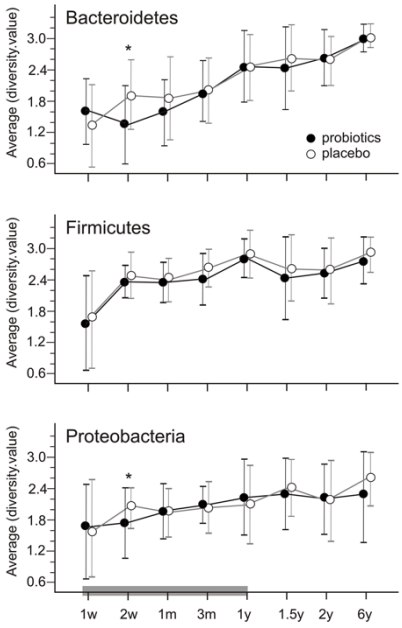


Figure 3. An example of a line graphs. *These graphs emphasize changes in specific bacterial taxa with and without probiotic supplementation (Rutten et al, 2015).*

Statistical Analysis

"Humans aren't too good at discerning subtle patterns that are really there, but equally so at imaging them when they are all together absent."
- Everitt and Hothorn, 2010

Summary Statisitcs

Mode
- data value that occurs most frequently

Median
- Data value that occurs at the precise middle of all data points

Mean
- Numerical average of all the data points

Variance
- Measure of the spread of the data, and is the average of the squared differences from the mean

Standard Deviations
Square root of the variance

Interquartile Range
- Where the 'middle' fifty percent of the data are located

Distribution
- Listing or function showing all the possible values of the data and how often they occur. Includes normal, skewed, uniform, Poisson and others (See Figure 5)

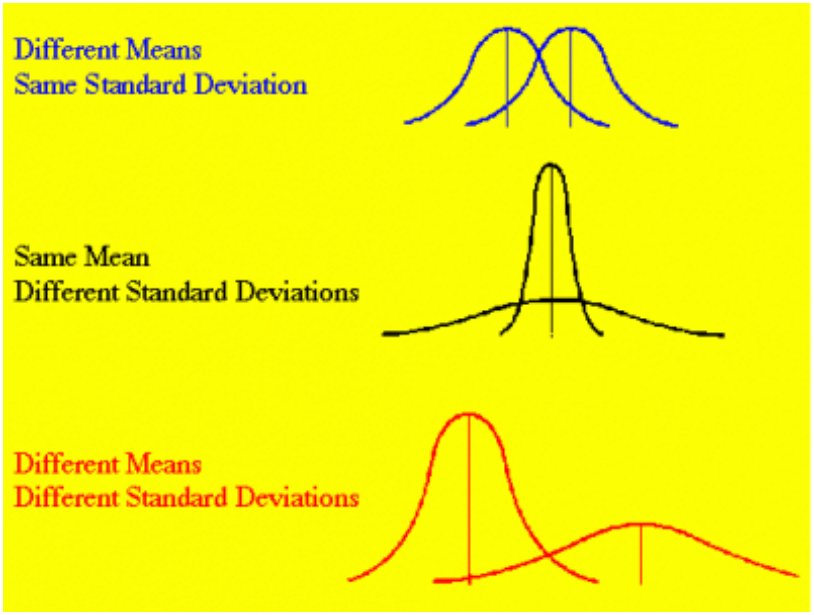


Figure 5. Examples of various distributions with different standard deviations.

Statistical Tests

Parametric Statistics

Parametric analyses are the oldest and most commonly used type of analysis. We will cover the three most common: correlation, t-test, and analysis of variance. All parametric statistics have three common assumptions that must be met before proceeding. In Figure 5, the top example could be tested with a parametric test.

1. All observations are independent of other observations (product of experimental design, no test needed). 2. The data are normally distributed (easily tested by examining the distribution). 3. The variances in the different treatment groups are the same. (Requires a test, such as the F-Max Test)

Student's t-Test

This analysis is used when you are comparing two different samples. A Student's t-test will report a t-statistic and a probability value (p-value). If the p-value is greater than or equal to our our alpha (usually 0.05) we reject our null hypothesis that there is a significant difference between the groups.

Analysis of Variance (ANOVA)

Analysis of variance is used to determine if differences exist between more than two treatment groups. The assumptions of ANOVA are identical to the t-test and the calculated statistic is called an F-value, with a corresponding p-value. As with the t-test, if our probability value is less than 0.05 we reject our null hypothesis (in this case that there is no difference among the treatment groups). This p-value only tells us if there are significant differences among our groups. It does not tell us where these differences are.

Regression

Regression is used to determine whether two variables are related. A highly used regression method is Pearson's r. The r statistic has a range of values from -1.00 (a perfect negative correlation) to 1.00 (a perfect positive correlation). A negative correlation means that as one variable increases in size, the other decreases. A positive correlation means that as one variable increases so does the other. When r=0.00 there is no relationship between the two variables. This test has the same three assumptions as other parametric analyses, but it also has the additional assumption that the relationship between the two variables is linear. A regression analysis also gives a coefficient of variation (R^2). The coefficient of variation has a range of values from 0%-100% and explains how much of the variation in the dependent variable is because of the independent variable.

Nonparametric statistics

Most nonparametric statistics are simple to use, do not require large data sets, and have few underlying assumptions. They are not as powerful as parametric statistics (i.e. they are not very good at detecting small differences between groups), Non-parametric tests all assume independence of observations. In general, these tests should be chosen over parametric alternatives when sample sizes are small (less than 10-20 replicates). We will use three non-parametric tests in this course.

Wilcoxin's Rank Sums Test and the Mann-Whitney U Test

These analyses are used to test for differences between two treatment groups and are analogous to a t-test.

Kruskall-Wallis Test or adonis

These tests for differences between more than two different treatment groups. They're basically nonparametric ANOVAs.

Spearman's Correlation

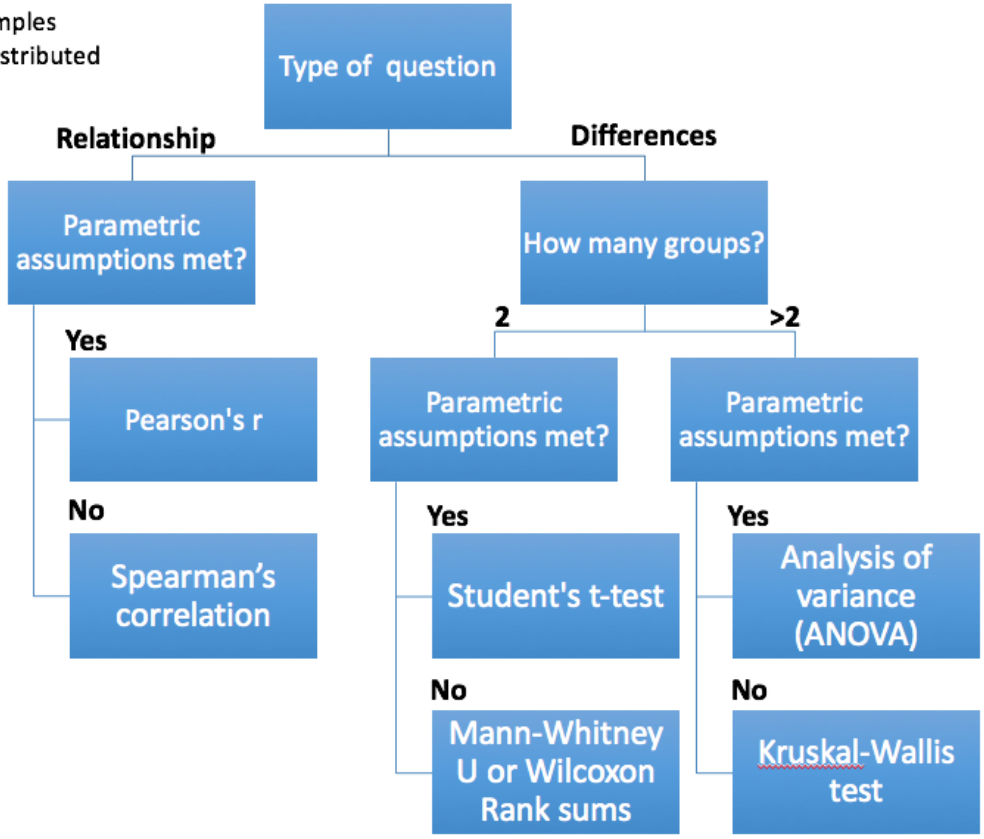
This analysis is a non-parametric regression analysis.

Choosing a Test

Below is a flowchart we will use to help us pick which test to use (Gerwien, 2016).

Parametric Assumptions

1. Independent samples
2. Data normally distributed
3. Equal variance



References

Everitt BS and Hothorn T. 2010. A handbook of statistical analyses using R. CRC Press.

Gerwien R. 2016. A painless guide to statistics. http://abacus.bates.edu/~ganderso/biology/resources/statistics.hmtl

Lender N, et al. 2014. Alimentary Pharmacology and Therapeutics. 40:24-31.

Rutten RBMM, et al. 2015. PLoS ONE. 10: e0137681.

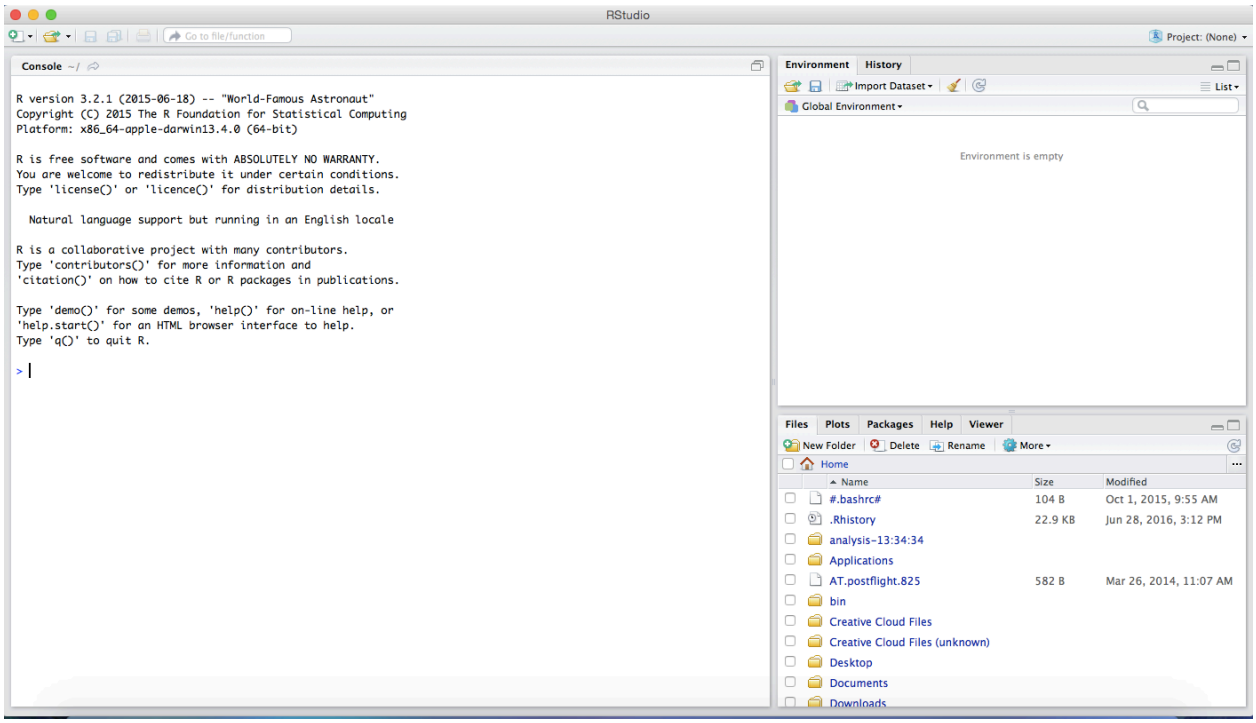
Streit M and Gehlenborg N. 2014. Nature Methods.11:117.

Using R Studio

This tutorial will help us learn how to use RStudio.

RStudio interface

The first time we open RStudio we are greeted by three panels. The left half of the screen is the console. The upper right corner is the variable inspector, and the lower right corner can show you different things depending on which tab is selected. The default for this panel is a file viewer. Let's go through each panel more in depth.



This is what RStudio looks like

Console

The console is like our terminal. Here we can type commands and R will perform them. The types of commands we use in the R console are, for the most part, specific to the R coding language. This language is different than the how you would write commands for your terminal.

Environment/Variable inspector

As we load and manipulate data, we can store the data as a 'variable'. The environment window shows us which variables we have created, and we can actually look to see what they are. Notice this panel also has a history tab where you can see all of the recent commands you have performed.

File System

This is just like the 'finder' on a Mac or the 'windows explorer' on a PC. This lists our files and folders. Notice this panel has other tabs as well. The plots tab will show us what our plots look like as we create them, the packages tab shows us all of the packages (sets of functions) you can import when running your analysis. The viewer tab is for more advanced interactive graphics and won't be used in this course.

Dropdown menus

These are the options listed across the top of RStudio (just like most other programs on our computers). These menus include many options you might need. For example, File, Save will save our work.

Using RStudio

To begin using Rstudio all we have to do is type a command into the console. For example, let's make a variable called `test_variable`. And in this variable we will store some data. The data will be a set of words ("bird", "dog", "cat"). We also call words strings, because the string of characters doesn't necessarily have to be an actual word. For example, "bird" is a string and so is "abcd".

```
test_variable <- c("bird", "dog", "cat")
```

Notice a couple of things:

1. the arrow `<-` is what assigns the data to a variable. You can read the whole command like this: "test_variable gets a vector containing the strings bird, dog, cat"
2. When we want to group things into one variable we can use `c()`, which is a function that combines values into a vector or list.
3. When we want to store strings we have to specify it's a string using quotes `""`. We can use either double `""` or single `''` quotes ("bird" or 'bird'). If we tried to use `c(bird, dog, cat)` instead of `c("bird", "dog", "cat")` R would read and try to interpret bird, dog and cat to be variables, not strings.

Once you have created this variable you will be able to see it in your environment panel on the right side of RStudio. It tells us that in `test_variable` contains characters (`chr`), there are three character values stored `[1:3]`, and they are `"bird", "dog" and "cat"`.

Using Existing R Code

In addition to typing things directly into the console, we can run R code from existing files. To do this we need to open a file that contains R code. Let's open the Intro-R.r file in R studio. To do that you can go to File, Open File, and find the 'Intro-R.r' file that you can download from Moodle. This file will open in a new panel above the console.

To run the code from this file we have two options.

1. Copy and paste the code into the console and press enter
2. Run the code directly from the file. To do this we can place our cursor on the line of the file we want to run. So, place your cursor on the `test_variable2 <- c("bacteria", "fungi")` line. For Mac users, you can run this command by holding down your 'command' key and pressing enter. For PC users, you can run this command by holding down your 'control' key and pressing enter. Another option is to place our cursor on the line we want to run and then pressing the `Run` button on the upper right side of the panel. We can run many lines of code, one line at a time by highlighting all the code we want to run in the file, and using the key combinations mentioned above.

Saving R Code

When we do an analysis in R, we definitely want to save our code so we can use it in the future. If we are working from an existing file, we can open the file and add new code as we write it. If we are starting a new file, we can use File, New File, R script to create a new file. As we run commands, they get stored in the 'History' tab of our 'Environment/Variable Inspector' (upper right panel). If we run a command, and it successfully does what we want it to, we can click on the command in the 'History' tab, and then click 'To Source' to add it to our file.

As we write and save our code, we must remember to comment it. Our comments will be used by anyone reading the code to figure out what was done. Our comments should:

1. Be close to the code we are specifically commenting (not just at the top of the file)
2. Be clear and concise
3. Capture intent

Remember that the comment symbol is #, and it is line-specific.

Here is an example of commented code:

```
# This stores the sum of 2,4,6 and 8 as a variable 'sum_numbers'
sum_numbers <- sum(2,4,6,8)

# This stores the square root of sum_numbers as a
# variable 'sqrt_numbers'
sqrt_numbers <- sqrt(sum_numbers)
```

Your comments don't have to be every line, but should be easily interpreted. If there are tricky parameters in your functions this is a good way to remind yourself why you have to specify certain things. Also, when naming variables, make sure to use a descriptive name that reflects what the variable is storing.

To save our R code files, we can use File, Save or 'command' s (Mac), or 'control' s (PC).

Important Concepts

Objects

Objects are the pieces of data stored as variables in R. There are different types of objects. We already mentioned one type, 'character', which includes letters and strings. Other types of objects we will use in this class include 'logical', which are either True or False, 'integer', which are integers, and many others.

Variables

Variables are what we store our data as in R. We name each variable (in our first example, it was 'test_variable'), and there are different types of variables. Some types include the following:

Vectors These can be considered a group of data. There are different types of vectors, some that we will use in this course include: 'logical' (trues or falses), 'integer' (numbers), and 'character' (strings).

```
vector_1 <- c(1,2,5.3,6,-2,4) # numeric vector
vector_2 <- c("one","two","three") # character vector
vector_3 <- c(TRUE,TRUE,TRUE,FALSE,TRUE,FALSE) #logical vector
```

Lists These are kind of like vectors, but the objects stored in the list do not have to be the same type.

```
list_1 <- c("one",2,TRUE)
```

Functions These are variables that perform a task. For example `c()` is a function that combines objects into a vector. R comes with many functions, and we write them with their name followed by parentheses.

```
length() #this is a function that will tell us the length of something
```

NULL Variables that are NULL contain nothing, and are not of a specific type. If we create a NULL variable, it will be listed in our environment but it will have no attributes.

```
nothing <- NULL
```

Matrices A matrix is a table, where all the columns in the matrix must have the same mode (numeric, character, etc.) and the same length.

```
# generates 5 x 4 numeric matrix
test_matrix <- matrix(1:20, nrow=5,ncol=4)
```

Data Frames A data frame is more general than a matrix, in that different columns can have different modes (numeric, character, factor, etc.).

```r
# This will make a dataframe where the columns are the filled with
# the vectors 'd', 'e' and 'f'
d <- c(1,2,3,4)
e <- c("red", "white", "red", NA)
f <- c(TRUE,TRUE,TRUE,FALSE)
test_dataframe <- data.frame(d,e,f)
```

Factors We can tell R that a variable is nominal by making it a factor. The factor stores the nominal values as a vector of integers and an internal vector of character strings (the original values) mapped to these integers.

```r
# Let's say there is a group of people, 3 female and 2 male
# Let's make a vector that stores how many female (F) and
# male (M) people there are
gender <- c("F", "F", "M", "M", "F")

# stores gender as a factor where 1=female, 2=male
gender <- factor(gender)
# R now treats gender as a nominal variable
```

Operators

Data is manipulated in programs using operators and functions. R has many built-in operators, the most commonly used include:

Arithmetic operators

* Numerical calculations (preserving the order of operations)
 + Addition +
 + Subtraction/change sign -
 + Multiplication *
 + Division /

Relational operators

* Comparing values
 + Less than <
 + Less than or equal to <=
 + Greater than >
 + Greater than or equal to >=
 + Equal to ==
 + Not equal to !=

Assignment operators

* Assigning values to objects

+ Global (you will generally use this one) `<-`
+ Local (often used within functions) =

Logical operators

* Conjunctions for combining/excluding terms
+ AND &&
+ OR ||
+ NOT !

Colon operator

* Creating regular sequences (often of numbers)
+ `:` example: `3:7` produces the output `[1] 3 4 5 6 7`

Now use the rest of the code in the 'Intro-R.r' file to make and inspect different variable types and operators.

Loading Tables in R

If you remember we generated 3 types of tables with QIIME:
* OTU table (.biom and .txt versions - rarefied and low-depth filtered) * Alpha diversity table (.txt) * Beta diversity tables (.txt)

OTU Table

Format

The first two lines include a spacer line detailing how the file was once a .biom format, and the column headers. Note that these lines start with a '#', which usually represents a comment line (something the computer doesn't read), so we will have to pay attention to how R reads our OTU table.

Rows

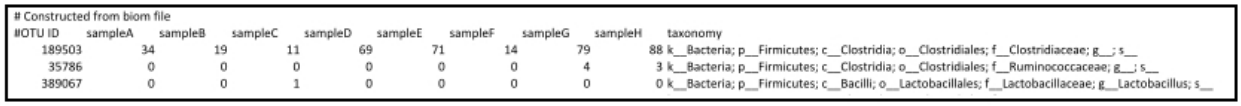
OTU ID, which is a unique ID for each set of sequences that are 97% identical.

Columns 1 through the second last

Each column represents a sample. The numbers in each row correspond to the number of reads that mapped to the specified OTU ID in the first column.

Last Column

The assigned taxonomic identity for each OTU (e.g. For k__Bacteria; p__Bacteroidetes; c__Bacteroidia; o__Bacteroidales; f__Prevotellaceae; g__Prevotella; s__copri). k = kingdom, the p = phylum, c = class, o = order, f = family, g = genus and s= species.
See example of the first 5 lines of an OTU table that is in the required format:



# Constructed from biom file									
#OTU ID	sampleA	sampleB	sampleC	sampleD	sampleE	sampleF	sampleG	sampleH	taxonomy
189503	34	19	11	69	71	14	79	88	k__Bacteria; p__Firmicutes; c__Clostridia; o__Clostridiales; f__Clostridiaceae; g__; s__
35786	0	0	0	0	0	0	4	3	k__Bacteria; p__Firmicutes; c__Clostridia; o__Clostridiales; f__Ruminococcaceae; g__; s__
389067	0	0	1	0	0	0	0	0	k__Bacteria; p__Firmicutes; c__Bacilli; o__Lactobacillales; f__Lactobacillaceae; g__Lactobacillus; s__

alt text

Loading OTU Table

First, use the function `read.table()` to read in your OTU table. These various arguments are all set specifically for the format of your OTU table in .txt format.

