Biomedical Data Analysis Protocol

Samuel Bergin

Matriculation No. : 22208243

The Set-up:

Before anything could be done related to Data Analysis first the virtual machine had to be set-up for use in Data Analysis.

The first step in this process was to install Miniconda onto the virtual machines.

Conda is a program that is part of the software package Anaconda or its smaller version Miniconda, which contains only Conda and its dependencies. It allows the generation of environments so that a user can install programs in different places on the same machine. In our case Conda was used to generate a bio environment which would be used to store our programs related to Biomedical Data Analysis. If I were to try to run one of our installed programs outside of the bio environment, nothing would happen as those programs are only installed on the bio environment.

To install Miniconda we use the following line of code:

```
sudo apt install wget
wget https://repo.anaconda.com/miniconda/Miniconda3-latest-Linux-x86_64.sh
bash Miniconda3-latest-Linux-x86_64.sh
```

Once Conda is installed the next step is to add channels from Conda to our machines. A channel in Conda is a grouping of URLs that link to packages that are related to that particular channel. The channels we used were “bioconda” and “conda-forge”. To add these channels we use the following code:

```
conda config --add channels new_channel
```

Where new_channel is the name of the channel that we wish to add.

With these channels added it is now possible to create and activate our environments. These are very important parts of the Conda system. Environments are like subsections of your virtual machine and allow for the installation of certain packages to different sections of your machine to avoid unnecessary clutter. To create an environment we use the code:

```
conda create --name myenv
#myenv is the name of the environment
#accept by typing y
#can select specific version of python with following code
conda create --name myenv python=3.9
```

And then these environments that were created can be activated by typing:

```
Conda activate myenv
```

You can use this code to swap between environments as well. Conda will always create an environment called base. This is where your machine will always start up and acts as your hub.

After creating environments we install a few packages off of Conda that will be useful for Biomedical Data Analysis (fastqc, jupyterlab, and bash_kernel). Each of these programs are useful. “fastqc” is used to run quality control tests on sequencing data, outputting a html file that details the quality of the sequenced data and other important pieces of information that determine the quality of your data. “jupyterlab” is used to create a connection between the online notebook service Jupyterhub and your virtual machine allowing the manipulation of files using Jupyterhub as long as jupyterlab is running on your virtual machine. Finally bash_kernel works with jupyterlab to allow the use of bash kernels within Jupyterhub.

To install each of these packages the following code was used:

```
conda install package_name
#package_name being the name of the package you wish to install

conda install -c channel_name package_name
# -c specifies the channel that you are installing from
```

In the case of each of fastqc we are installing from bioconda so we enter bioconda after -c. For jupyterlab and bash_kernel we are installing from conda-forge and so we enter conda-forge after -c.

When installing these packages it is important to ensure that you have activated the environment that you want these packages to be installed in. You cannot select the environment you want to install to in the code so you must be in the environment that you want.

By typing `conda env list` we can list all of the environments that have been created so far, and by typing just `conda list` we can list all of the programs in your current environment.

Jupyter:

Jupyter is a very helpful program which allows the creation of notebooks where you can both store text and write executable code in many different programming languages. However, to use these languages you first must register the language. We have seen above how to register bash kernels into jupyter but now we must register R.

To do this we first entered this line of code into the virtual machine:

```
R -e “IRkernel::installspec()”
```

Jupyter will already have python registered so there is no need to do anything to use python.

Once all wanted languages are registered we then set up the jupyter server that we would use. To do this we first generate a server and the password for your personal server:

```
Jupyter server –generate-config

Jupyter server password
```

The second command will then prompt you to create a password for your server.

You can then start your server using the command:

```
Jupyter lab –no-browser –ip "*"
```

The server will open however, you cannot manipulate anything within the server from the virtual machine. To get around this you can exit and then re-log in to your virtual machine with an addition to the code:

```
ssh ubuntu@ip -I /path/to/key -L 8888:localhost:8888
```

This addition allows the user to connect to the jupyter server in both the virtual machine and your browser. Since Jupyter is a browser application you can then manipulate things on the jupyter server.

The next problem is this process will not start up again once you restart your virtual machine. The moment you exit your virtual machine the whole process ends and is removed from your machine. This means that each time you want to use the jupyter server you must re-run the code.

