CLUSTard

An automated pipeline for metagenomic clustering using read abundance over time

Annabel Cansdale

Master of Science (by Research)

Biology Department, University of York

January 2020

Abstract

Metagenomics, the study of genetic material generated from culture-independent shotgun sequencing of environmental samples, facilitates the investigation of environmental communities as a whole. However, in order to determine biological context from a metagenomic assembly it is necessary to group sequences to allow for the study of the community dynamics and individual organisms. This process, known as metagenomic binning, is accomplished by utilising an aspect of the sequence's composition. There is little metagenomic binning software available that utilises the change in a community over time in order to cluster metagenomes. Here, I present `CLUSTard,` an automated pipeline that accomplishes metagenomic binning by utilising sequence abundance values over time, this pipeline clusters large datasets efficiently and requires minimal user input or installation. `CLUSTard` enabled the resolution of a previously undefined metagenomic dataset. The pipeline allowed for reproducible analysis vastly reducing the time and effort required. I found that the most important factor impacting the `CLUSTard`'s success of clustering was the quality of the input assembly, with a highly contiguous Nanopore assembly polished by Illumina sequences producing the best clustering result. The results demonstrate that the use of abundance information enables efficient and accurate clustering and also highlights the importance of a reproducible analysis pipeline. I anticipate this pipeline to be beneficial for those who want to produce metagenomic clusters using time-series data and to provide a starting point for further analysis.

Table of Contents

List of tables

List of figures

Acknowledgements

I would like to thank my supervisor, Prof. James Chong, for his support and guidance throughout this project and for dealing with my thousands of questions. With thanks to Dr John Davey, Dr Katherine Newling, Dr Sally James and Dr Peter Ashton from the Bioinformatics facility at the University of York for the advice, support and biscuits.

With thanks to my Dad who read all of this despite “not understanding a word” and my Mum and Reggie for the food and company. And with thanks to Laura, Heidi, Ali, Emma, and Zainab for keeping me sane, and Kim, James R, and the rest of the Chong lab for doing the opposite.

This project was undertaken on the Viking Cluster, which is a high-performance compute facility provided by the University of York. I am grateful for computational support from the University of York High Performance Computing service, Viking and the Research Computing team.

Declaration

I declare that this thesis is a presentation of original work and I am the sole author. This work has not previously been presented for an award at this, or any other, University. All sources are acknowledged as References.

The `bwa_Snakefile` and `para_Snakefile` parts of the `CLUSTard` pipeline are based on earlier work by Prof. James Chong. DNA extraction of the NAB dataset was done by Dr Anna Alessi. The NAB Nanopore assembly and polishing were undertaken by the Bioinformatics Core Facility at the University of York.

Introduction

Metagenomics, the study of genetic material generated from culture-independent shotgun sequencing of environmental samples, proves an involved and time-consuming task when analysing data from complex relatively unknown communities. Contiguous sequences (contigs) from a metagenomic assembly can be grouped together into taxonomic bins based on shared characteristics and then must undergo many further analysis steps in order to validate these groups and determine community composition.

In this study due to existing datasets, it was first deemed necessary to develop a method of metagenome binning which can deal with a mix of sequencing technologies and utilise the information present in a time series dataset. Then to allow for further analysis, chain this method of binning to other analysis steps in a reproducible workflow. Finally, the optimal operational parameters for this method of binning needed to be determined and the binning results validated.

Here a specialised binning strategy for time-series metagenomic datasets was developed and then a metagenomic analysis pipeline developed around it. This pipeline, known as `CLUSTard`, enabled reproducible and fast metagenomic binning and downstream analysis, allowed for the optimal operational parameters of the binning strategy to be determined and also ultimately enabled the straightforward analysis of a previously un-resolved large metagenomic dataset.