`comment =` is telling R what should be interpreted as a comment versus as a line of code. The default for this is the pound sign '#' but since we want the column header information we turn off the interpretation of comments using the option `comment = ''`

`header =` is telling R whether the first line of code should be assigned as row 1 or as the column names. We set this to `TRUE` or `T`.

`sep =` defines the field separator character which in this case is a tab, so `sep = '\t'`

`skip` = tells R how many rows to skip when reading in the table. The default for this is 0, but in this case, we want to ignore the first line '# Constructed from biom file' so we skip the first line.

`as.is` = controls the interpretation of character variables as a character string vs. as a factor. To avoid having thousands of levels associated with our taxonomy column, we specify `as.is=T`

`check.names` = determines whether the names of variable in the data frame are syntactically valid. Because our sample names in our data set start with numbers, which would cause problems in R, we have to set `check.names=F`

`row` = will tell R if we would like to set one of the columns to be the row names. In this case we would like to set the first column, which is the OTU IDs to be the row names. (row=1)

You will have to change the name of the OTU table to be the name of your table.

```
# Now we can read in the table - This the the rarefied one
otu <- read.table("otu_rare2000.txt",
                  comment="",
                  header=TRUE,
                  sep="\t",
                  skip=1,
                  as.is=TRUE,
                  check.names=F,
                  row=1)

# Read in the low depth removed OTU table
otu_low <- read.table("otu_rare2000.txt",
                  comment="",
                  header=TRUE,
                  sep="\t",
                  skip=1,
                  as.is=TRUE,
                  check.names=F,
                  row=1)
```

Remember, you can always find out more about a function by using the help() function or the ?. So, to find out about the `read.table()` function, you could do the following:

```
?read.table()
```

To find out information about our table we can use different functions. For example, we can find out the row names and column names using `rownames()` and `colnames()`, respectively. We can find the dimensions with `dim()`, and we can print the first couple of lines (default is 10) with `head()`. We can also click on our table in the Environment panel to view the whole table.

```
# View first 2 lines using head()
head(otu, n=2)

# View dimensions
dim(otu)

# Print row names (which are OTU IDs)
row.names(otu)

#Print column names (which are samples IDs the taxonomy header)
colnames(otu)
```

Alpha Diversity File

Prior to plotting in R, we need to generate an alpha diversity table in QIIME. This file will be the output of `alpha_diversity.py`, and will be a tab-delimited, plain text file. The format for the alpha diversity file is the following:

Format

Rows

The rows are the sample IDs.

Columns

Each column represents a diversity metric (e.g. `PD_whole_tree`, `simpson`, `shannon`, or `observed_species`). The numbers in each row correspond to alpha diversity estimate for the associated sample.

Loading Alpha Diversity

```
# Read in the alpha diversity table
alpha <- read.table("Alpha_Div.txt",
                    sep='\t',
                    header=TRUE,
                    as.is=TRUE,
                    check.names=FALSE,
                    row=1)
```

Notice:

* We set the header to be the first row (alpha diversity metrics)
* We set the rownames to be the first column (sample IDs)

Beta Diversity File

Prior to plotting in R, we need to generate an a distance matrix generated by with QIIME. This file will be the output of `beta_diversity.py`, and will be a tab-delimited, plain text file. The format for the beta diversity file is the following:

Format

Rows

The rows are the sample IDs.

Columns

Each column is also a sample ID and the distances from one sample to another are the values.

You should have one for each metric you used (Unweighted UniFrac, Weighted UniFrac, and Bray-Curtis).

```
# Load the beta diversity matrix, notice that we use read.table(),
# but then change from a dataframe to a matrix with as.matrix()
beta <- as.matrix(read.table("unweighted_unifrac_dm.txt",
                             sep = "\t",
                             header=T,
                             row = 1,
                             as.is = T,
                             check.names = F))
```

Notice:

* We set the header to be the first row (These are sample IDs)
* We set the rows names to be the first column (These are also sample IDs)

Metadata File

Your metadata file (also called a mapping file) is a data table containing information about the samples in your dataset. In order to assess how taxa correlate with variables of interest (e.g. country, body site, species, ecoregion, BMI, etc.), we need to have that information about our samples accessible. The metadata file for our data set is `HMP_5BS_metadata.txt`.

Format

Rows

The actual mapping file starts with '#SampleID' as the first header. This contains a the sample IDs, which are unique IDs for each sample in the dataset. To work in QIIME, this must have a '#' at the start. Remember that '#' usually represents a comment line (something the computer doesn't read), so we will have to pay attention to how R reads in this file.

Columns 1 - last column

Each column represents a description of the sample. It can be anything including details about the patient, person, animal or location the sample was taken from. This file should contain no spaces or empty columns/rows.

Loading Metadata File

We load the metadata table just like the OTU table, but notice that the `skip` parameter is left out, because the metadata table doesn't have the additional first line that the OTU table has.

```
metadata <- read.table('HMP_5BS_metadata.txt',
                       header=T,
                       sep='\t',
                       check.names=F,
                       comment='',
                       row=1)
```

Notice:
* We set the header to be the first row (These are sample IDs)
* We set the rows names to be the first column (These are also sample IDs)
* We told R to ignore the '#' in the first line

What are the dimensions of the metadata file? How would you find this out? We went over this in the 'Normalizing OTU Table tutorial'

What variables do we have available for this data set? They are the column headers. You can find this out using `colnames()`.

```
colnames(metadata)
```

```
## [1] "BarcodeSequence"      "LinkerPrimerSequence" "Sex"
## [4] "BodySite"             "SRS_SampleID"         "FASTA_FILE"
## [7] "Description"          "Age"
```

The mapping file is what we use in the majority of our QIIME commands so it contains information about the sequencing files (e.g. `BarcodeSequence`, `LinkerPrimerSequence`, `FASTA_FILE`, and `SRS_SampleID`), that are not necessary for our analysis. We need the sample IDs to match our variables to the microbial abundance information contained in our OTU table.

Each column in this file is a variable (also called a covariate), which can be defined as being continuous or categorical. Categorical variables are described as factors, the levels of which are the categories within it. You can view the number and identity of levels for a categorical variable by calling it, or using the `str()` function.

```
# View the 'Sex' column of the mapping file dataframe
metadata[,'Sex']
```

```
##   [1] female female female female male   female male   male   male
female
##  [11] female female male   male   female male   male   female female
female
##  [21] male   male   male   male   male   female female female male   male
##  [31] male   male   female male   male   female female male   male
female
##  [41] female male   female female female male   female female female male
##  [51] female female male   male   female female female female male   male
```

```
##  [61] male   male   female female male   female female female female male
##  [71] male   male   female male   male   female male   male   male
female
##  [81] female male   male   female female female male   male   male   male
##  [91] male   female female female female male   male   female male
female
## [101] female female male   female
## Levels: female male
```

Notice how we wrote the command to access the 'Sex' column. `[,]` is a way to specify rows and columns or a matrix or dataframe. Inside the square brackets, the first index specified is the row, and the second (after the comma) is the column. So what we wrote was, "display all the rows (left blank), in the 'Sex' column, or `metadata[,'Sex']`. We can also use the row number or column number `metadata[,4]`.

Because our mapping file is loaded as a dataframe, we can also do this using the "$".

```
# Notice that using "$" only works for dataframes and not matrices
metadata$Sex
```

```
##   [1] female female female female male   female male   male   male
female
##  [11] female female male   male   female male   male   female female
female
##  [21] male   male   male   male   male   female female female male   male
##  [31] male   male   female male   male   female female male   male
female
##  [41] female male   female female female male   female female female male
##  [51] female female male   male   female female female female male   male
##  [61] male   male   female female male   female female female female male
##  [71] male   male   female male   male   female male   male   male
female
##  [81] female male   male   female female female male   male   male   male
##  [91] male   female female female female male   male   female male
female
## [101] female female male   female
## Levels: female male
```

```
# The class function will also tell you whether your variable is
# a factor, numeric, character, etc.
class(metadata[,'Sex'])
```

```
## [1] "factor"
```

Formatting Your Data

In order to assess relationships between sample information in our OTU table, alpha diversity and beta diversity, we need to match the order of our data frames. For that we will use the `intersect()` function. Because we have potentially removed one or more samples from our OTU table during rarefaction, filtering or other manipulations, we can first define the subset of samples in all of our tables.

`intersect()` can retain all the sample IDs that are in the OTU table and also in the metadata file. We can then subset all of our tables to keep just those samples.

```
# First, define all the samples in the OTU table.
# Remember, when we load in the OTU table, samples are columns
# Remember, the last column in the OTU table is taxonomy, so ommit the last
column
samples1 <- colnames(otu)[1:(ncol(otu)-1)]

# Now let's see what the intersect with the  metadata row names are
IDs_Keep <- intersect(samples1, rownames(metadata))

# Now let's filter the metadata to keep only those samples
# We do this by telling R to make a new data frame that only has the rows we
want
metadata <- metadata[IDs_Keep,]

# Now let's filter the OTU table to keep just the intersecting samples
# We will store it as a new otu table (incase we need the old one)
# Remember, OTU table has columns as samples!
# This will also remove the taxonomy, because it's not a sample ID we want
otu2 <- otu[,IDs_Keep] #for rarefied
otu_low2 <- otu_low[, IDs_Keep] #for low depth removed

# To add the taxonomy back, we can use the taxonomy info from
# the orignal table
otu2$taxonomy <- otu$taxonomy
otu_low2$taxonomy <- otu_low$taxonomy

# Now let's filer the alpha diversity table to keep those samples too
# Alpha diversity has the samples as row names
alpha <- alpha[IDs_Keep, ]

# Now let's filer the beta diversity table to keep those samples too
# Beta diversity has the samples as row names AND column names
# We must filter both the rows and columns
```

```
beta <- beta[IDs_Keep,IDs_Keep]

#Let's check to make sure the samples match
as.character(rownames(metadata)) == colnames(otu2)[1:(ncol(otu2)-1)]
```

```
##   [1] TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE
TRUE
##  [15] TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE
TRUE
##  [29] TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE
TRUE
##  [43] TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE
TRUE
##  [57] TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE
TRUE
##  [71] TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE
TRUE
##  [85] TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE
TRUE
##  [99] TRUE TRUE TRUE TRUE
```

```
#Let's see how many samples are in the otu table (columns) and mapping

ncol(otu) #There should be one more here because there is also a taxonomy row
```

```
## [1] 103
```

```
nrow(metadata)
```

```
## [1] 102
```

Plotting in R

ggplot

To visualize our data in R we will use the package `ggplot2()`. This package allows us to make detailed and specific visualization needed to best show our results. Let's start with the packages we need to load. If these are not installed you can install them first with `install.packages()`.

```
library(ggplot2)
```

```
## Warning: package 'ggplot2' was built under R version 3.3.2
```

ggplot input

ggplot likes to have all the data for the plot in one table. Specifically, it like to be able to access the information needed by using columns. Let's use alpha diversity as an example. We will use our alpha diversity measurements as the input data for our examples.

First we have to combine our alpha diversity results with the metadata that tells us which body site each samples comes from. **Make sure you have your metadata and alpha diversity tables loaded and that the samples are subsetted and in the correct order for both tables.**

```
# We will make a copy of our metadata to work with
combined_alphadata <- metadata

# Because our sample order is the same, we can make a new column in the table
# This column will contain all the Shannon index measurements for the samples
combined_alphadata$shannon <- alpha$shannon
```

ggplot Format

ggplot creates plots in layers. First we make the base layer with `ggplot()` and then add on different types of plotting types and aesthetics.

Aesthetic Mapping

In ggplot, aesthetic means **something you can see**. For example:

* position (i.e., on the x and y axes)
* color (“outside” color)
* fill (“inside” color)
* shape (of points)
* linetype
* size

Geometic Objects (geom)

Geometric objects are the actual characters we put on a plot. For example:

* points (e.g. geom_point, for scatter plots)

* lines (e.g. geom_line, for line graphs)
* boxplot (e.g. geom_boxplot, for box plots)

Plotting

Before we plot our data, we need to think about what type it is. In the first example the independent variable will be "BodySite", which is discrete. Our dependent variable, alpha diversity (Shannon Index), falls in a range of values. We could use a bar chart, but a box plot will tell us more about the dataset.

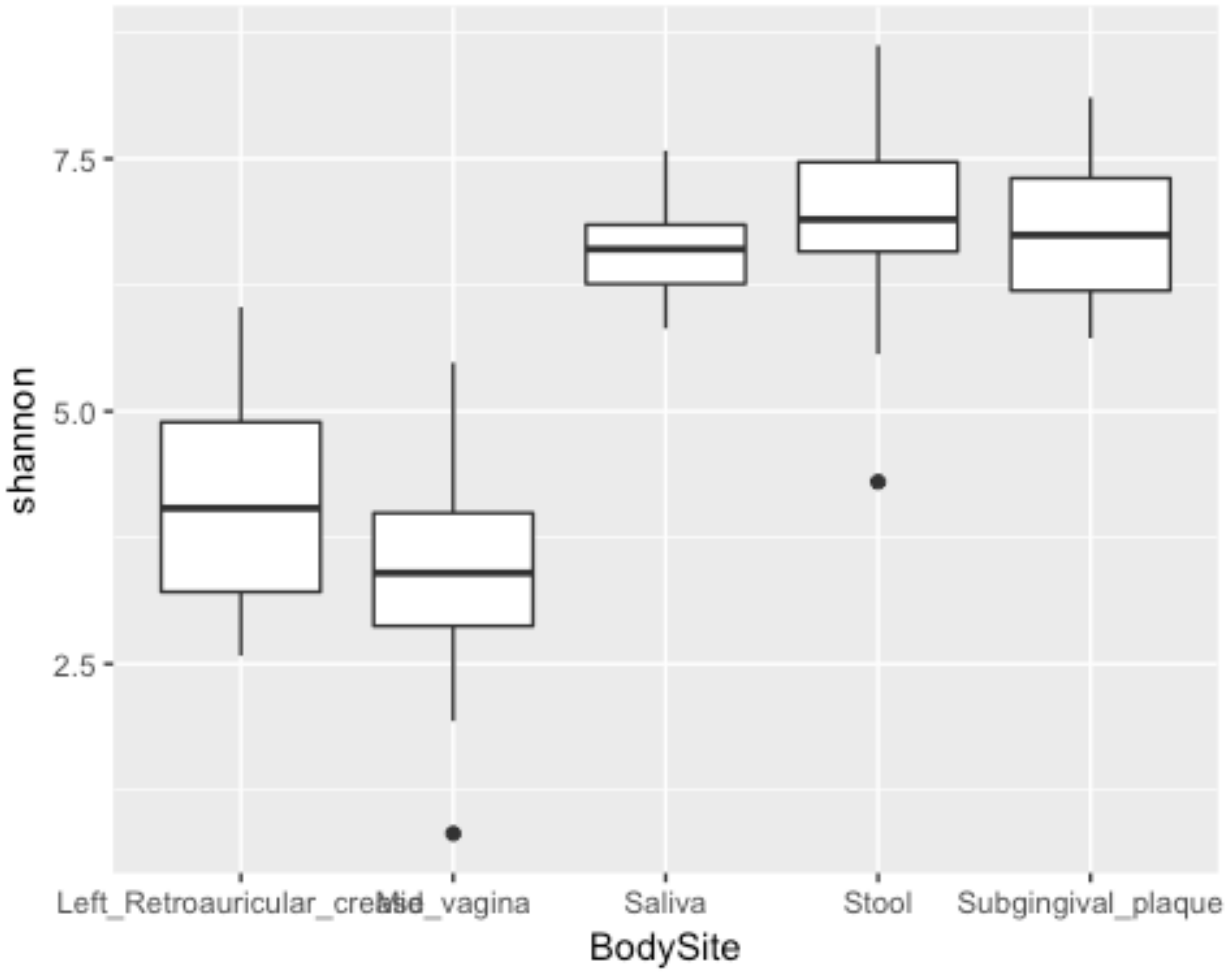
Box Plots

If we remember, Box plots consist of several features: **the box**, which extends from the first quartile (Q1) to the third quartile (Q3), respectively, with the median (Q2) depicted by a vertical line within the box; **whiskers**, which extending vertically from the box and indicate the range of variability outside of the upper and lower quartiles; and **outliers**, which are individual points outside of the box and whiskers.

It is important to note that box plots are good for **non-parametric** data. They display variation in samples of a statistical population without making any assumptions of the underlying statistical distribution. The spacing between the interquartile range of the box indicates the degree of dispersion (spread) and skewness in the data.

Plotting Example 1 - Discrete X-Variable

```
ggplot() +
  geom_boxplot(data=combined_alphadata, aes(x= BodySite, y= shannon))
```



Notice:

* We created the base layer with `ggplot()` * We add the next layer with+* We added a boxplot withgeom_boxplot()* We specified which table to use withdata=* We specified the aesthetics withinaes()`

* The dependent variable (y) is called by its column name

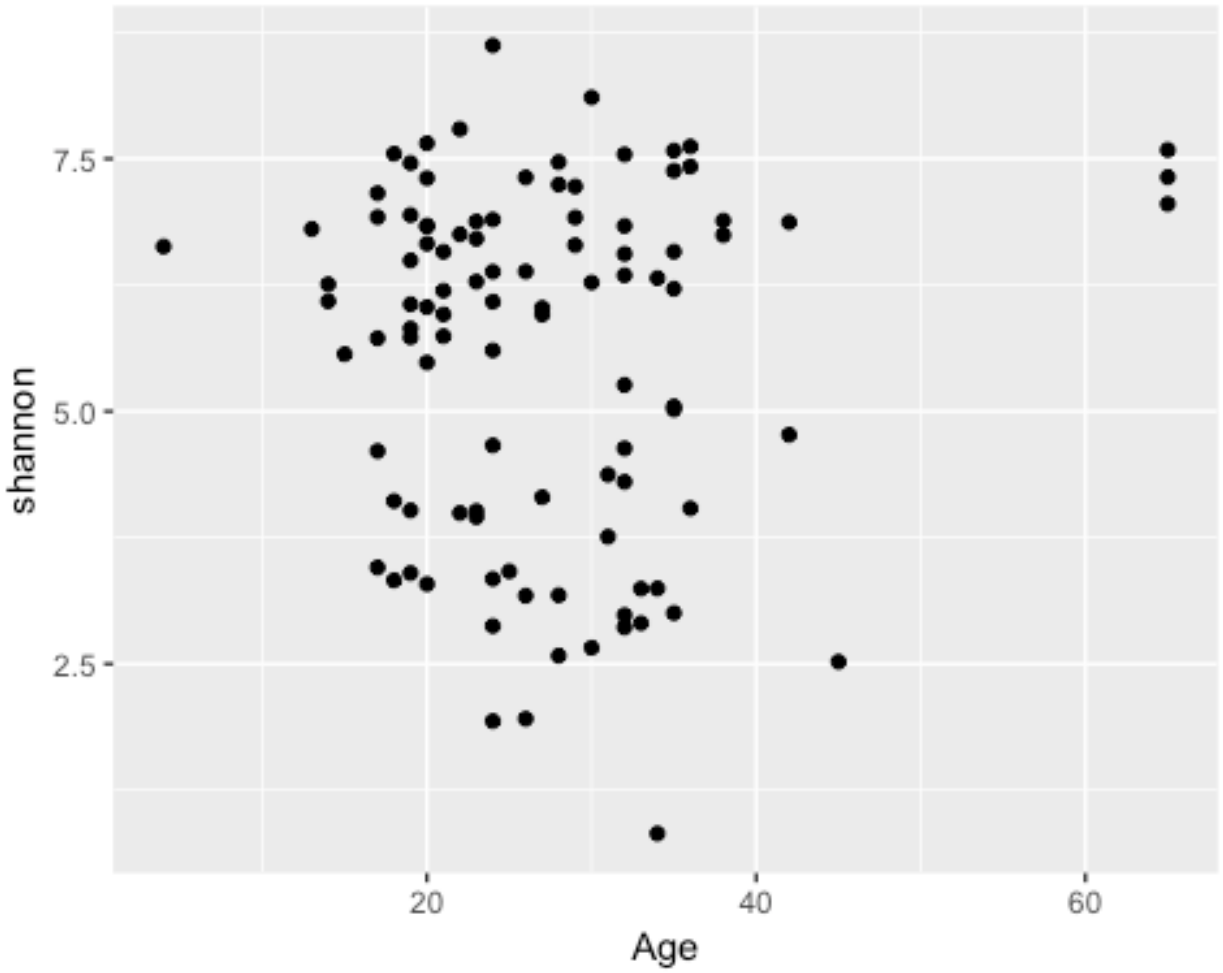
* The independent variable (x) is called by its column name

Scatter Plots

In the second example the independent variable will be "Age", which is continuous. Our dependent variable, alpha diversity (Shannon Index), falls in a range of values. Therefore we can use a scatter plot to look for trends.