To solve this problem we used tmux to create a new session that runs the process to start the jupyter server.

tmux is a software package that allows the user to create sessions which persist after the virtual machine is closed. These sessions act as separate machines that are connected to your base machine.

tmux is installed the same way any other conda package is intalled:

```
conda install -c bioconda tmux
```

A new session can then be generated :

```
tmux new -s mysession
#mysession is the name of the session
```

Once a new session is created you can enter(attach) a session using the code `tmux a -t mysession` or you can connect to just the most recent session by typing just `tmux a`.

Each session has the option to generate windows and panes. Windows are basically tabs within your session, you can have multiple windows which you can swap between and contain different processes running at the same time. Each window can then be split into panes. These are sub sections within your window that act as separate terminals to the

same window. For example you could have a window that contains an important data analysis process. On one pane you can have the text file open to manipulate the code, and on the other pane have the terminal open to test the code without having to exit the text editor.

With tmux running our server process we could then enter our jupyter server on our browsers and begin using jupyter notebooks to write and store code related to Biomedical Data Analysis.

It is important to understand how to store text in jupyter notebooks. By default jupyter notebooks start a cell in code mode. This means that anything you type, the notebook is going to try to run as executable code, which is only going to cause problems. Thus you must switch the cell over to Markdown mode.

Markdown is a language for jupyter which just allows you to type into cells in your notebook like you are typing in MS Word.

There are a few peculiarities to Markdown however. For starters, to create a header you must put # before whatever you are typing. Adding more hashtags makes the header smaller and smaller.

If you want to make your text appear in bold, italics or both you can use asterixis. Bold text uses two asterixis either side e.g **this test will be bold**. Italics uses one either side e.g *this text will be italicized*. To have the text be both italicized and bold you use three asterixis e.g ***this text will be both bold and italicized***.

Markdown is interesting in that numbered lists are really easy to create. As long as your text begins with a number with a full stop after, it will generate a numbered list. In fact you don't need to enter the numbers in order, even if you only enter 1. it will create an accurately numbered list. The easiest way to create a non-numbered list is to make the list the same way as a numbered list but instead of numbers use a hyphen (-).

You can create blockquotes within your notebook by adding > before your text, this will indent and quote your text which can be very helpful for quotations within a report.

Tables are a little more tricky to generate. To generate a table you must use hyphens and pipes(|) to make the general shape of the table you wish to generate and then jupyter will generate a table from that information. For example (without the quotation marks):

"|Title 1|Title 2|"

"|---------|--------|"

"|Item 1|Description|"

"|Item 2|Description|"

Depending on the version of Markdown you are using you can generate footnotes in your notebook by typing [^1] after your text you wish to footnote. The 1 can be substituted with whatever you wish the footnote to look like. At the end of your notebook you must then

retype what your footnote looks like with a colon (:) after and then whatever you wish to say. This creates a link to that text which can be clicked on. As mentioned this is only in certain versions of Markdown, our version this time around did not have this feature and so it did not work for us.

Finally, Markdown allows for the highlighting of syntax within code. This is achieved by typing: ```` ```json ```` above the code and ```` ``` ```` at the bottom.

Sequencing Data:

With our virtual machines set up the next thing to understand is what our data is and what it means. In Biomedical Data Analysis we are analysing sequencing data generated in a wet lab, however the sequencing data itself is not all that we need. For this data analysis we also need reference genomes and annotations which are specific to the gene we are sequencing. It is important to understand what file type each of these files are because that will tell us what kind of data is provided.

Raw sequencing data is provided in a FASTA format. This is a file format which stores either nucleotide or protein sequences and lists them with the name of the sequence and some other arbitrary information that is not as important. This is our base file that the rest of the analysis will use.

Often FASTA files will be converted into a FASTQ file. This is a very similar file type however there are two extra lines added, a separator line that only holds a +, and a quality score line. The Q in FASTQ stands for quality because this file type informs us of the quality score for each base that has been sequenced. This quality score is provided in an ASCII format and there is a table online that will help you convert these scores into numerical values.

The GTF file format is a file type which details the genetic features of a sequence. Usually, your annotations are going to be in a GTF file format. The GTF file format provides 9 pieces of information,

1. seqname: The name of the sequence or the gene ID whichever is provided, in the case of chromosomes the name can be given with or without the prefix ‘chr’ depending on how you would prefer it.
2. source: The name of the program which generated the data
3. feature: The feature type e.g Gene, Variation etc.
4. start: Gives the position of the start codon on the DNA.
5. end: Gives the position of the stop codon on the DNA.
6. score: Gives a floating point value, it is usually filled with a “.” meaning there is no given information or the data is not stranded.
7. strand: If the data is stranded tells us if it is a forward strand (+) or a reverse strand (-).
8. frame: Tells us the position of the first base on a codon. 0 means it is on the first base of a codon, 1 means it is on the second and 2 means it is on the third.

9. attribute: Is separated from the rest of the columns by a semicolon and gives additional information on the feature.

Quality Control:

Once we understood that we began to learn how to analyze sequencing data. The first step is quality control. It is important to check the quality of the sequencing data. If the quality score is too low there is no point in analyzing the data because the resulting data will be inaccurate.

We used `wget` to download sequencing data, annotations, a reference genome and an adapter sequence. All of which will be important later.

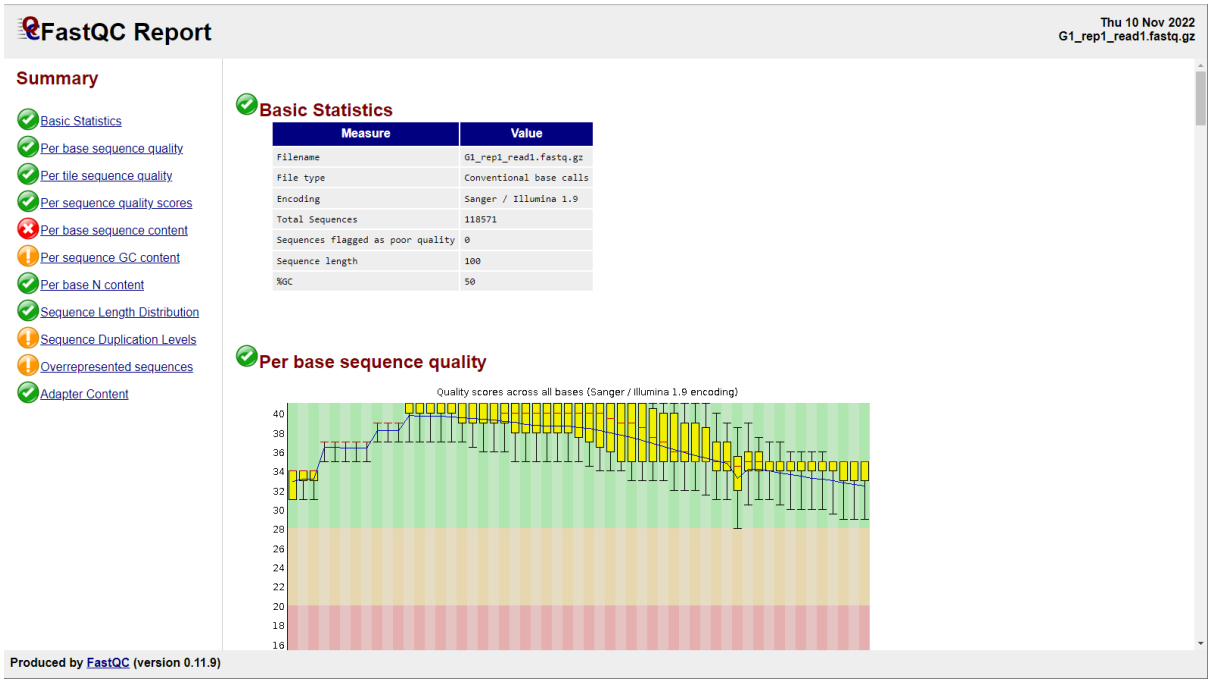
Quality Control is run by the program fastqc. This program takes the given data and runs a full quality check on the given FASTQ data and provides a HTML file which will provide not only written data but visuals to help understand the quality score.

We first installed fastqc onto our virtual machines:

```
conda install -c bioconda fastqc
```

Using fastqc we can then run a quality control check on individual sequencing data files:

```
fastqc seq_data
#seq_data being the file that contains the sequencing data
```



The per base sequence quality section gives most of the information that you will need from fastqc. The different coloured bands inform you on the acceptability of the quality score. Anything found within the green band is perfectly acceptable, anything in the yellow is questionable and can be used but really shouldn't, and anything in the red is really low quality. If any base is in the red section you should throw out that sequencing data and get

new data. The graph uses boxplots as the quality score naturally has some wiggle room regarding error and so it's best to show the range of possible quality scores.