Plotting Example 2 - Continuous X-Variable

```
ggplot() +
  geom_point(data=combined_alphadata, aes(x= Age, y= shannon))
```



Notice:

* We created the base layer with ggplot() * We add the next layer with+* We added a scatter plot withgeom_point()* We specified which table to use withdata=* We specified the aesthetics withinaes()`

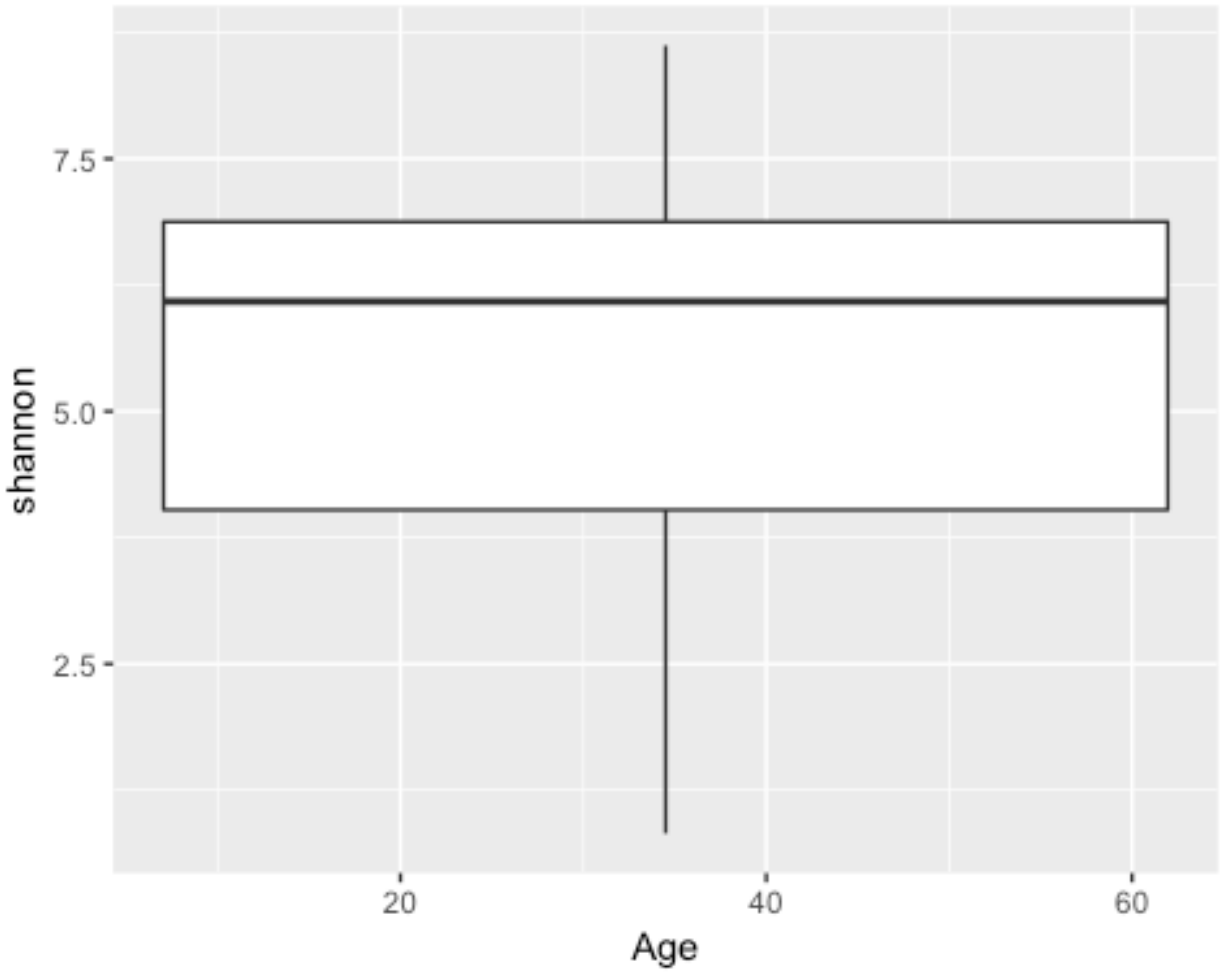
* The dependent variable (y) is called by its column name

* The independent variable (x) is called by its column name

Other Plotting Examples

Plotting Example 3 - Wrong Plot Types

```
ggplot() +
  geom_boxplot(data=combined_alphadata, aes(x= Age, y= shannon))

## Warning: Continuous x aesthetic -- did you forget aes(group=...)?
```



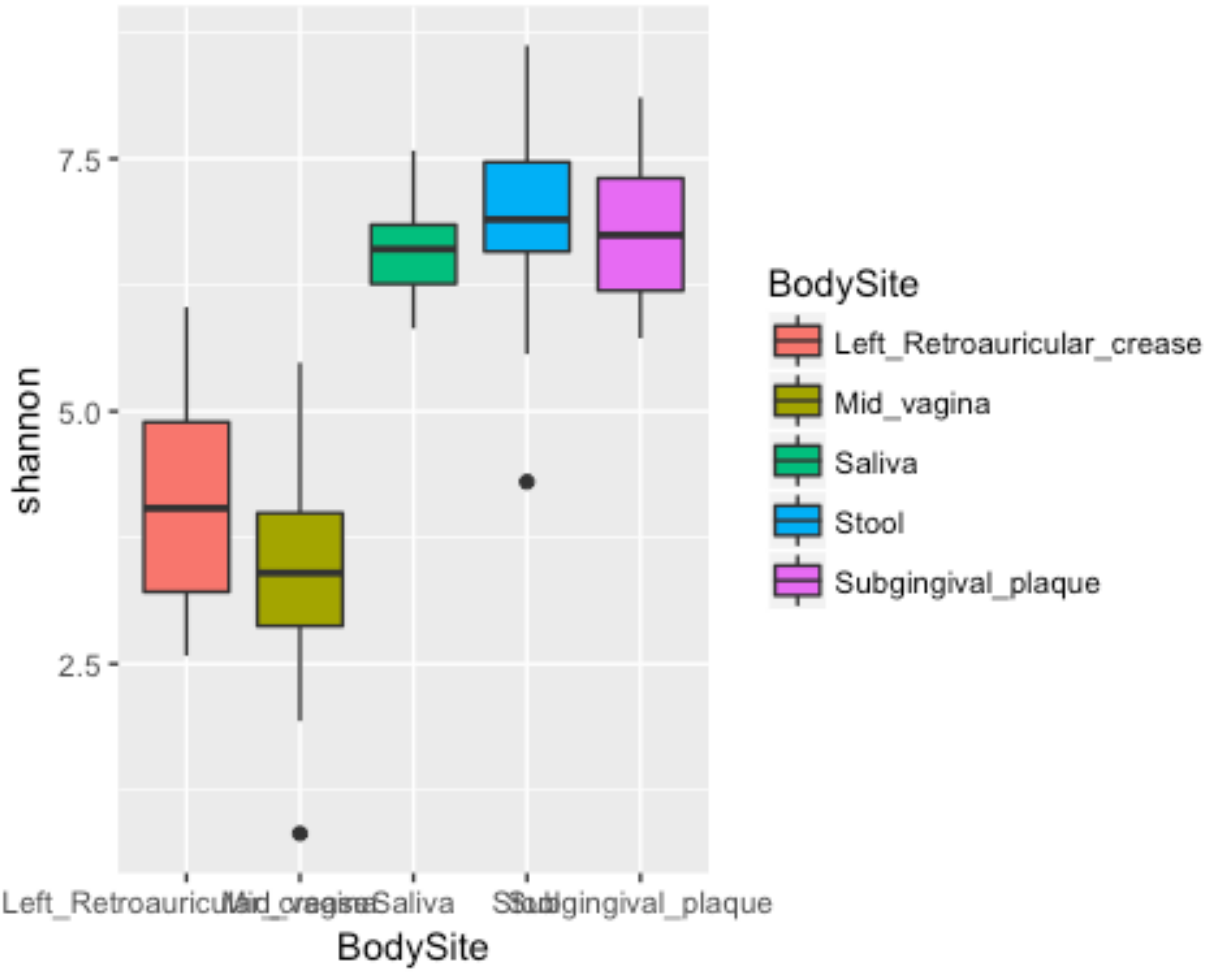
Notice:

* We tried to make a boxplot with continuous data. It clearly doesn't work well!

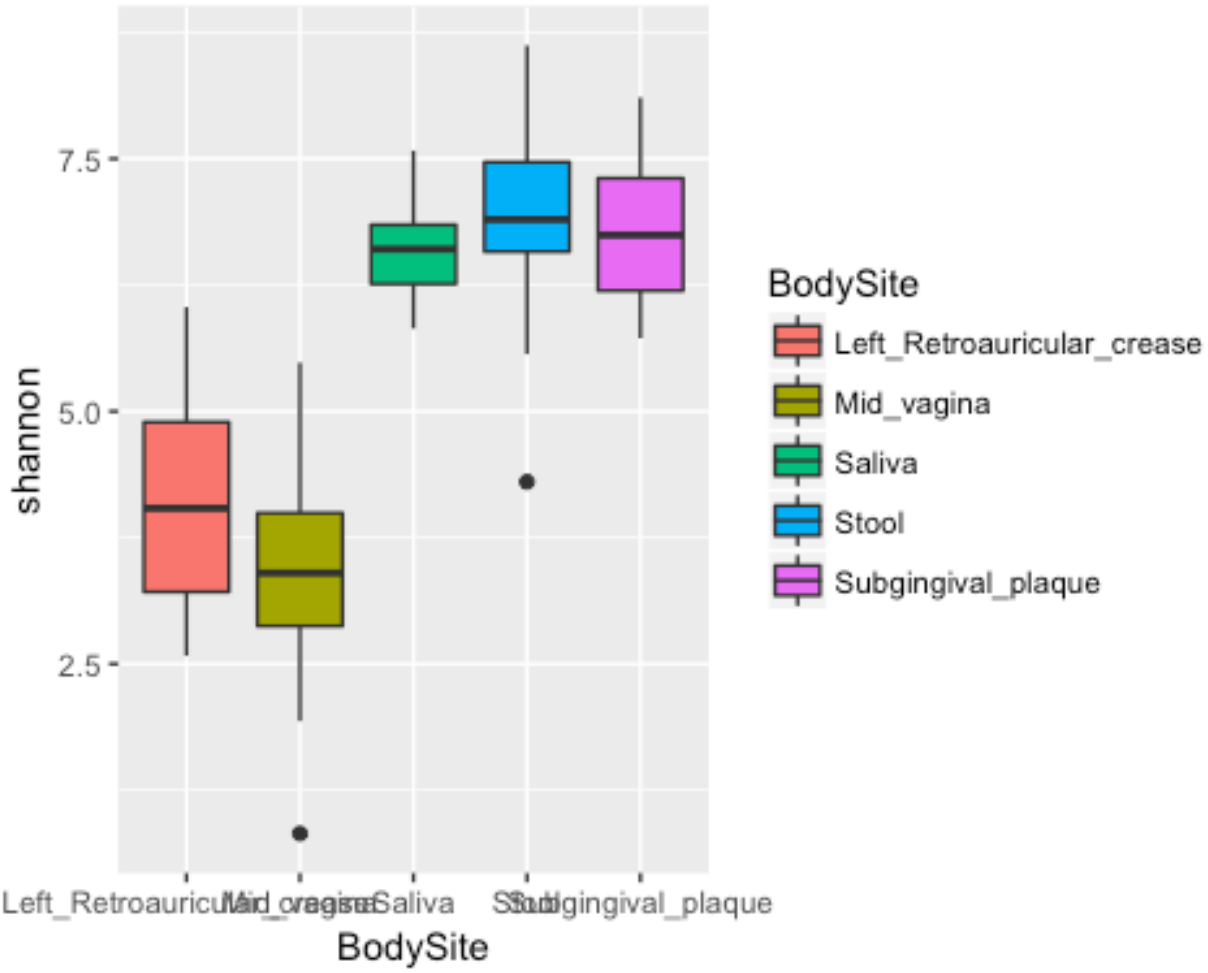
Plotting Example 4 - Adding in more aesthetics

```
# Specifying the color changes the outline color
ggplot() +
  geom_boxplot(data=combined_alphadata, aes(x= BodySite, y= shannon,
color=BodySite))
```

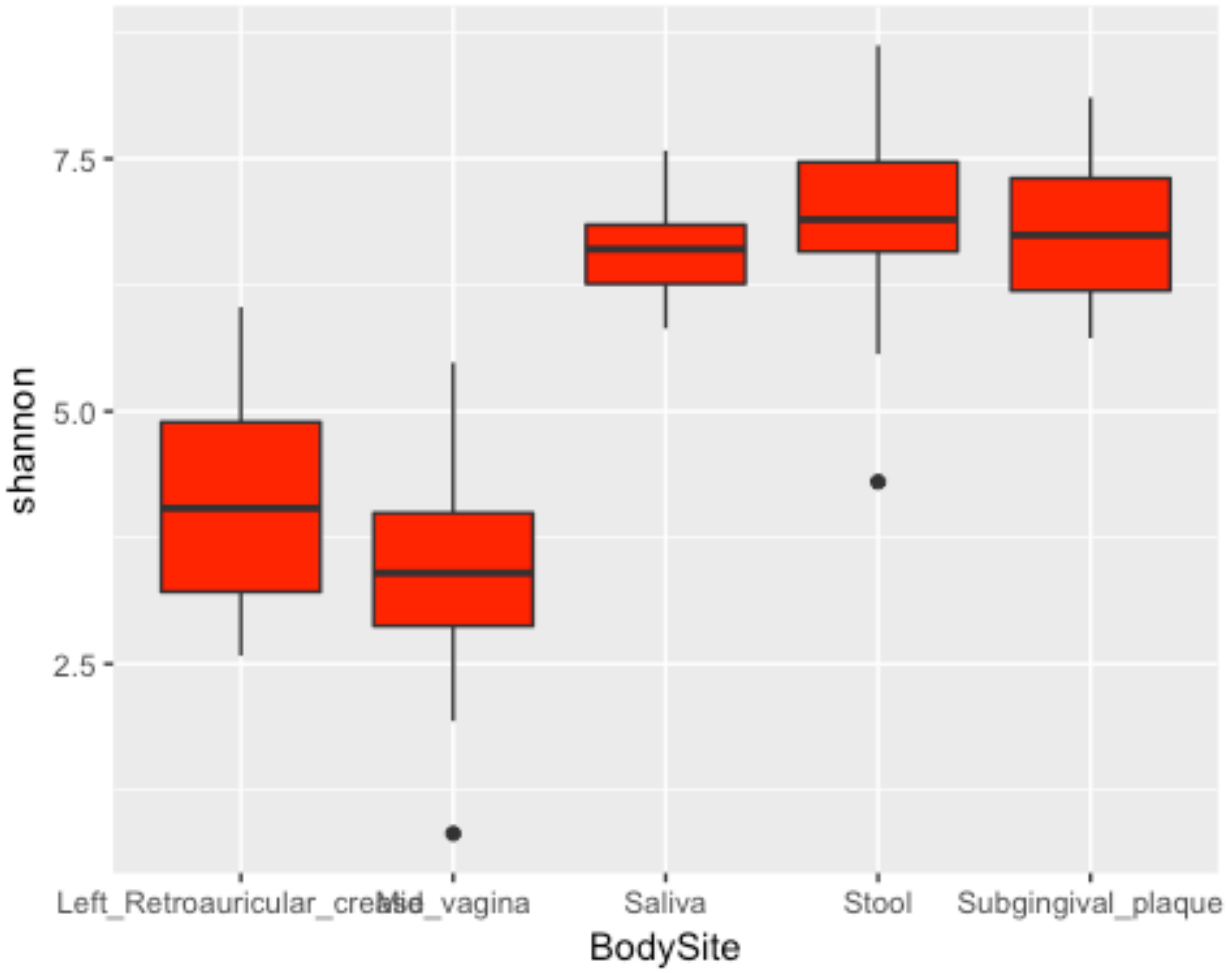

```
# Specifying the fill changes the interior color
ggplot() +
  geom_boxplot(data=combined_alphadata, aes(x= BodySite, y= shannon, fill=
BodySite))
```



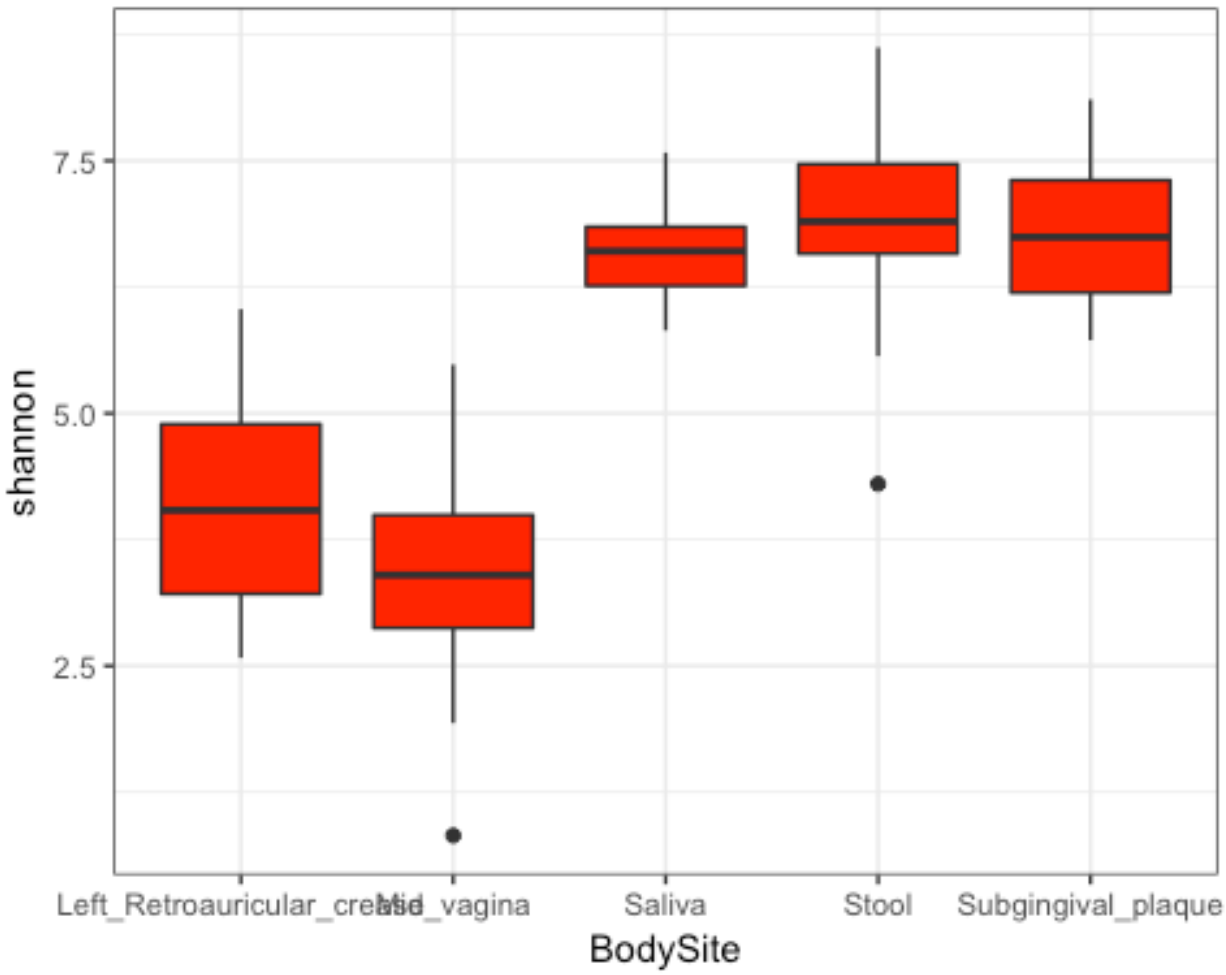

```
# Specifying the fill to a color changes the interior color for all
ggplot() +
  geom_boxplot(data=combined_alphadata, aes(x= BodySite, y= shannon, fill=
BodySite))
```



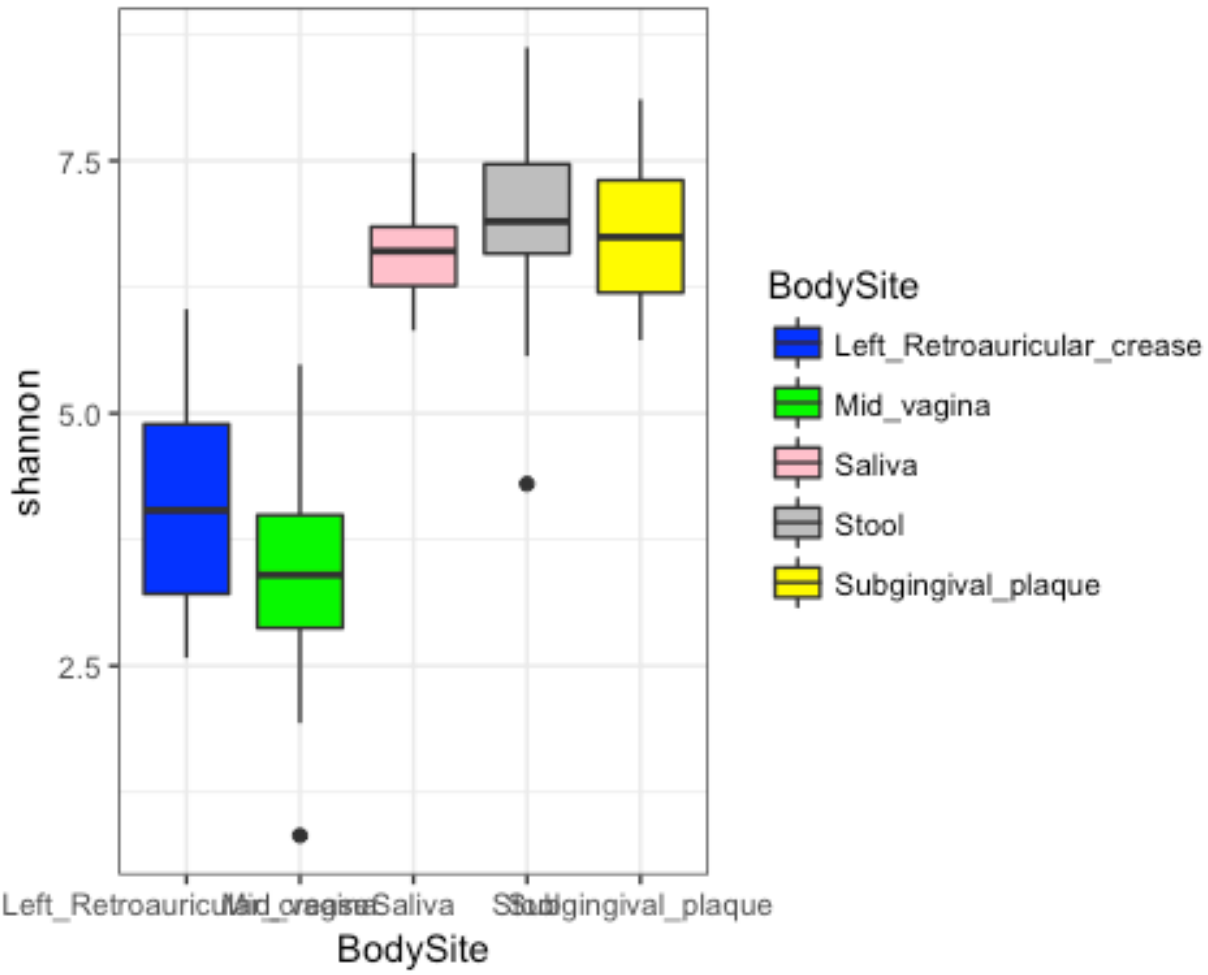

```
# Specifying the fill outside of aes() changes it for all x values
# You must pick an exact color if you'd like to do this
ggplot() +
  geom_boxplot(data=combined_alphadata, fill = "red", aes(x= BodySite, y=
shannon))
```




```
# Specifying the theme changes the background
ggplot() +
  geom_boxplot(data=combined_alphadata, fill="red", aes(x= BodySite, y=
shannon)) +
  theme_bw() #this is the black and white theme
```

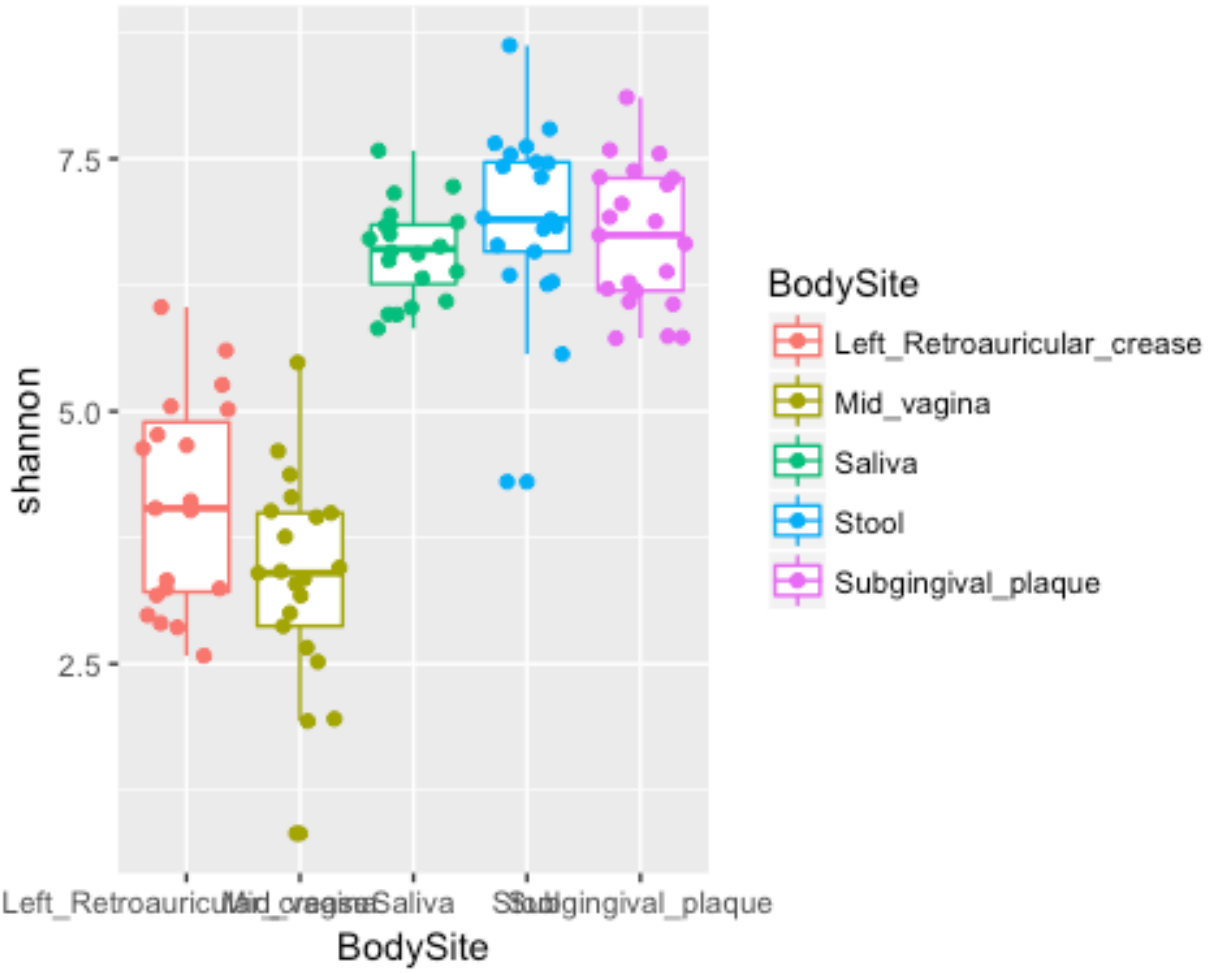



```
# We can pick any colors we want to fill by
ggplot() +
  geom_boxplot(data=combined_alphadata,  aes(x= BodySite, y= shannon,
fill=BodySite)) +
  theme_bw() +
  scale_fill_manual(values= c("blue", "green", "pink", "grey", "yellow"))
```

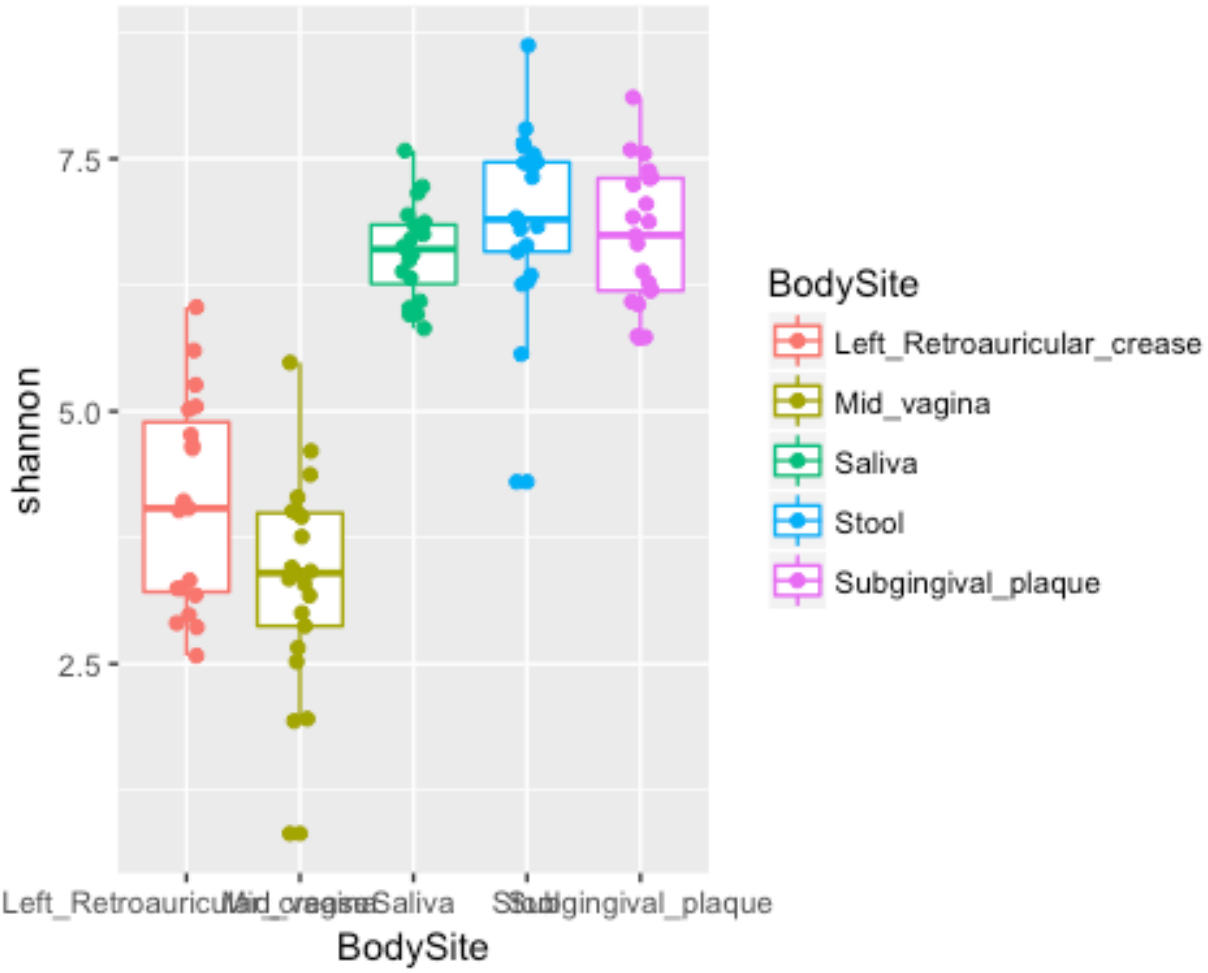


Plotting Example 5 - Adding Layers of Plots

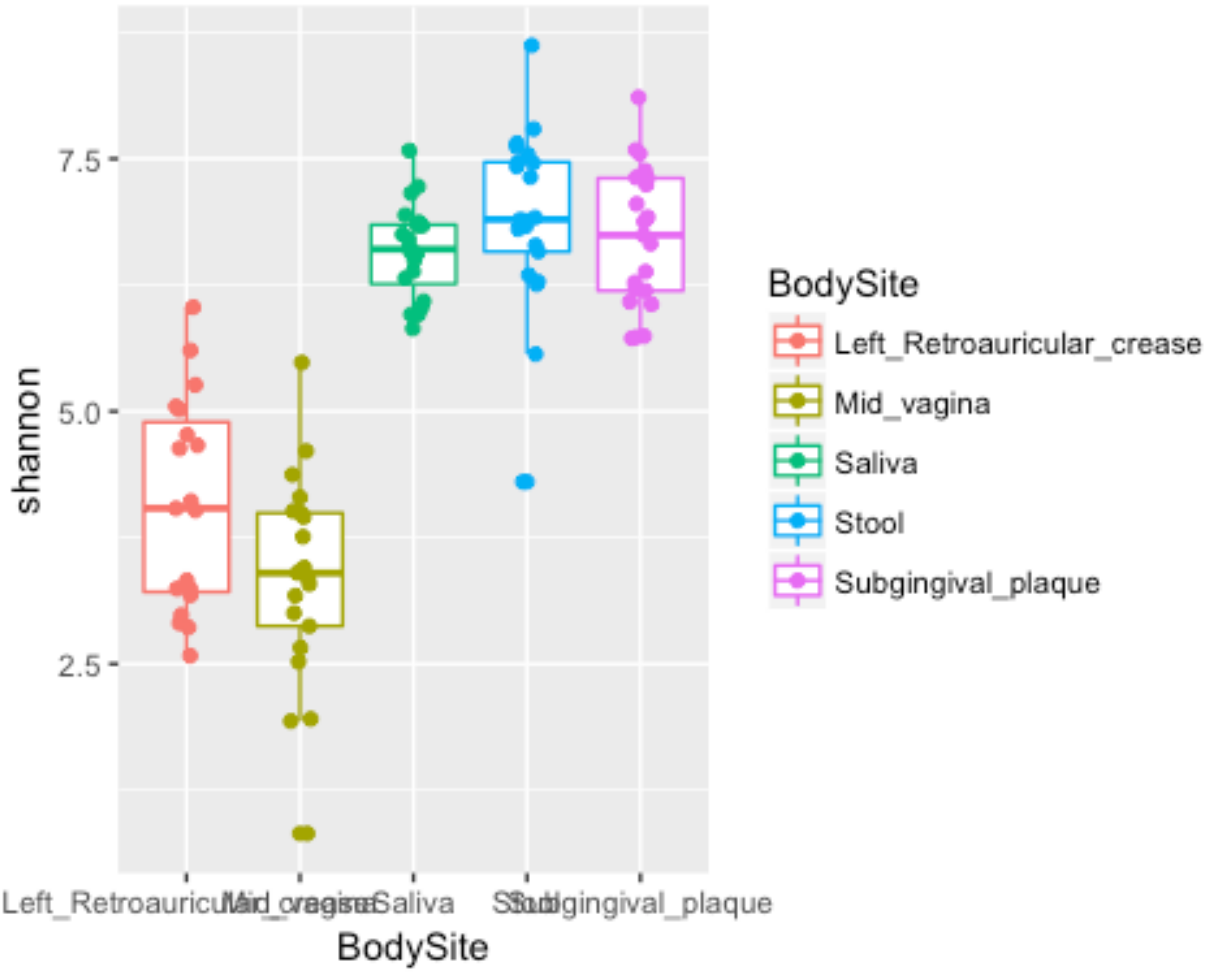
```
# Add a scatter on top of the boxplots
# To do this, we have to specify the data frame for each layer and the aes()
for each layer
ggplot() +
  geom_boxplot(data=combined_alphadata, aes(x= BodySite, y= shannon,
color=BodySite)) +
  geom_jitter(data=combined_alphadata, aes(x=BodySite, y=shannon,
color=BodySite))
```



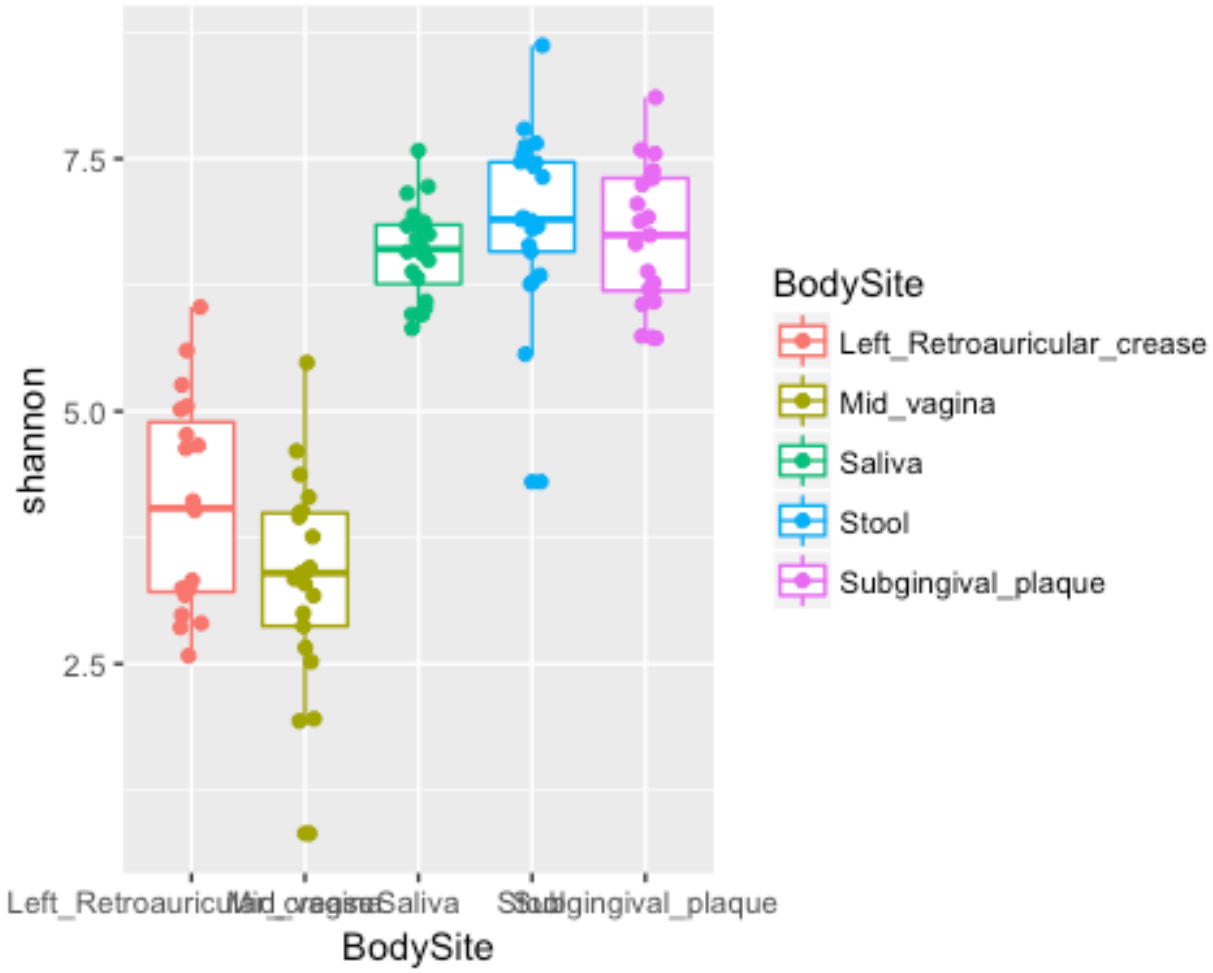

```
# We can decrease the jitter on the scatter plot with width=
ggplot() +
  geom_boxplot(data=combined_alphadata, aes(x= BodySite, y= shannon,
color=BodySite)) +
  geom_jitter(data=combined_alphadata, width= 0.1, aes(x=BodySite, y=shannon,
color=BodySite))
```



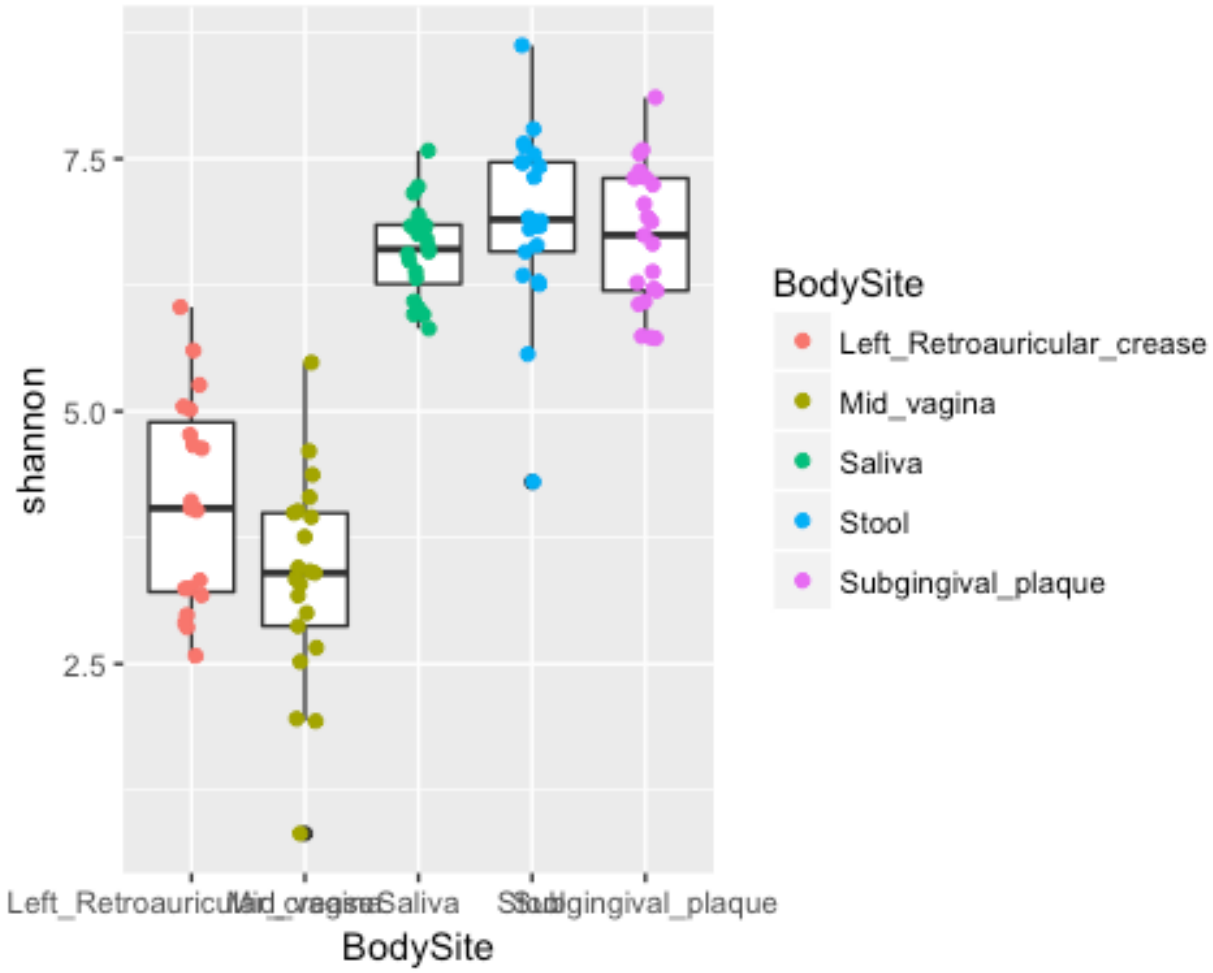

```
# If all the data for your entire plot will use the same data frame you can
specify that in the ggplot()
ggplot(data=combined_alphadata) +
  geom_boxplot(aes(x= BodySite, y= shannon, color=BodySite)) +
  geom_jitter(width= 0.1, aes(x=BodySite, y=shannon, color=BodySite))
```




```
# If all the data and aes() for your entire plot will be the same, you can
specify it in ggplot()
ggplot(data=combined_alphadata, aes(x=BodySite, y=shannon, color=BodySite)) +
  geom_boxplot() +
  geom_jitter(width= 0.1)
```




```
# You can make this specific to just one plot layer
# We are using color ONLY in the geom_jitter here
ggplot(data=combined_alphadata, aes(x=BodySite, y=shannon)) +
  geom_boxplot() +
  geom_jitter(width= 0.1, aes(color=BodySite))
```


shannon
7.5
5.0
2.5
Saliva
BodySite
BodySite
Left_Retroauricular_crease
Mid_vagina
Saliva
Stool
Subgingival_plaque

Alpha Diversity Differences in R

Input data

Alpha Diversity Table and Metadata

Your alpha diversity table and metadata table should be loaded. Remember to subset the tables so that the samples IDs are correct and in the same order. In the "Loading Tables in R" section we saved these tables as `alpha` and `metadata`

Now we need to pick which covariate we would like to use for the plot, and which alpha diversity metric we would like to visualize. We will use "Sex" and the "shannon" diversity metric.