Adapter Trimming:

Adapter Trimming is the act of using a program like Flexbar to remove adapter sequences that could be found within your sequencing data.

Flexbar is installed the same way as all of the other programs we have used so far and once installed the program can be called using this code:

```
flexbar -r test_file.fastq -a adapter_sequence.fa -n 8
#-r identifies the input file (reads), -a identifies the adapter sequence,
-n gives the number of cores wanted
```

In our case the data is a paired end sequence and so has two files for each sequence. In this case the code is modified slightly to accommodate the extra file:

```
flexbar -r file_name.fastq -p file_name_2.fastq -a adapter_sequence.fa -ap
ON -n 8 -t output -x 13 --adapter-min-overlap 7 --min-read-length 25 --
threads 1 -z GZ
#-p tells the programs it's paired end sequencing, -ap tells the program to
remove paired end overlap, -x tells the program to remove a number of bases
from the start of the sequence,
#--adapter-min-overlap is the minimum overlap for removal (not paired end),
--min-read-length is the minimum read length that should be left after
trimming, --threads is the number of threads used, -z tells the program to
compress the output.
```

Flexbar has an interesting quirk where it will delete not just the adapter sequence but everything to the left of the adapter sequence provided. This is important to know because if your adapter sequence is on the right side of your sequence then it is best to find and use a different adapter trimming software because flexbar will delete your entire sequence.

Alignment:

After the adapter sequences are removed from your sequencing data, the next step is to align the sequence with a reference genome. The program we used is the Spliced Transcripts Alignment to a Reference (STAR) program. This program both creates a gene index using a reference genome which is needed to align a genetic sequence, and then maps the gene index with your data. This helps to see where there are mutations, deletions or frame shifts.

The program is installed the same way as all the other programs above and runs using these line of code:

```
STAR --runMode genomeGenerate --genomeDir Directory_name --genomeFastaFiles
file_name.fa --sjdbGTFfile annotation.gtf --genomeSAindexNbases 11
```

```
# --runMode tells the program what mode to run in with genomeGenerate being the mode which creates a genome index, --genomeDir tells the program where to output the index, --genomeFastaFiles details what fasta files to use in aid of the index generation,
#--sjdbGTFfile tells the program where the annotation file is, --genomeSAindexNbases allows the downscaling of this factor for smaller sequences like ours.


STAR --genomeDir Directory_name --readFilesIn file_name.fastq.gz file_name_2.fastq.gz --outFileNamePrefix Prefix/wanted --readFilesCommand zcat
# --genomeDir this time tells the program where the index was stored for use, --readFilesIn tells the program what files to map (two files for paired end sequencing data), --outFileNamePrefix allows the setting of a prefix for the output file name otherwise they will all come out the same,
# --readFilesCommand zcat allows the program to unzip compressed files and read them otherwise the program is unable to run.
```

The first set of code generates a new gene index in the provided directory. The second set of code takes this index along with the provided fastq files and an optional prefix (if you want your output files to be listed in a certain way) and outputs a .sam or .bam file with the sequence alignment.

Feature Counting:

Feature Counting is the process of counting the number of mapped and unmapped reads in your aligned files and summarizes the data into a simpler style. We used the program featureCounts to complete this process.

featureCounts is part of a package called subread. Because of this to install featureCounts we need to install subread:

```
conda install -c bioconda subread
```

The program then is run using the following code:

```
featureCounts -p -a annotation.gtf -g gene_id -o output.txt Path/to/Previous/Alignment.out.sam

#-p specifies the input is paired end, -a specifies the annotation file, -o specifies what you want the output file to be named, the final argument must be the path to the alignment file you wish to count.
```

An important thing of note is that the file produced by featureCounts cannot be opened in python and must be opened in the R language.

R language:

R is another open source coding language which was made mainly for statistical analysis but can also be used for data analysis which we did with featureCounts.