Testing for Differences

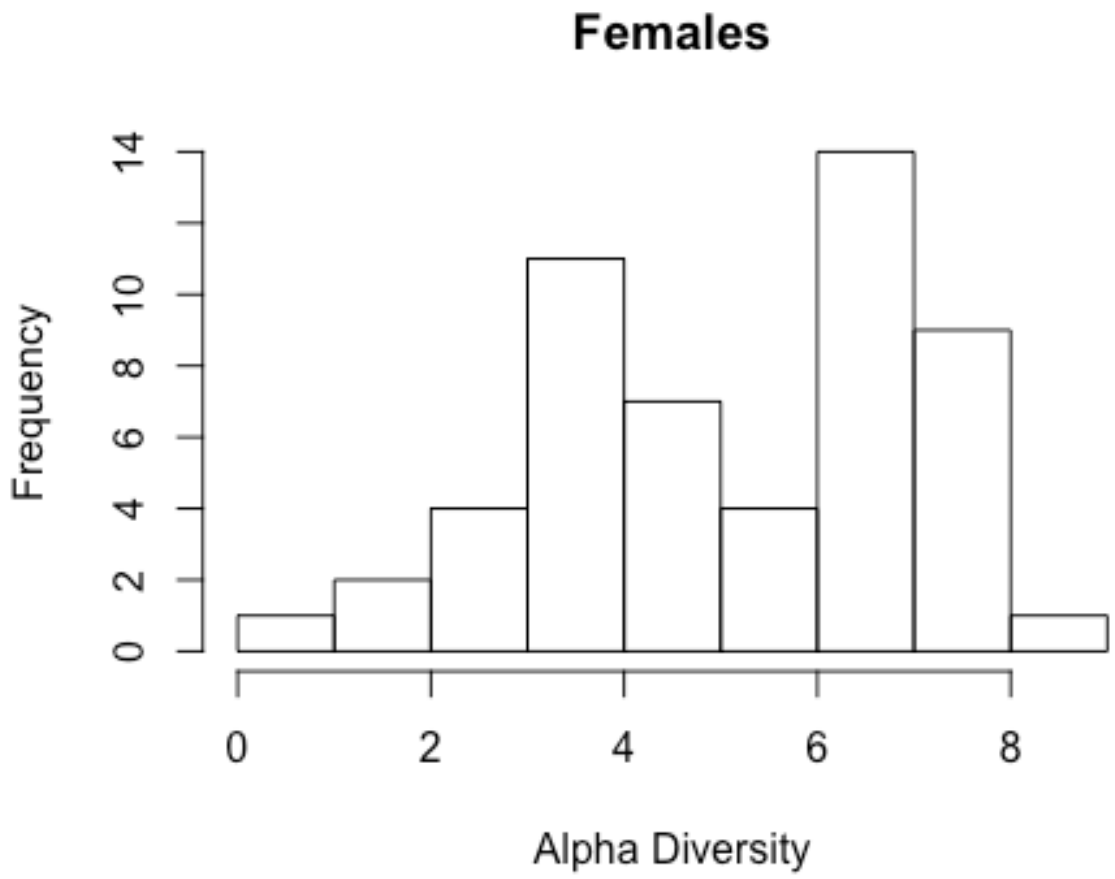
t-Tests

A *t*-test can be used to determine if two sets of data are significantly different from each other based on the population means. It assumes the data are normally distributed. Although it is typically assumed that data of a large enough sample size are normally distributed, this is not always the case.

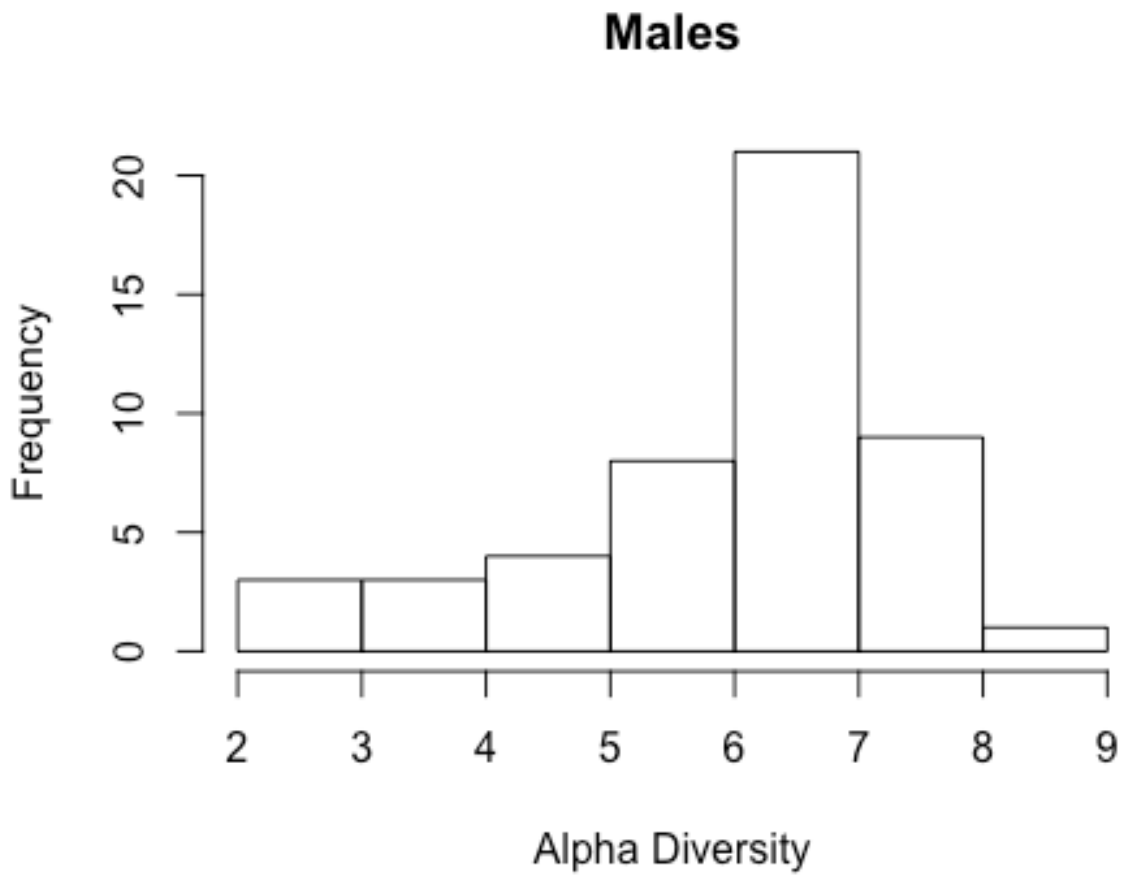
Testing for Normality

Let's see if our data are normally distributed using the hist() function. We will do this for each group in the covariate we are testing (in this case, 'sex').

```
# We will find the samples that are male in the metadata
males.ix <- metadata$Sex == 'male'
# And subset the alpha table to include only those, and store it as 'males'
males <- alpha[males.ix,]

# We will do the same for females
females.ix <- metadata$Sex == 'female'
females <- alpha[females.ix,]

# Now we can plot the histograms
hist(females$shannon, xlab="Alpha Diversity", main='Females')
```




```
hist(males$shannon, xlab="Alpha Diversity", main='Males')
```



Do those look normally distributed? For the most part they do, but it is sometimes hard to tell. The best way to determine if your data are normally distributed is to do a statistical test.

The Shapiro-Wilk Normality Test

This test can be run with the `shapiro.test()` function in R. It will generate an approximate p-value, which is adequate in assessing normality. In this case, **p-values less than 0.1 indicate the data are significantly different from normal distribution**. We will run this test for each group in our covariate of interest.

```
shapiro.test(females$shannon)
```

```
##
##  Shapiro-Wilk normality test
##
## data:  females$shannon
## W = 0.94434, p-value = 0.01548
```

```
shapiro.test(males$shannon)
```

```
##
##  Shapiro-Wilk normality test
##
## data:  males$shannon
## W = 0.89861, p-value = 0.0004951
```

Based on the results, should you run a *t*-test?

Mann-Whitney U Test

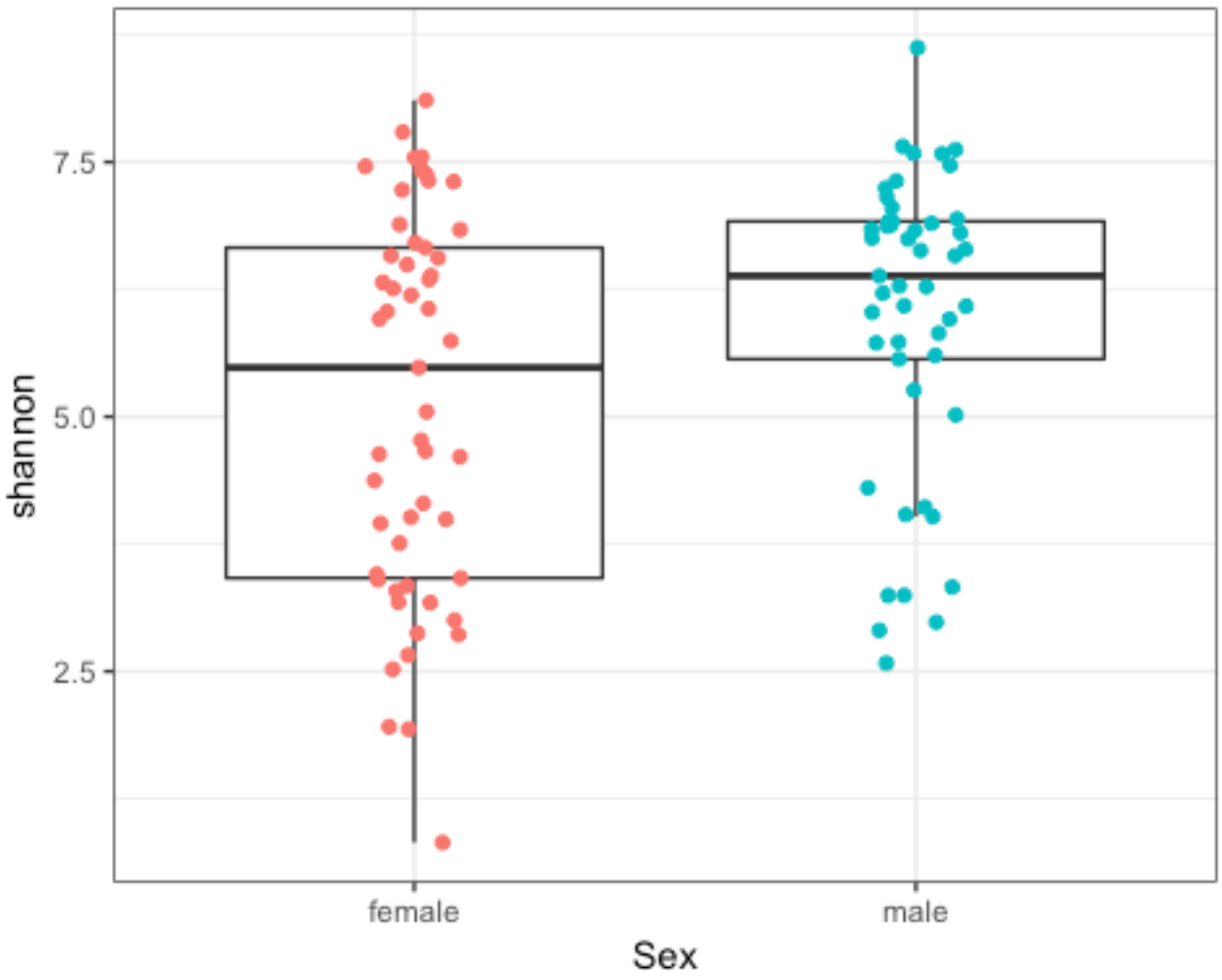
Based on these results, it is better to test for differences in our data using a statistical test that does **not** require normal distributions. The Mann-Whitney U test (aka Wilcoxen-rank_sum test) is smilar to the *t*-test. The null hypothesis of this test would be that that groups means differ only by chance. We can do a Mann-Whitney U test using the `wilcox.test()` function in R.

```
wilcox.test(females$shannon, males$shannon,  na.rm=TRUE)

##
##  Wilcoxon rank sum test with continuity correction
##
## data:  females$shannon and males$shannon
## W = 973, p-value = 0.0295
## alternative hypothesis: true location shift is not equal to 0

# This is telling are to use a wilcox test to comapre females
# and males for the diversity metric we set earlier. It is also
# telling R to remove any NAs (missing data)


# Note that you would run a t-test in the same way, only using
# the t.test() function
```

The p-value here, which is less than an alpha of 0.05. This means that we **reject** the null hypothesis that these two body sites are **not significantly different** for this metric of alpha diversity.

We can also see what the box plot looks like.

```
alpha2 <- alpha
alpha2$Sex <- metadata$Sex


ggplot(data=alpha2, aes(x=Sex, y= shannon)) +
  geom_boxplot(outlier.color = NA) + # removes outlier points becuase we add
in the jitter anyways
  geom_jitter(width= 0.1, aes(color=Sex)) +
  theme_bw() +
  guides(color=F) #because the x-axis is already labeled
```



In our example we only have two groups, 'male' and 'female'. If we used a different covariate, like 'BodySite' (which contains 5 groups) we would have 10 tests to do pairwise. Luckily, for loops can come to the rescue! This loop will perform the `wilcox.test()` on every unique combination of groups in the covariate.

```
# First let's set all the groups available for the variable we care about
# In this case we will use BodySite instead of what we set as cov1
# (because sex only has two values)
groups <- unique(metadata$BodySite)

# We create empty vectors to store the pair-wise pvalues and the
# groups tested (names)
pw.pvalues <- NULL
pw.names <- NULL

# We set two counters, 'i' starts at 1 and goes until one less than
# the number of groups. 'j' will start at 2, and go until the full
```

```r
# number of groups. This will end up comparing: 1 vs 2, 2 vs 3,
# 3 vs 4, and so on.
for(i in 1:(length(groups) - 1)){
            for(j in (i+1):length(groups)){
              #we use this to pick the groups assigned to 'i'
                ix.metric.i <- metadata$BodySite == groups[i]
                #and this for 'j'
                ix.metric.j <- metadata$BodySite == groups[j]
                #this stores the pvalue from the test
                pvalue <- wilcox.test(alpha[ix.metric.i,"shannon"],
                                      alpha[ix.metric.j,"shannon"])$p.value
                #appends the new p-value to the list
                pw.pvalues <- c(pw.pvalues, pvalue)
                #sets the names of the groups tested
                test.name <- paste(groups[i], "_vs_", groups[j],sep='')
                #appends the names of the groups tested to the list
                pw.names <- c(pw.names, test.name)
            }
}
names(pw.pvalues) <- pw.names

pw.pvalues
```

```
##         Mid_vagina_vs_Left_Retroauricular_crease
##                                     6.925356e-02
##                               Mid_vagina_vs_Saliva
##                                     7.431382e-12
##                  Mid_vagina_vs_Subgingival_plaque
##                                     3.715691e-12
##                                Mid_vagina_vs_Stool
##                                     2.600984e-11
##              Left_Retroauricular_crease_vs_Saliva
##                                     3.482133e-10
## Left_Retroauricular_crease_vs_Subgingival_plaque
##                                     1.066403e-10
##               Left_Retroauricular_crease_vs_Stool
##                                     2.117572e-09
##                     Saliva_vs_Subgingival_plaque
##                                     5.100514e-01
##                                  Saliva_vs_Stool
##                                     4.815795e-02
##                       Subgingival_plaque_vs_Stool
##                                     2.298969e-01
```

False Discovery Rate Correction

When we use the 'sex' covariate, we only have one test to perform. If we are comparing more than two groups and we are running multiple tests we have to correct of the number of comparisons we are making. We do this with the `p.adjust()` function. This will correct for type I errors, which are rejections of a true null hypothesis (also known as a false positive).

```
# We will correct using 'fdr', which is the false discovery rate
fdr.pvalues <- p.adjust(pw.pvalues,'fdr')
fdr.pvalues
```

```
##         Mid_vagina_vs_Left_Retroauricular_crease
##                                     8.656695e-02
##                               Mid_vagina_vs_Saliva
##                                     3.715691e-11
##                 Mid_vagina_vs_Subgingival_plaque
##                                     3.715691e-11
##                                Mid_vagina_vs_Stool
##                                     8.669946e-11
##             Left_Retroauricular_crease_vs_Saliva
##                                     6.964267e-10
## Left_Retroauricular_crease_vs_Subgingival_plaque
##                                     2.666008e-10
##              Left_Retroauricular_crease_vs_Stool
##                                     3.529287e-09
##                      Saliva_vs_Subgingival_plaque
##                                     5.100514e-01
##                                  Saliva_vs_Stool
##                                     6.879708e-02
##                       Subgingival_plaque_vs_Stool
##                                     2.554410e-01
```

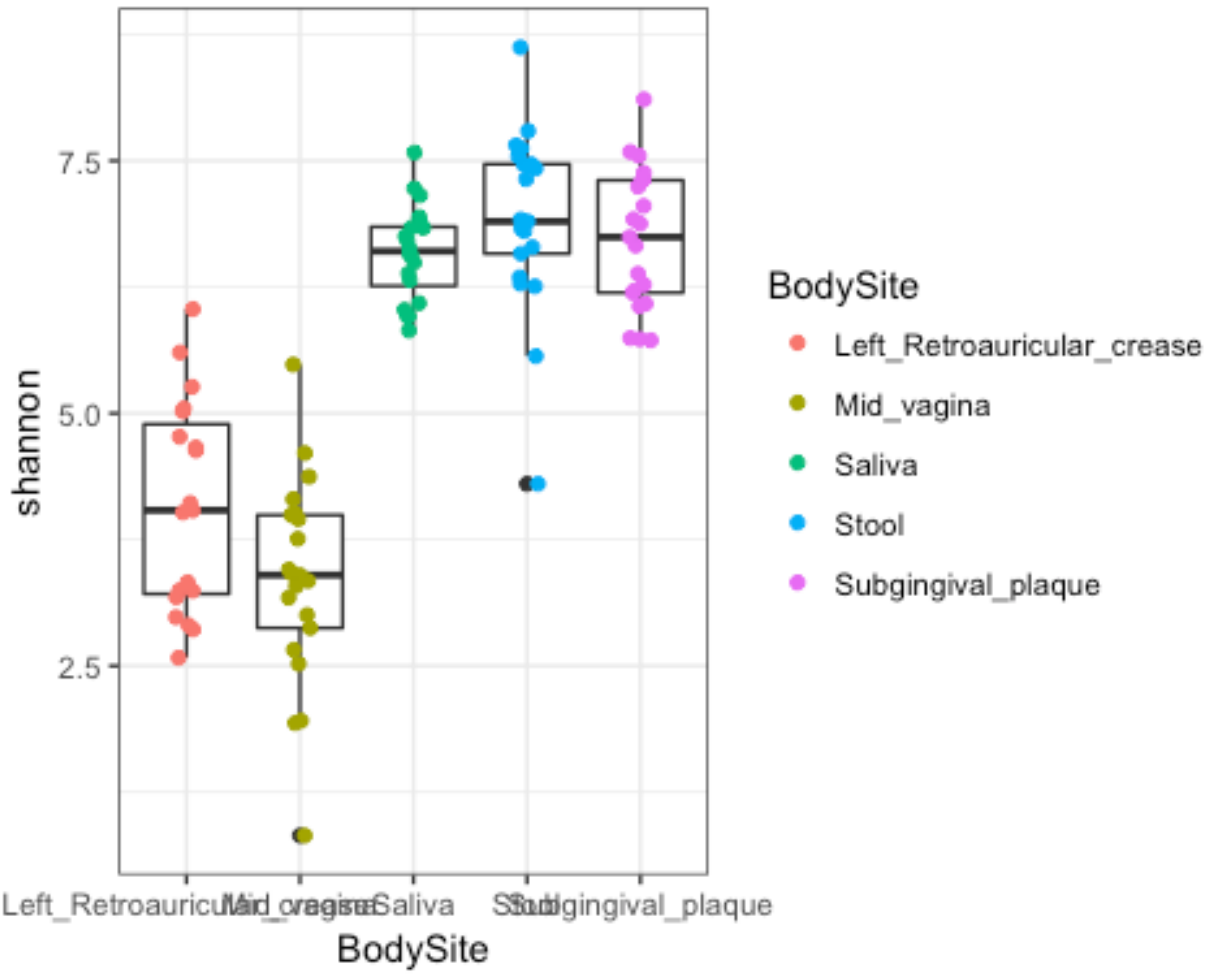
Now we can view the relative p-values for each pairwise comparison, and we can save this table as a file.

```
# sink() will write whatever is listed below it to a file.
# You close that file by listing sink() again.
sink("alpha_stats.txt")

cat("\nNumber of samples in each group:\n")
print(table(metadata$BodySite))
#This prints a table of the number of samples at each body site

cat("\nMean Alpha Diversity:\n")
print(tapply(alpha$shannon, metadata$BodySite, mean))
```

```r
# This will get the mean of alpha diversity at each body site
# by using tapply() to apply the mean function across the alpha
# table (subsetted into body site groups)

cat("\nMedian Alpha Diversity:\n")
print(tapply(alpha$shannon, metadata$BodySite, median))
# This will get the median of alpha diversity at each body site

cat("\nStandard Deviation:\n")
print(tapply(alpha$shannon, metadata$BodySite, sd))
# This will get the standard deviations of alpha diversity at
# each body site

cat("\nPairwise Mann-Whitney-Wilcoxon Tests were performed.\n")
cat("Pairwise p-values are:\n")
print(pw.pvalues)

cat("\nFDR-corrected pairwise p-values are:\n")
print(p.adjust(pw.pvalues,'fdr'))

sink()
```

We can also see what the box plot looks like.

```r
alpha2 <- alpha
alpha2$BodySite <- metadata$BodySite

ggplot(data=alpha2, aes(x=BodySite, y= shannon)) +
  geom_boxplot() +
  geom_jitter(width= 0.1, aes(color=BodySite)) +
  theme_bw()
```



We can also print this plot to a pdf with the `pdf()` function followed by `dev.off()` to close the pdf.

```
plot_output <- ggplot(data=alpha2, aes(x=BodySite, y= shannon)) +
  geom_boxplot() +
  geom_jitter(width= 0.1, aes(color=BodySite)) +
  theme_bw() +
  scale_x_discrete(labels=c("ear fold", "vagina", "saliva", "stool",
"plaque")) +
  guides(color=F) #because they are labeled at the x- axis

pdf("Alpha_Diversity.pdf", height=4, width=6)
plot(plot_output)
dev.off()
```

```
## quartz_off_screen
##                 2
```

Taxa Summary Plots in R

Visualizing which taxa are in your samples can be an effective way to see patterns in the data. Here we will learn how to make taxa summary plots based on your input OTU table, a covariate of interest, and other specified parameters.

Input data

OTU table and Metadata

Your OTU table should be loaded and we can use the rarefied version. Your metadata file should also be loaded. Remember to subset the tables so that the samples IDs are correct and in the same order. In the "Loading Tables in R" section we saved these tables as `otu2` and `metadata`

Manage Taxonomy

We want to set a taxa level by number: 1 = kingdom
2 = phylum
3 = class
4 = order
5 = family
6 = genus
7 = species

Let's work with phylum (level 2). The taxa are listed with a letter representing the level followed by two underscores, and a semicolon separating each level (k__kingdom; p__phylum; ...). We will do some string parsing to replace the full taxonomy label with the appropriate level.

```
# In this example we are using 2 (or phylum).
# This can be for any level you want.
level = 2

# First we make an empty table (array) for our new names
# The array will have the number of rows equal to the number of OTUs in the
table
# and one column for each taxonomy level
names_split <- array(dim=c(length(otu2$taxonomy), level))

# We will store our taxonomy as a list of names
otu_names <- as.character(otu2$taxonomy)

# Then we run through each name and split based on the level we are
# interested in. We make a for loop to split every name stored in
# otu_names. strsplit() splits the string (otu_names[i]) at ";".
# This retains all the levels as separate strings. head() takes the
```

```
# first items (the total will be the number you specified with level)
# from the string split output and stores it in the names_split
# array at the specied row.
for (i in 1:length(otu_names)){
    names_split[i,] <- head(strsplit(otu_names[i], "; ", fixed=T)[[1]],
n=level)
}

# Now we will collapse the strings together into one string
otu_names <- apply(names_split, 1, function(x) paste(x[1:level], sep = "",
collapse = ";"))

# Replace the old taxonomy with the truncated version
otu2$taxonomy <- otu_names
```

Now we want to consolidate our OTU table by the taxa levels we've set, just like we learned in the loading and manipulating tutorial. We will use the `aggregate()` function.

```
# Get the number of samples (the last column is taxonomy)
sample_no <- ncol(otu2)-1

# Collapse the otu table and save it as a new table
otu3 <- aggregate(otu2[,1:sample_no], by=list(otu2$taxonomy), FUN=sum)

# Name the first column taxonomy because R stores the column
# we told it to aggregate by as the first column
names(otu3)[1] <- "taxonomy"

# We can see that the consolidating worked by checking how many rows we
# now have - that's how many phyla there are (level=2)
nrow(otu3)
```

```
## [1] 17
```

Let's replace the rownames with the taxonomy, and get rid of the taxonomy column.

```
# Set rownames as taxonomy
rownames(otu3) <- otu3$taxonomy

# Keep all columns in the otu table that do NOT (!) have the column
# header "taxonomy"
otu3 <- otu3[,!names(otu3) == "taxonomy"]
```

Filtering OTUs and Samples

Let's filter the OTU table to keep only OTUs that are in at least 5 people, and that have at least 100 counts.

```
#Set the number of samples cut off
nsamples <- 5

# `otu > 0` tells R to take all values and see if they are greater
# than 0. If so it will store it as TRUE, if not greater than 0 they
# get a false. Then we take the `rowSums()` of that value, where
# TRUE=1 and FALSE=0. Then we ask if the row sums are greater than
# then number of samples we set as the cut off. It will store
# TRUE/FALSE values for each row.
cutoff_nsamples <- rowSums(otu3 > 0) > nsamples
# Keep only samples that are 'TRUE' (meet the cutoff value)
otu3 <- otu3[cutoff_nsamples,]

ncounts <- 99
# This cutoff is different than the previous. We care about how MANY
# counts each taxon has.  We only want to keep those with a minimum
# of 100 counts across all samples (greater than 99)
cutoff_ncounts <- rowSums(otu3) > ncounts
#Keep only taxa that meet the cutoff
otu3 <- otu3[cutoff_ncounts,]
```

Calculating relative abundances

We took out some taxa when filtering, so we need to convert count into the relative abundance of that sample. To do this, we will use a for loop

```
# We want to use all the columns (since we already took out taxonomy)
for(i in 1:ncol(otu3)){
  otu3[,i] <- otu3[,i]/sum(otu3[,i])
}
```

In order to make our OTU table and results more easily compatible with our metadata, we want the sample IDs as the rows and the taxa as the columns. We'll use the function `t()` to transpose the data frame. We can then make a column SampleID that will be useful later.

```
# Transpose as a data frame
otu3 <- data.frame(t(otu3))

# Make a column that is the Sample IDs (which are the rownames)
otu3$SampleID <- rownames(otu3)
```

```
# Let's save a backup of this filtered OTU table
otu_backup <- otu3
```

If you remember, `ggplot` likes to have all the data in one table. Now we will use a function called `melt()` from the library `reshape2` to convert our data frame into three columns: one that has the sample ID, one that has taxa IDs, and one that has the relative abundances of the taxa in our sample. We'll also use the package `plyr` for it's function `ddply()` to aggregate our data nicely.

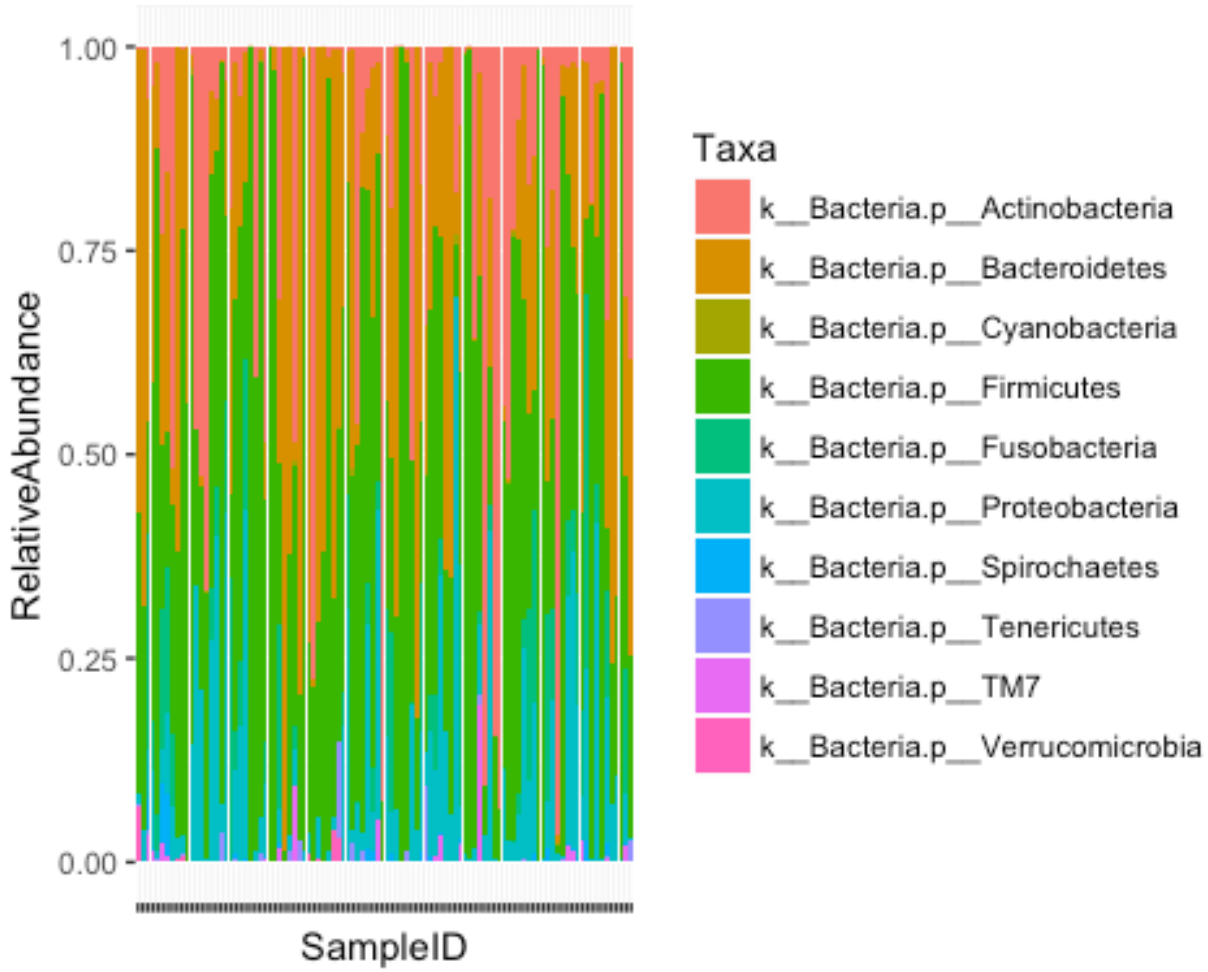
Before we can move forward, you must get the packages needed to run the functions we will use. You can install packages with the `install.packages()` function.

```
#You'll want to install these packages if you don't already have them
library(reshape2)
library(plyr)

otu3 <- melt(otu3, id.vars = "SampleID",
             variable.name = "Taxa",
             value.name = "RelativeAbundance")
```

Plotting

Now we have a filtered table with three columns with which we can make a basic taxa summary plot (just grouped by sample ID). We'll use `ggplot`.

```
library(ggplot2)
# This will make a plot with the OTU table (otu), using the column
# headers specified
ggplot(otu3, aes(x = SampleID, y = RelativeAbundance, fill= Taxa)) +
  geom_bar(stat = "identity", position="fill") +   # This makes it a bar plot
(geom_bar())
  scale_x_discrete(labels = NULL)    #This takes off the x-labels (too hard
to read)
```



This plot is kind of messy! There are so many samples you can't easily see one sample from another.

Adding Metadata

Let's try piloting by a covariate. The easiest way to do that is to simply add our metadata values to our table. More columns means more potential variables to plot by. Let's go back to our full, filtered OTU table, melt it and then add metadata.

```
otu3 <- otu_backup
otu3 <- melt(otu3,
             id.vars = "SampleID",
             variable.name = "Taxa",
             value.name = "RelativeAbundance")
```

Now we can add in our metadata, using the function `merge()`. First let's make sure we have the covariates/header names we think we do, and we can rename any that aren't right, and only

keep the ones we're interested in. If you're looking at real metadata, you'll have a much longer list than the tutorial files.

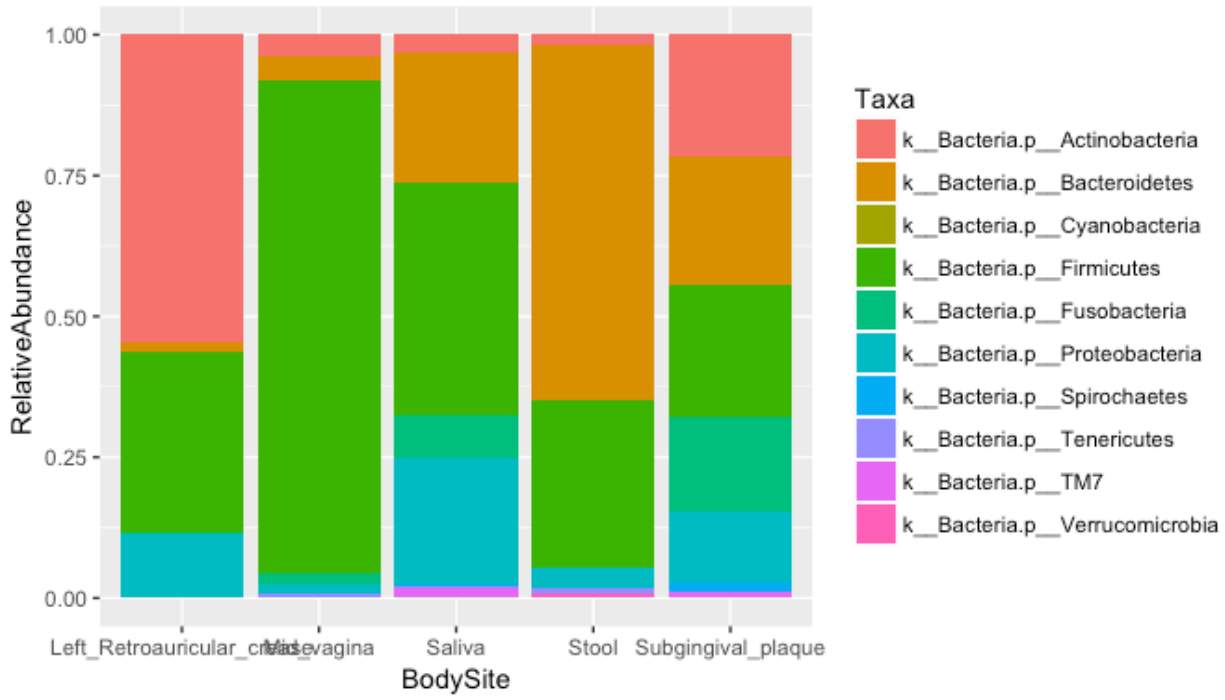
```
colnames(metadata)
```

```
## [1] "BarcodeSequence"      "LinkerPrimerSequence" "Sex"
## [4] "BodySite"             "SRS_SampleID"         "FASTA_FILE"
## [7] "Description"          "Age"
```

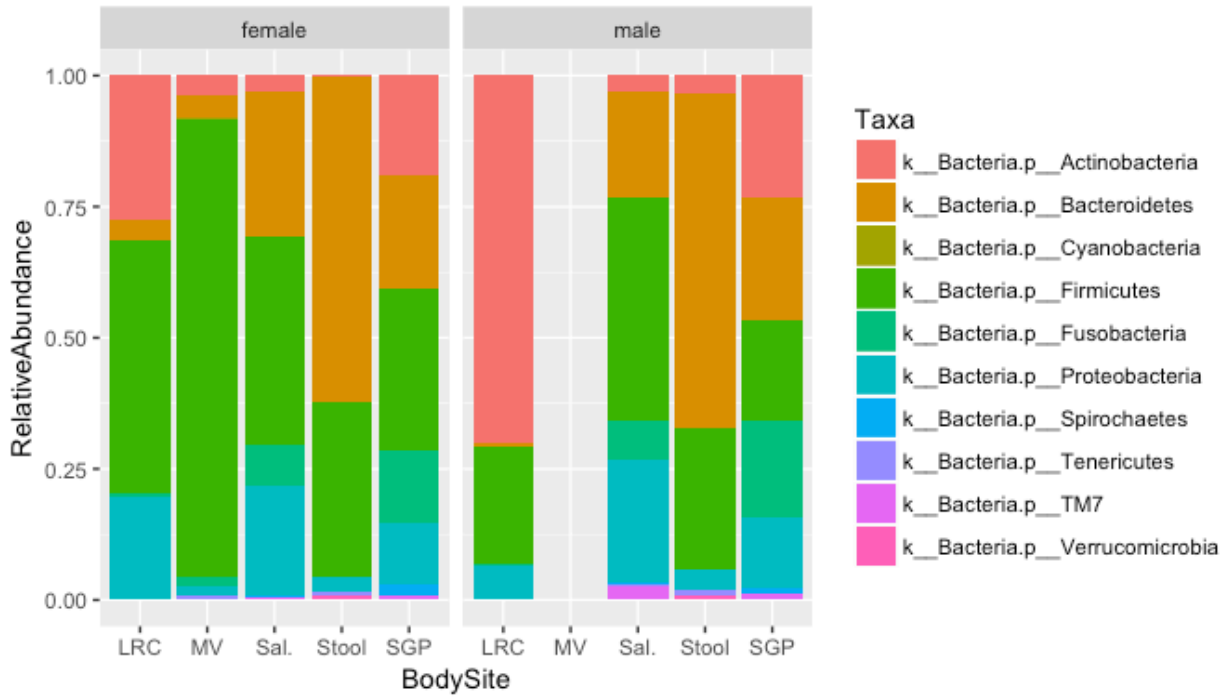
```
# We only want to keep "Sex", "BodySite", and
# "Description", which is the area of the body

#This will keep only the columns with the headers we want
columns_keep <- c("Sex","BodySite","Description")
metadata2 <- metadata[,columns_keep]

# Now we merge covariates to sample ids
# First we need to make a column that is the sample IDs in the
metadata2$SampleID <- rownames(metadata2)

# This will drop any samples in the mapping file that aren't in the OTU table
otu3 <- merge(otu3, metadata2, by="SampleID")
```

Now we can plot according to body site.

```
ggplot(otu3, aes(x=BodySite, y=RelativeAbundance, fill=Taxa)) +
  # using position="fill" makes sure it sums to 1
  geom_bar(stat ="identity", position="fill")
```




```
# We will want to shorten the x-labels.
# We can even split our data up by sex using this method,
# using an option called facet_grid():
ggplot(otu3, aes(x=BodySite, y=RelativeAbundance, fill=Taxa)) +
  geom_bar(stat ="identity", position="fill") +
  facet_grid(.~Sex) + # This will separate by sex
  scale_x_discrete(labels=c("LRC", "MV", "Sal.","Stool","SGP"))    # This
relabels the x axis
```