R is very good with table creation and manipulation which can be seen in the following example of how to open and manipulate the data in a .txt file that was outputted by R.

```
data = read.table("featureCounts_output.txt", header=T, stringsAsFactors = T)
data[,7:12]
```

A data.frame: 1378 × 6

...results.align_STAR.G1_rep1_Aligned.out.sam	...results.align_STAR.G1_rep2_Aligned.out.sam	...results.align_STAR.G1_rep3_Aligned.out.sam	...results.align_STAR.G2_rep1_Aligned.out.sam	...results.align_STAR.G2_rep2_Aligned.out.sam	...results.align_STAR.G2_rep3_Aligned.out.sam
<int>	<int>	<int>	<int>	<int>	<int>
0	0	0	0	0	0
0	0	0	0	0	0
0	0	0	0	0	0
0	0	0	0	0	0
0	0	0	0	0	0
0	0	0	0	0	0
0	0	0	0	0	0
0	0	0	0	0	0
0	0	0	0	0	0
0	0	0	0	0	1
0	0	0	0	2	0
0	0	0	0	1	0
0	0	0	0	1	0
0	0	0	0	0	0
0	0	0	0	0	0
0	0	0	0	0	0
0	0	0	0	2	0
0	0	0	0	1	0
0	0	0	0	2	0
0	0	0	0	0	0

A data.frame: 1378 × 6

...results.align_STAR.G1_rep1_Aligned.out.sam	...results.align_STAR.G1_rep2_Aligned.out.sam	...results.align_STAR.G1_rep3_Aligned.out.sam	...results.align_STAR.G2_rep1_Aligned.out.sam	...results.align_STAR.G2_rep2_Aligned.out.sam	...results.align_STAR.G2_rep3_Aligned.out.sam
<int>	<int>	<int>	<int>	<int>	<int>
0	0	0	0	0	0
0	0	0	0	1	0
0	0	0	0	1	1
0	0	0	0	0	0
0	0	0	1	2	0
0	0	0	0	0	0
0	0	1	7	5	10
0	0	0	0	2	0
0	0	0	0	0	0
0	0	0	2	1	2
⋮	⋮	⋮	⋮	⋮	⋮
143	139	135	989	656	940
0	0	0	0	0	0
1	0	0	0	0	0
0	0	0	0	0	0
3	3	5	12	2	11
3783	4906	4045	33528	20755	26500
19	31	20	143	80	125
0	0	1	0	0	0
235	284	225	2136	1413	1678
0	0	0	1	2	0
1	0	0	0	1	0
0	0	0	0	1	0
0	3	1	4	5	3

A data.frame: 1378 × 6

...results.align_STAR.G1_rep1_Aligned.out.sam	...results.align_STAR.G1_rep2_Aligned.out.sam	...results.align_STAR.G1_rep3_Aligned.out.sam	...results.align_STAR.G2_rep1_Aligned.out.sam	...results.align_STAR.G2_rep2_Aligned.out.sam	...results.align_STAR.G2_rep3_Aligned.out.sam
<int>	<int>	<int>	<int>	<int>	<int>
1	1	4	10	5	4
449	633	515	748	462	572
0	0	0	0	0	0
1	0	5	3	1	4
0	3	1	2	4	0
1	0	1	1	7	4
0	0	0	0	0	0
9	7	13	13	12	11
0	0	2	1	1	1
7	19	13	11	9	8
26	22	24	50	36	28
4	7	13	4	4	9
0	1	0	0	0	0
32	60	37	49	26	33
2	1	1	0	1	0
1	2	3	17	14	15
864	1139	985	1831	1226	1557

In [70]:

```
x = data[,7:12]
colnames(x) = 1:6
x
```

A data.frame: 1378 × 6

1	2	3	4	5	6
<int>	<int>	<int>	<int>	<int>	<int>
0	0	0	0	0	0

A data.frame: 1378 × 6

1	2	3	4	5	6
<int>	<int>	<int>	<int>	<int>	<int>
0	0	0	0	0	0
0	0	0	0	0	0
0	0	0	0	0	0
0	0	0	0	0	0
0	0	0	0	0	0
0	0	0	0	0	0
0	0	0	0	0	0
0	0	0	0	0	0
0	0	0	0	0	1
0	0	0	0	2	0
0	0	0	0	1	0
0	0	0	0	1	0
0	0	0	0	0	0
0	0	0	0	0	0
0	0	0	0	0	0
0	0	0	0	2	0
0	0	0	0	1	0
0	0	0	0	2	0
0	0	0	0	0	0
0	0	0	0	0	0
0	0	0	0	1	0
0	0	0	0	1	1
0	0	0	0	0	0
0	0	0	1	2	0
0	0	0	0	0	0