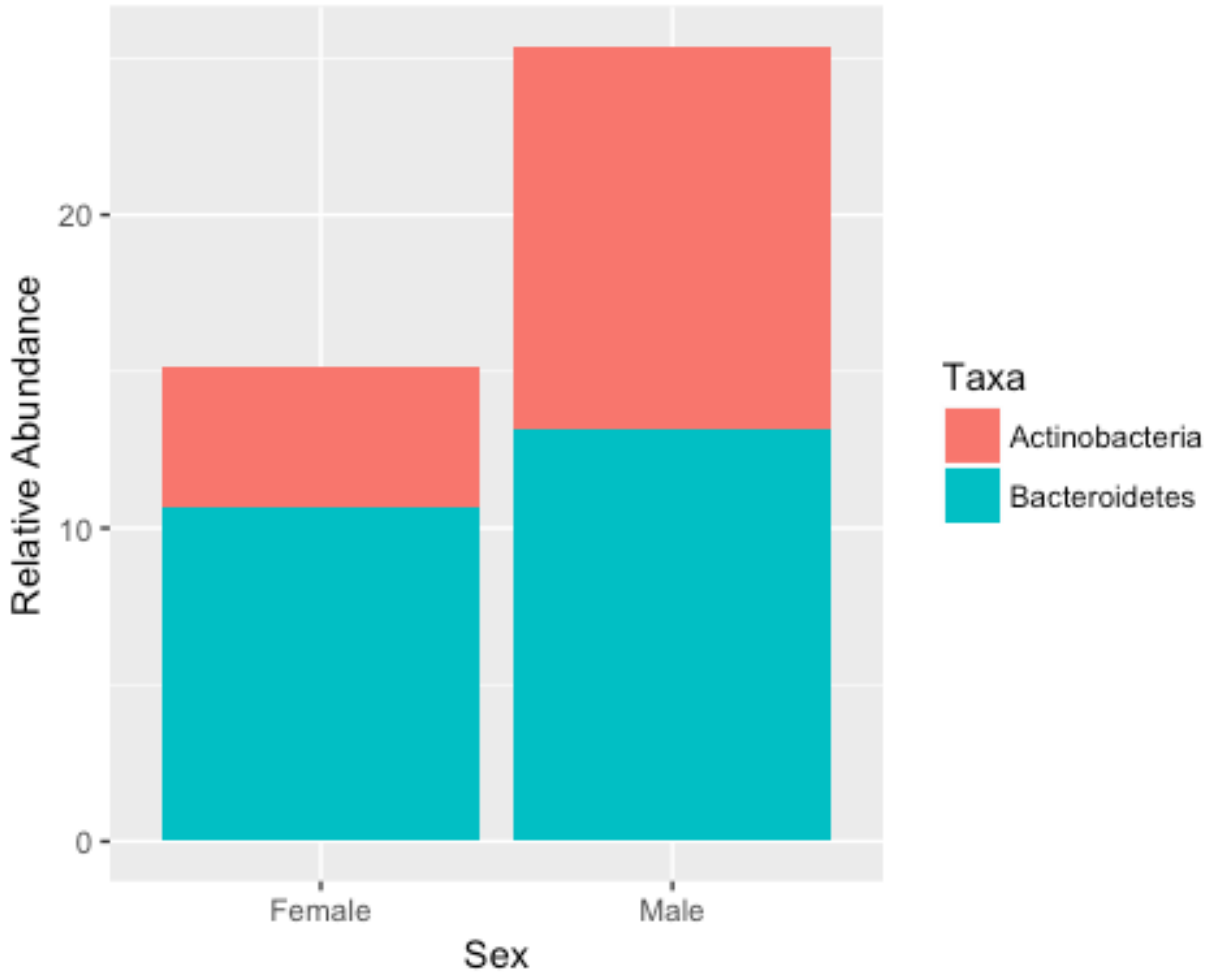
Plot Specific Taxa

We can also plot just specific taxa. For that, we can use the aggregated relative abundance table `otu` from the `sex` in the example above, and pull out a subset of the taxa we're specifically interested in. You'll need the exact taxa labels from the table to match. Say we want to look at Firmicutes and Actinobacteria:

```
# If we don't remember the spelling, we can print all the taxa and
# copy and paste:
unique(otu3$Taxa)

##  [1] k__Bacteria.p__Cyanobacteria   k__Bacteria.p__Tenericutes
##  [3] k__Bacteria.p__Fusobacteria    k__Bacteria.p__Bacteroidetes
##  [5] k__Bacteria.p__Verrucomicrobia k__Bacteria.p__TM7
##  [7] k__Bacteria.p__Actinobacteria  k__Bacteria.p__Spirochaetes
##  [9] k__Bacteria.p__Proteobacteria  k__Bacteria.p__Firmicutes
## 10 Levels: k__Bacteria.p__Actinobacteria ...
k__Bacteria.p__Verrucomicrobia

# Let's subset to just Bacteroidetes and Actinobacteria
taxaList <- c("k__Bacteria.p__Bacteroidetes",
"k__Bacteria.p__Actinobacteria")

# Let's make a new subsetted table that is just those phyla
filtered <- subset(otu3, is.element(otu3$Taxa, taxaList))
```

We plot things the same, making sure **not** to use the option `position="fill"`, since our abundances now should **not** add up to 1. Let's make our labels a little nicer, as well.

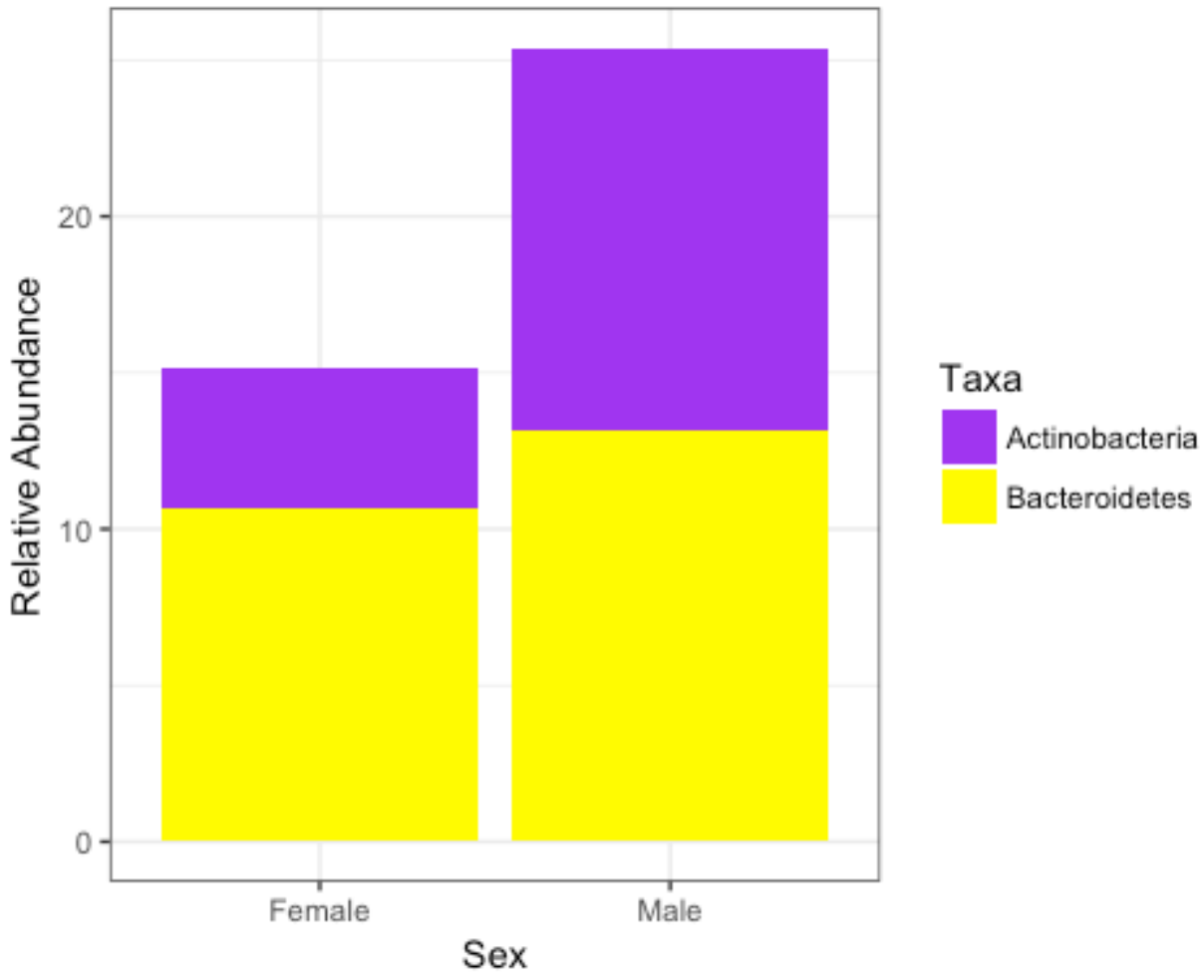
```
ggplot(filtered, aes(x = Sex, y = RelativeAbundance, fill=Taxa)) +
  geom_bar(stat="identity") +
  labs(y = "Relative Abundance") +
  scale_fill_discrete(labels = c("Actinobacteria", "Bacteroidetes")) +
  scale_x_discrete(labels = c("Female", "Male"))
```



There are almost unlimited parameters that you can play with to change the actual look of your plots. Below we use `theme_bw()` to make the background white, and modified the colors by making a color variable `cols` that we use to color the different taxa with `scale_fill_manual()`.

```
cols <- c("purple","yellow")

#Note that we have to use scale_fill_manual() instead of scale_fill_discrete
# to specify colors
```

```
ggplot(filtered, aes(x = Sex, y = RelativeAbundance, fill=Taxa)) +
  geom_bar(stat="identity") +
  labs(y = "Relative Abundance") +
  scale_x_discrete(labels = c("Female", "Male")) +
  theme_bw() +
  scale_fill_manual(labels = c("Actinobacteria", "Bacteroidetes"),
values=cols)
```



Differentiated OTUs in R

We can test for taxa or OTU that are differentially abundant across sample types. To do this, we need to first transform our data out of the simplex. This means we want to go from working with compositional data to non-compositional data.

Inputs

The input data you need include the metadata and your OTU table that has low depth samples removed. Don't use the rarefied OTU table. The tables should be subsetted and ordered for sample ID.

First we will take the taxonomy out of the OTU table, filter low abundant OTUs and low occurring OTUs:

```
# We can store taxonomy and which OTUs they are to use for later
# drop=F makes sure it stays as a table
taxonomy_table <- otu_low2[,"taxonomy",drop=F]

#Keep only the samples, drop taxonomy from table
otu_low3 <- otu_low2[, ! names(otu_low2) =="taxonomy"]

#Filter OTUs that are in low abundance
#Change those less than 1/1 millionth of read depth to 0
otu_low3[otu_low3 < sum(colSums(otu_low3))/1000000] <- 0

#Change singletons to 0 (needed for low depth OTU tables)
otu_low3[otu_low3 < 2] <- 0

#Filter the OTU table to keep OTUs in at least 5% of samples
otu_low3 <- otu_low3[rowSums(otu_low3 > 0) > (0.05*ncol(otu_low3)),]
```

Now we will transform the data using a centered log-ratio transformation. This needs the robCompositions package.

```
library(robCompositions)
```

```
## Warning: package 'robCompositions' was built under R version 3.3.2
```

```
## Loading required package: robustbase
```

```
## Warning: package 'robustbase' was built under R version 3.3.2
```

```
## Loading required package: data.table
```

```
##
## Attaching package: 'data.table'
```

```
## The following objects are masked from 'package:reshape2':
##
##     dcast, melt

## Loading required package: e1071

## Warning: package 'e1071' was built under R version 3.3.2

## Loading required package: pls

## Warning: package 'pls' was built under R version 3.3.2

##
## Attaching package: 'pls'

## The following object is masked from 'package:stats':
##
##     loadings

## sROC 0.1-2 loaded

##
## Attaching package: 'robCompositions'

## The following object is masked from 'package:robustbase':
##
##     alcohol
```

```
#Convert any 0 to 0.65 to allow for CLR transform
#Ref: Palarea-Albaladejo J, et al. 2014. JOURNAL OF CHEMOMETRICS. A bootstrap estimation scheme for chemical compositional data with nondetects. 28;7:585-599.
otu_low3[otu_low3 == 0] <- 0.65

#Centered log-ratio transform for compositions
#Ref: Gloor GB, et al. 2016. ANNALS OF EPIDEMIOLOGY. It's all relative: analyzing microbiome data as compositions. 26;5:322-329.

#convert to samples as rows
otu_table <- t(otu_low3)

#Centered log-ratio tranform the data
otu_table <- cenLR(otu_table)$x.clr
```

Test For Differences

Now our otu table has samples as rows and OTUs as samples. We can now loop through the OTUs and test for differences according to our metadata. Lets test for differences by bodysite.

Because we transformed our data, we can now use parametric tests to look for differentiated OTUs. We can use ANOVA or t-test depending on the number of groups to test.

```
# Let's test the first OTU (first column) in the OTU table
# What is the name of this OTU? We can look it up in our table
# We pick the row we want using the otu id in the column
this_taxa <- taxonomy_table[colnames(otu_table)[1],"taxonomy"]
this_taxa
```

```
## [1] "k__Bacteria; p__Firmicutes; c__Bacilli; o__Lactobacillales;
f__Lactobacillaceae; g__Lactobacillus; s__"
```

```
# Now lets run the test using the first column and according to bodysites in
the metadata
aov_test <- aov(otu_table[,1] ~ metadata$BodySite)
summary(aov_test)
```

```
##                   Df Sum Sq Mean Sq F value   Pr(>F)
## metadata$BodySite  4  47.77  11.942   27.95 1.91e-15 ***
## Residuals         97  41.44   0.427
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

The output from `aov()` is more complicated than `kruskal.test()`. `aov()` output is a list that includes information about the degrees of freedom (`Df`), the Sum of Squares (`Sum Sq`), the Mean Square (`Mean Sq`), the F statistic/ratio (`F value`); and the P-value (`Pr(>F)`). For now, we are interested in the **p-value**, which can be indexed from `summary(aov_test)` with `summary(aov_test)[[1]][1,5]`.

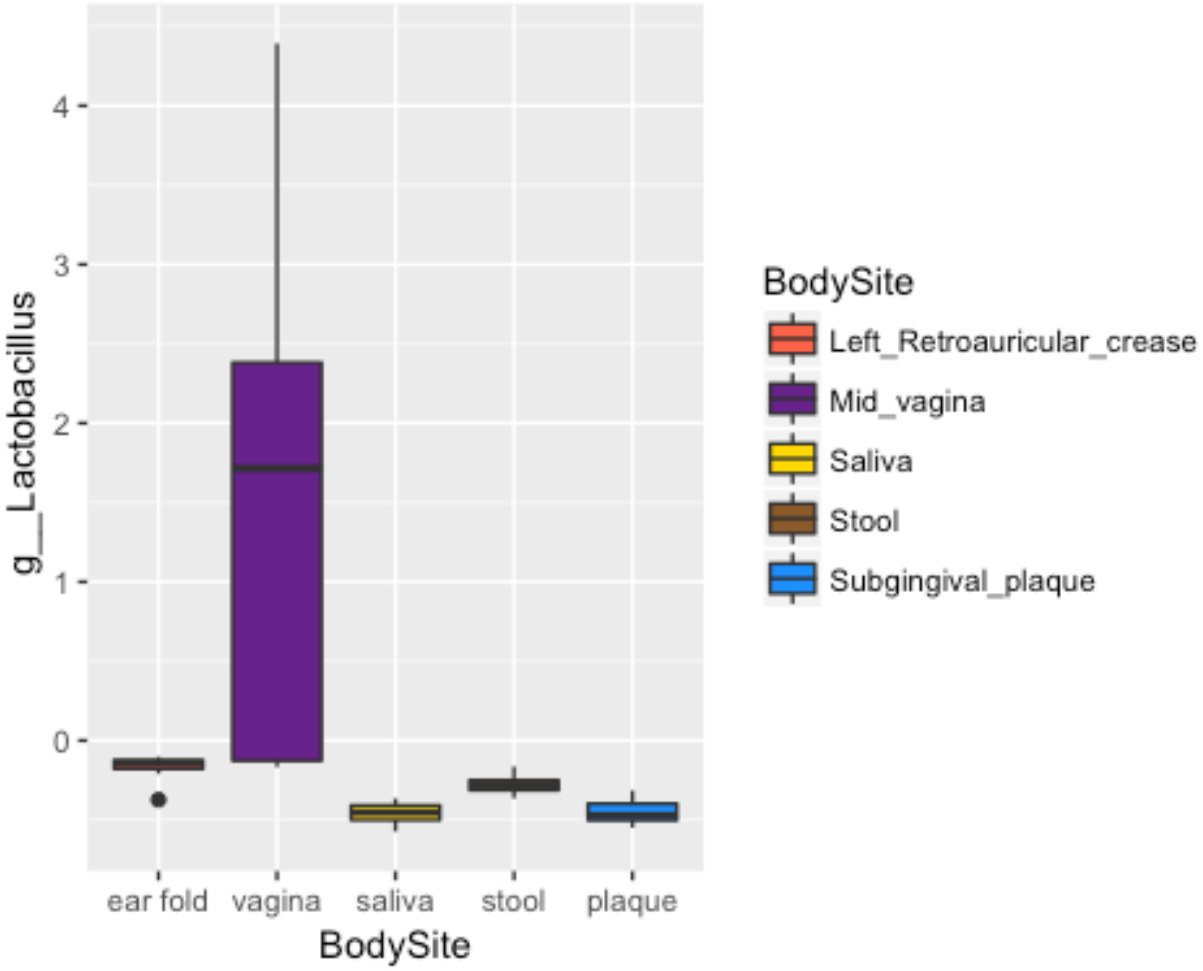
```
summary(aov_test)[[1]][1,5]
```

```
## [1] 1.913061e-15
```

Let's plot this example and see what it looks like.

```
# Because ggplot likes to have all the data in one table, let's make a new
table to plot with
plot_table <- data.frame(otu_table)
#Note that this will store and x infront of all the numerical column names
plot_table$BodySite <- metadata$BodySite

#store which column (header) you want to plot
this_otu <- colnames(plot_table)[1]

# We can also store its name
# will split the taxonomy based on the ";"
```

```
# Then take the last two values (genus and species) to shorten the name
name = strsplit(this_taxa, ";", fixed=T)[[1]]
names_tail = tail(name, n=2)

# This will plot the transformed abundnces for each body site
# Note that we have to use aes_string() because we are filling in the y
column header with a string
ggplot(plot_table) +
  geom_boxplot(aes_string(x="BodySite", y=this_otu, fill="BodySite")) +
  scale_fill_manual(values = c("tomato", "darkorchid4",
"gold","tan4","dodgerblue")) +
  scale_x_discrete(labels=c("ear fold", "vagina", "saliva", "stool",
"plaque")) +
  labs(y=names_tail)
```



Test All Taxa

Using for-loops we can apply this test to all of the OTUs in our table. In the loop, the transformed abundance of each OTU for all samples will be assigned to the variable 'y', the **dependent variable**. The next line in the loop calls the `aov()` function, and the last line assigns each p-value to a pvals vector.

```
#The first step is to make an empty vector that will store our p-values.
pvals <- c()

#Loop through each column except the last (because it's body site)
for(i in 1:(ncol(plot_table)-1)){
  aov_out <- aov(plot_table[,i] ~ plot_table$BodySite)
  pvals[i] <- summary(aov_out)[[1]][1,5]
}
```

Find Significant p-Values

Let's see how many p-values are significant for each covariate. We will use an **alpha** of 0.05.

```
sum(pvals < 0.05)
```

```
## [1] 579
```

False Discovery Rate

Because we did so many statistical comparisons, we need to correct for type I errors (rejection of a true null hypothesis, also known as a *false positive*). Controlling the false discovery rate helps to control the expected proportion of false positives. To do this we use the `p.adjust()` function with the `'fdr` parameter.

```
pvals.fdr = p.adjust(pvals, "fdr")
```

Let's see how many p-values are significant for each covariate after the false discovery rate correction.

```
sum(pvals.fdr < 0.05)
```

```
## [1] 577
```

Plotting Significant OTUs

If we wanted to plot all the significantly different taxa we could do so with a for loop. We will plot the first three significantly different taxa across the body site.

```
# Index just the first three significantly different OTUs
# which() tells us the position of the values that are true (< 0.05), and
[1:3]
# takes the first 3.
```

```r
first_three <- which(pvals.fdr < 0.05)[1:3]

# This loops through the significant OTUs, stores their name
# and makes a box plot of the transformed abundances of the taxa
# We then store the plots in a list
plot_list <- list()
for(i in 1:length(first_three)){
  index <- first_three[[i]]
  this_otu <- colnames(plot_table)[i]
  this_taxa <- taxonomy_table[i,"taxonomy"]
  name <- strsplit(this_taxa, ";", fixed=T)[[1]]
  taxon <- paste(name[6], name[7], sep=" ")
  # Note that we have to use aes_string() because we are filling in the y
column header with a string
  plot_out <- ggplot(plot_table) +
    geom_boxplot(aes_string(x="BodySite", y=this_otu, fill="BodySite")) +
    scale_fill_manual(values = c("tomato", "darkorchid4",
"gold","tan4","dodgerblue")) +
    scale_x_discrete(labels=c("ear fold", "vagina", "saliva", "stool",
"plaque")) +
    labs(y=taxon)
  plot_list[[i]] <- plot_out
}

# Now lets print the three plots to a pdf
# each plot will be a new page in the pdf
pdf("Diff_taxa.pdf", height=4, width=6)
for(i in 1:length(plot_list)){
  plot(plot_list[[i]])
}
dev.off()
```

```
## quartz_off_screen
##                 2
```

PCoA in R

We use QIIME to calculate our distance matrices using `beta_diversity.py` or `beta_diversity_through_plots.py` command. We then can use R to make 2D PCoA plots of this data. Let's start with the packages we need to load. If these are not installed you can install them first with `install.packages()`.

```
library(ape)
library(vegan)
library(ggplot2)
```

Load Data

You'll need to have your beta diversity and metadata files loaded and subsetted to the correct number and order of samples.

Principal Coordinates Analysis

Now we can use the function `pcoa()` from the R package ape to actually calculate our principal coordinate vectors. To make plotting easier, we save the vectors as a data frame, set up new column titles, and add a column of sample IDs.

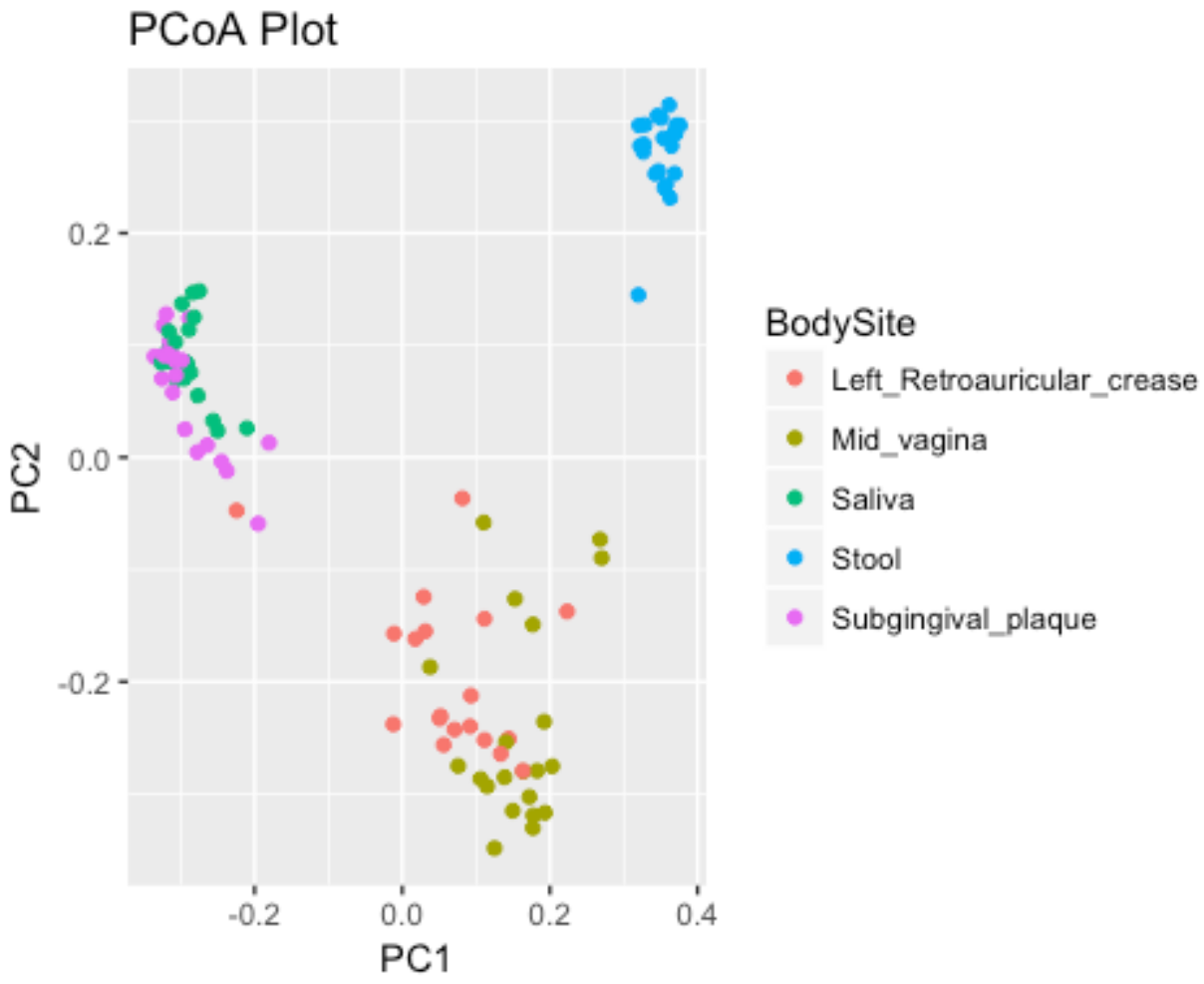
```
# Run the pcoa() function on the beta diversity table,
# and store the vectors generated as a dataframe
PCOA <- data.frame(pcoa(beta)$vectors)

# If you look at the PCOA table, you'll see the column names
# are the 'axes' and the row names are sample IDs. We want them to
# be labeled "PC" instead of "axis"

# We will make a vector with place holders
new_names <- rep("", ncol(PCOA))

# Fill in first with PC followed by the number (e.g. PC1, PC2, PC3...)
for(i in 1:ncol(PCOA)){
  new_names[i] <- paste("PC",i, sep="")
}

# Replace the column names of PCOA
names(PCOA) <- new_names

# Create a column that is SampleIDS for PCOA
PCOA$SampleID <- rownames(PCOA)

#Create a column that is SampleIDs for the metadata
metadata$SampleID <- rownames(metadata)
```

```r
# Merge the metadata and beta diversity
PCOA <- merge(PCOA, metadata, by = "SampleID")
```

Plotting the PCoA

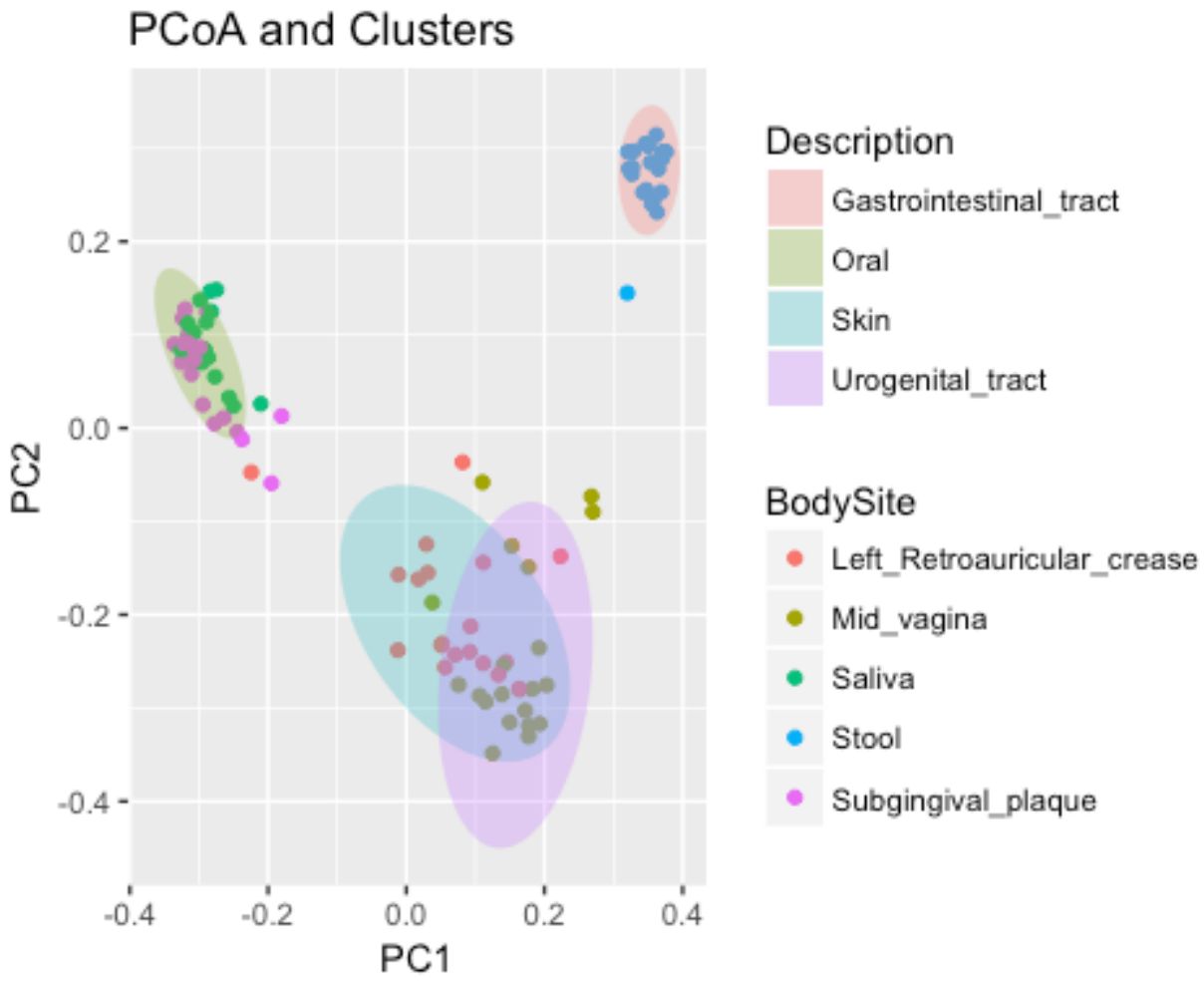
Now you have a data frame that has all of your PCOA vectors and all the relevant metadata, matched up by sample ID. In this example we will plot the first two principal coordinates (PC1 and PC2). If you remember, the first principal coordinates should explain the majority of the variation in the data. These will be pretty simple scatter plots.

```r
# Note that geom_point() makes it a scatter plot where the points
# are colored according to BodySite
ggplot(PCOA) +
  geom_point(aes(x = PC1, y = PC2, color = BodySite)) +
  labs(title="PCoA Plot")
```




```r
# Now let's add some clusters.  This makes it look great, but can
# also be misleading and make us think there are groups when there
```

```
# aren't. Note that we are using BodySite to color the points and body
# AREA to fill the clusters
ggplot(PCOA) +
  geom_point(aes(x = PC1, y = PC2, color = BodySite)) +
  labs(title="PCoA and Clusters") +
  stat_ellipse(alpha = 0.3, geom="polygon", linetype="blank", aes(x = PC1, y
= PC2, fill = Description))
```



Notice that the color of the ellipses don't really match the color of the points they are clustering. The colors are determined by which order the body area is factored by. We can make this order line up with the order of the body sites.

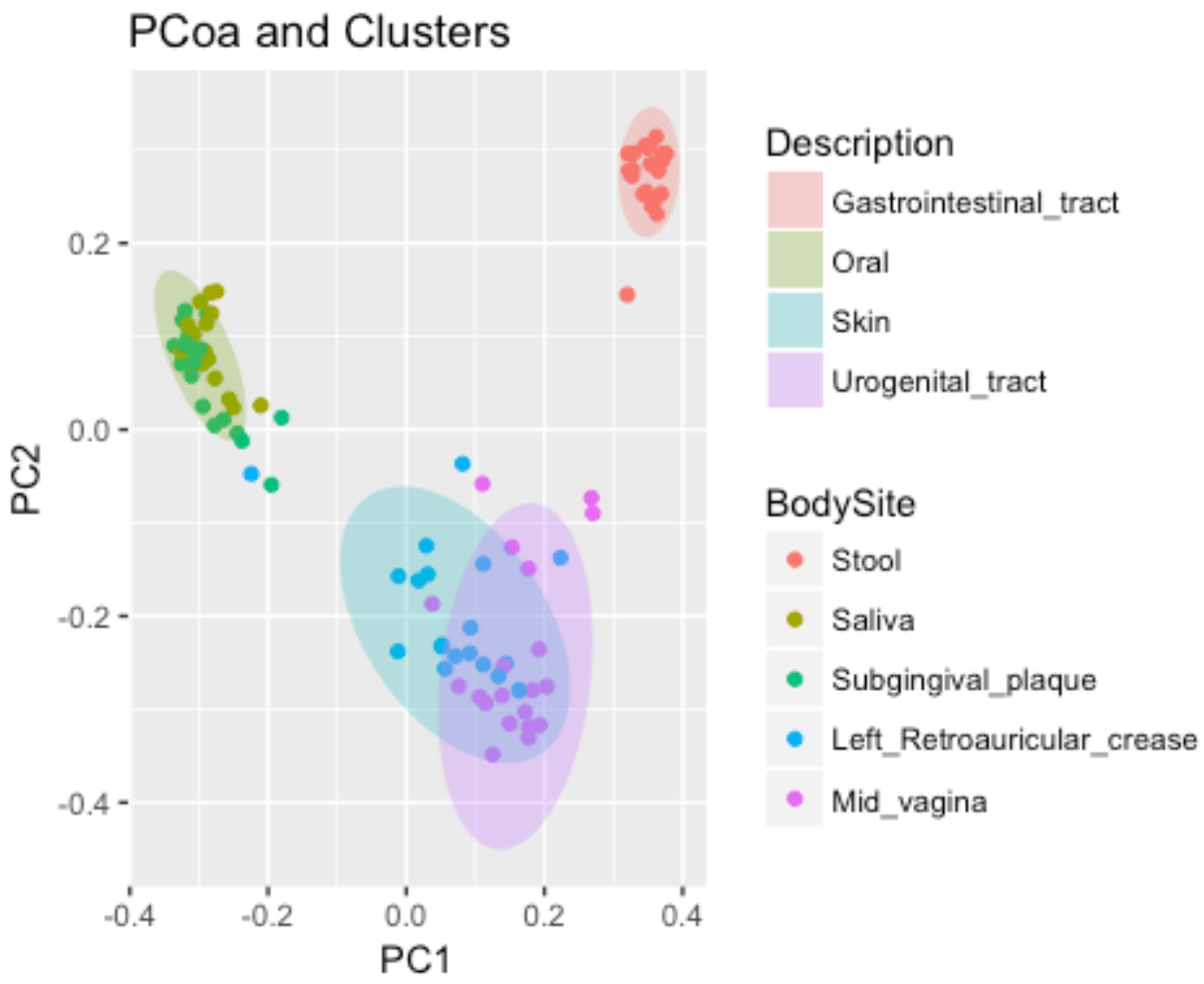
```
# Check order of levels of body area (Description)
levels(PCOA$Description)
```

```
## [1] "Gastrointestinal_tract" "Oral"
## [3] "Skin"                   "Urogenital_tract"
```

```r
# Check order of levels in BodySite
levels(PCOA$BodySite)
```

```
## [1] "Left_Retroauricular_crease" "Mid_vagina"
## [3] "Saliva"                     "Stool"
## [5] "Subgingival_plaque"
```

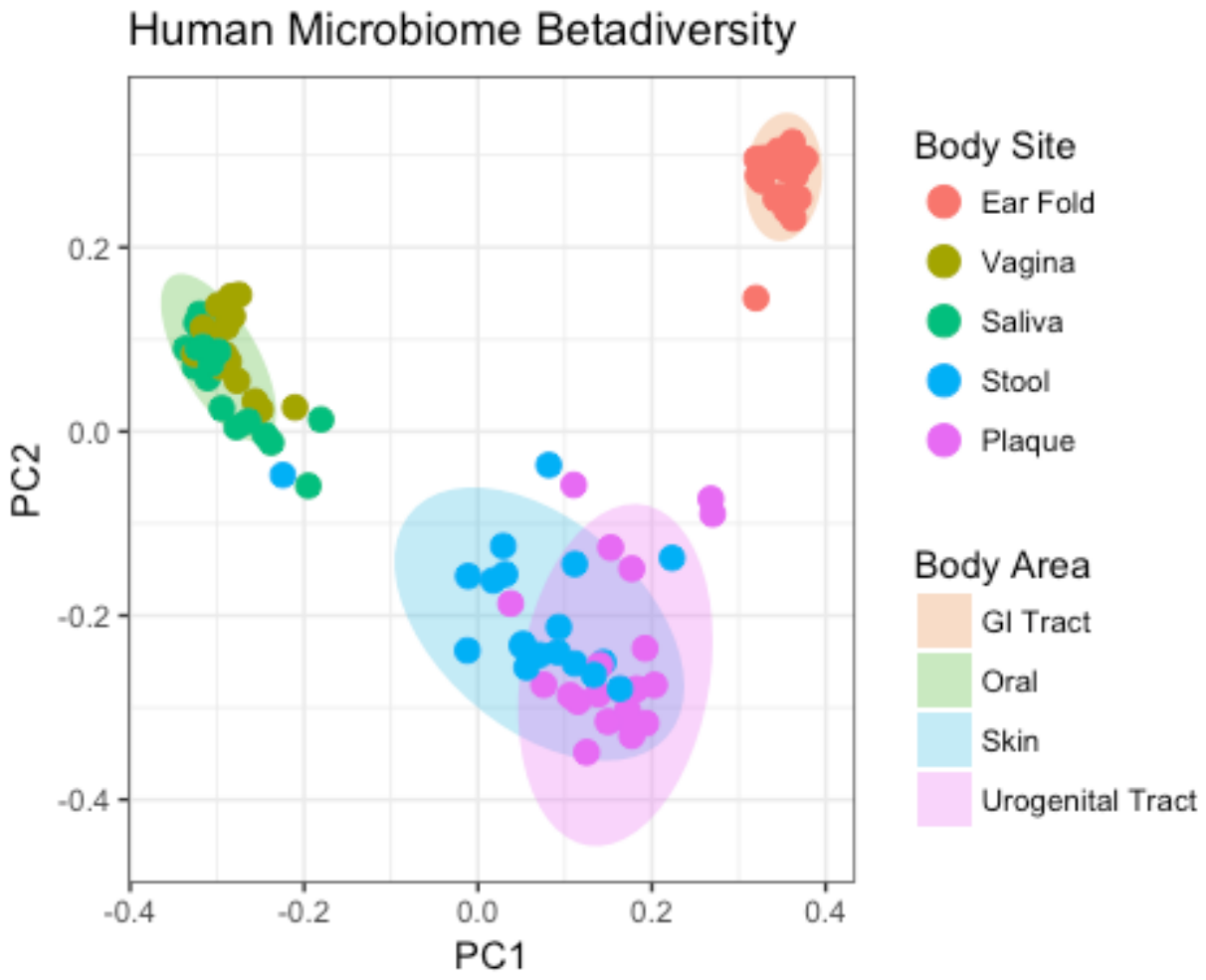
```r
# Reset levels of Bodysite to match levels of body area
PCOA$BodySite <- factor(PCOA$BodySite, levels =
                          c("Stool", "Saliva", "Subgingival_plaque",
                            "Left_Retroauricular_crease", "Mid_vagina"))


#Replot
ggplot(PCOA) +
  geom_point(aes(x = PC1, y = PC2, color = BodySite)) +
  labs(title="PCoa and Clusters") +
  stat_ellipse(alpha = 0.3, geom="polygon", linetype="blank",
               aes(x = PC1, y = PC2, fill = Description))
```



Changing Plotting Parameters

The following long command throws a whole pile of customization bells and whistles at ggplot - the fill colors are tweaked, the points are a bit bigger, the font sizes are bigger. This is just to give you a taste of all the different aesthetic options you can play around with. You should try modifying each parameter and see what it does to the plot.

```
ggplot(PCOA) +
    stat_ellipse(alpha = 0.3, geom="polygon", linetype="blank",
                 aes(x = PC1, y = PC2, fill = Description)) +
    geom_point(size = 3, aes(x = PC1, y = PC2, color = BodySite)) +
    labs(title="Human Microbiome Betadiversity") +
    scale_color_discrete(name = "Body Site",
          labels = c("Ear Fold","Vagina", "Saliva", "Stool","Plaque")) +
    scale_fill_hue(h.start = 20,
          name = "Body Area",
          labels = c("GI Tract", "Oral","Skin", "Urogenital Tract")) +
    theme(panel.background = element_rect(color = "grey97"),
          plot.title = element_text(size = 16),
          axis.title = element_text(size = 14),
          axis.text = element_text(size = 12),
          legend.title = element_text(size = 14),
          legend.text = element_text(size = 12)) +
          theme_bw() +
    guides(color = guide_legend(override.aes = list(fill = "grey97", size =
4)),
           fill = guide_legend(override.aes=list(shape = NA)) )
```



Testing for Signifcant Differences

Just like we did for alpha diversity, we can test for significant differences in beta diversity. For example, let's say we want to test for significant differences between body sites with the Unweighted UniFrac data. We already loaded that data when ran it through the `pcoa()` function above. But since we are learning let's load it again so we can get comfortable with the code.

adonis

adonis is a non-parametric statistical test, which means it uses permutations of the data to determine the p-value, or statistical significance.

It requires:

* a distance matrix file, such as a UniFrac distance matrix
* a mapping file, and a category in the mapping file to determine sample grouping from

It computes an $\mathbf{R^2}$ value (effect size) which shows the percentage of variation explained by the supplied mapping file category, as well as a **p-value** to determine the statistical significance.

More information of the adonis test can be found here:
http://qiime.org/tutorials/category_comparison.html,
http://cc.oulu.fi/~jarioksa/softhelp/vegan/html/adonis.html

```
# Turn the beta table into resemblance matrix using as.dist()
beta_dist = as.dist(beta)

# Test for a significant difference across all groups.
# This will run an ADONIS test.
ad = adonis(beta_dist ~ metadata[,"BodySite"], data=metadata,
permutations=999)
ad

##
## Call:
## adonis(formula = beta_dist ~ metadata[, "BodySite"], data = metadata,
permutations = 999)
##
## Permutation: free
## Number of permutations: 999
##
## Terms added sequentially (first to last)
##
##                          Df SumsOfSqs MeanSqs F.Model      R2 Pr(>F)
## metadata[, "BodySite"]   4    12.725  3.1811  15.789 0.39433  0.001 ***
## Residuals                97    19.544  0.2015         0.60567
## Total                   101    32.269                 1.00000
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

Note:
Pr indicates that at an alpha of 0.05, the grouping of samples by 'BodySite' is statistically significant.

The **R2** value indicates that approximately 39% of the variation in distances is explained by this grouping. It's important because a p-value can indicate significance but we must also notice how much of the variation the input variables contribute.

Now let's write our output to a file.

```
# This takes just the analysis of variance table (aoc.tab)
# from the output
a.table <- ad$aov.tab

# This writes it to a file
```

```
write.table(a.table, file="analysis.txt", quote=FALSE, sep="\t", col.names =
NA)
```