A data.frame: 1378 × 6

1	2	3	4	5	6
<int>	<int>	<int>	<int>	<int>	<int>
0	0	1	7	5	10
0	0	0	0	2	0
0	0	0	0	0	0
0	0	0	2	1	2
⋮	⋮	⋮	⋮	⋮	⋮
143	139	135	989	656	940
0	0	0	0	0	0
1	0	0	0	0	0
0	0	0	0	0	0
3	3	5	12	2	11
3783	4906	4045	33528	20755	26500
19	31	20	143	80	125
0	0	1	0	0	0
235	284	225	2136	1413	1678
0	0	0	1	2	0
1	0	0	0	1	0
0	0	0	0	1	0
0	3	1	4	5	3
1	1	4	10	5	4
449	633	515	748	462	572
0	0	0	0	0	0
1	0	5	3	1	4
0	3	1	2	4	0
1	0	1	1	7	4
0	0	0	0	0	0

A data.frame: 1378 × 6

1	2	3	4	5	6
<int>	<int>	<int>	<int>	<int>	<int>
9	7	13	13	12	11
0	0	2	1	1	1
7	19	13	11	9	8
26	22	24	50	36	28
4	7	13	4	4	9
0	1	0	0	0	0
32	60	37	49	26	33
2	1	1	0	1	0
1	2	3	17	14	15
864	1139	985	1831	1226	1557

In [72]:

```
means = rowMeans(x)
range(means)
```

1. 0
2. 17737.5

In [53]:

```
nrow(data)
```

1378

In [75]:

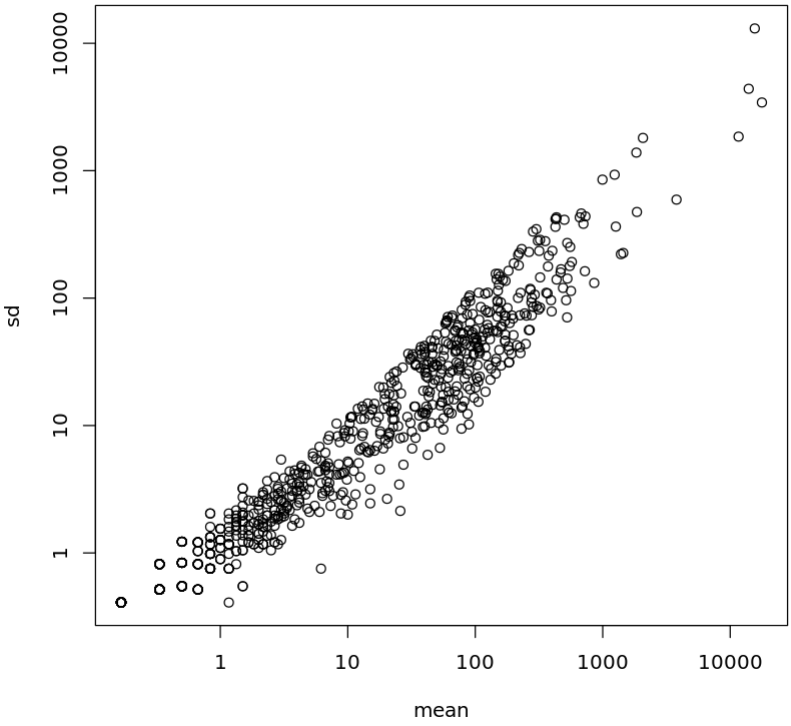
```
y = numeric(nrow(data))
for(i in 1:nrow(data)){
    y[i]=sd(data[i,7:12])
    }
```

In [83]:

```
selmeans = means > 0
sely = y > 0
x = means[selmeans&sely]
y2 = y[selmeans&sely]
```

In [86]:

```
plot(x,y2, log="xy", xlab="mean", ylab="sd")
```



Workflow Managers:

The last thing we needed to know was about workflow managers. Everything learned before was very important but it's awkward and difficult to type out each and every line of code every time you get a new file of sequencing data.

To help with this there are workflow managers, programs which allow the automation of multiple jobs. We can use workflow managers to be able to run everything with a single line of code. Meaning with one line of code a raw sequencing data file can be tested in quality control, trimmed, mapped and feature counted.

We used the workflow manager snakemake. Snakemake is based on the same idea as GNU Make, it follows a set of rules that you have written in a text file which states the input and output files for each process and the shell code that makes the process run. Because it is a set of rules this allows for the use of wildcards so that no matter when you receive the file, as long as it follows the naming convention snakemake can take the file and run it through all of the provided rules.

Snakemake also has the added benefit of not rerunning files. What that means is, for example, if you had a large number of files that all needed to be run through snakemake and once you finish a new file is sent to you to also be analyzed, if you run the snakemake program again it will not input any of the files that were already completed, reducing processing time and meaning there will be no duplicate output files.

Snakemake is installed via conda but it is not from any subcategory so the code looks like this:

```
conda install snakemake
```

An example of the rules system and how snakemake is structure is given below:

```
SAMPLES = ["G1_rep1","G1_rep2","G1_rep3","G2_rep1","G2_rep2","G2_rep3"]

rule all:

    input:

        expand("../results/fastqc_raw/{sample}_read1_fastqc.html",
sample=SAMPLES),

        expand("../results/fastqc_raw/{sample}_read2_fastqc.html",
sample=SAMPLES),

        expand("Trimmed/{sample}_trim_1.fastq.gz", sample=SAMPLES),

        expand("../results/STAR_MAP/{sample}_Aligned.out.sam",
sample=SAMPLES),

        #expand("../results/STAR_MAP/GenomeDir"),


expand("../results/Feature_Counts/{sample}_featureCounts_output.txt",
sample = SAMPLES)


rule Quality_Control:

    input:

        "../fastq_raw/{sample}_read1.fastq.gz",

        "../fastq_raw/{sample}_read2.fastq.gz"


    output:

       "../results/fastqc_raw/{sample}_read1_fastqc.html",

       "../results/fastqc_raw/{sample}_read2_fastqc.html"


    shell:

        "fastqc -o ../results/fastqc_raw {input}"


rule adapter_trimming:

    input:

        input_1 = "../fastq_raw/{sample}_read1.fastq.gz",

        input_2 = "../fastq_raw/{sample}_read2.fastq.gz"


    output:

        "Trimmed/{sample}_trim_1.fastq.gz"


    params:
```

```
        output = "Trimmed/{sample}_trim"


    shell:
        "flexbar -r {input.input_1} -p {input.input_2} -a
../illumina_adapter.fa -ap ON -t {params.output} -z GZ"


rule star_mapping_index:
    input:
        Annotation = "../genome/annotation.gtf",
        Genome = "../genome/genome.fa"


    output:
        directory = directory("../results/STAR_MAP/GenomeDir")


    shell:
        "STAR --genomeFastaFiles {input.Genome} --runMode genomeGenerate --
genomeDir {output.directory} --sjdbGTFfile {input.Annotation} --
genomeSAindexNbases 11"


rule star_mapping_map:
    input:
        FASTA_1 = "../fastq_raw/{sample}_read1.fastq.gz",
        FASTA_2 = "../fastq_raw/{sample}_read1.fastq.gz"


    output:
        "../results/STAR_MAP/{sample}_Aligned.out.sam"


    params:
        output = "../results/STAR_MAP/{sample}_"


    shell:
        "STAR --genomeDir ../results/STAR_MAP/GenomeDir --readFilesIn
{input.FASTA_1} {input.FASTA_2} --readFilesCommand zcat --outFileNamePrefix
{params.output}"


rule feature_counts:
    input:
        Annotation = "../genome/annotation.gtf",
```

```
        STAR = "../results/STAR_MAP/{sample}_Aligned.out.sam"


    output:
        "../results/Feature_Counts/{sample}_featureCounts_output.txt"


    params:
        prefix = "../results/Feature_Counts/{sample}_featureCounts_"


    shell:
        "featureCounts -p -O -a {input.Annotation} -o
{params.prefix}output.txt {input.STAR}"
```

I think it is important to note in this protocol that my code for STAR mapping index didn't work on my machine but for some reason worked on two classmates machines with no modification. My only assumption is that there is either a pathing issue or an issue with my virtual machine as each time I ran the code it would delete the directory at the end of my path instead of adding to it.