Microbial Communities

The definition of a new bacterial species relies on the growth of the bacteria in pure culture (Chan *et al*. 2012). Due to the challenges faced when attempting to culture bacteria, (Stewart 2012) a great deal of the overall microbial diversity remains uncultured and therefore many species remain undefined (Bernard *et al*. 2018). It is possible that these uncultured organisms play important roles in specific microbial communities or for drug discovery (Stewart 2012). This so-called “microbial dark matter” has become somewhat less dark following the emergence of targeted 16S rRNA gene sequencing, where the DNA sequence could be obtained from an environmental sample by utilising PCR amplification, and Next-Generation sequencing (Stewart 2012). 16S rRNA genes are well conserved, occurring in all cells and organelles (Pace 2009) and contain hypervariable regions which exhibit large sequence diversity among bacteria (Shah *et al*. 2011).

With the arrival of 16S sequencing the ‘candidate’ phyla emerged, which contains organisms with no pure cultures to represent them (Rappé and Giovannoni 2003). Members of this Candidate Phyla Radiation (CPR) which includes over 70 phyla, exist in many different environments and contribute a significant portion of the diversity amongst them (Danczak *et al*. 2017). One such microbe, a member of the Lokiarchaeota phylum, *Candidatus* Prometheoarchaeum syntrophicum strain MK-D1 was recently isolated and cultured (Imachi *et al*. 2019). The Lokiarchaeota phylum was first identified by Spang *et al*. (2015) through a combination of 16S rRNA and metagenomic sequencing.

Whilst the use of 16S sequencing has enabled the resolution of many otherwise unclassified organisms it is not without its shortcomings. As rRNA sequences are so well conserved they cannot be reliably used to determine closely related species, such as *Escherichia coli* and *Shigella dysenteriae* (Pace 2009). It has also been found that phylogenetic analysis based on only the 16S rRNA gene proved unreliable when compared to a phylogenetic tree built using a core genome (Chan *et al*. 2012) and 16S rRNA sequencing is also impacted by PCR biases as it relies on targeted primers to amplify the marker gene (Shah *et al*. 2011; Hillmann *et al*. 2018).

An alternative to 16S rRNA sequencing that enables the resolution of complex communities containing uncultured microbes is the field of metagenomics. Metagenomics is the study of genetic material generated from culture-independent shotgun sequencing of environmental samples (Vollmers *et al*. 2017). While 16S rRNA analysis is cheaper both in cost and computational time (Hillmann *et al*. 2018), the whole-genome sequencing (WGS) of metagenomes provides the opportunity to capture the community dynamics at scale without a biased amplification step (Shah *et al*. 2011) and also to capture more of the overall genetic material. As sequencing technology has become cheaper over recent years the field of metagenomics has grown and enabled the simple resolution of previously uncharacterised communities. For example, the production of 913 draft genomes, the majority of which were previously unsequenced, from metagenomic sequencing of the cow rumen (Stewart *et al*. 2018) and over 150,000 genomes reconstructed from human microbiome metagenomes by Pasolli *et al*. (2019).

Whilst capturing all sequences from all members of a microbial community is not realistic (Zaheer *et al*. 2018), sequencing depth - i.e. the number of different reads that cover a base in the sequences - is important when it comes to metagenomics. Deeper metagenomic sequencing is potentially able to determine novel gene content that is not possible to obtain from shallow sequencing (Hillmann *et al*. 2018). Deep metagenomic sequencing is often too expensive for large-scale projects, whilst it can provide strain-level resolution, sequencing at this depth is not necessary to acquire more reliable species profiles than 16S sequencing (Hillmann *et al*. 2018). Along with the too shallow sequencing depth, often not deep enough to capture rare species in complex populations (Shah *et al*. 2011) the field of metagenomics also faces biases. Despite the lack of amplification bias with shotgun sequencing, biases can be introduced by the DNA extraction or sequencing method (Morgan *et al*. 2010).

Anaerobic Digestion

Anaerobic digestion (AD) is a process that produces biogas from the anaerobic decomposition of organic waste, an important aspect towards the production of green energy (Treu *et al*. 2016). Whilst the operational conditions of anaerobic digesters have been finely tuned to maximise efficiency, the microbial community less so (Peces *et al*. 2018). Microbes play an important part in the creation of biogas during anaerobic digestion as different members of the microbial community perform different steps in the biologically mediated process (Campanaro *et al*. 2019). By harnessing the microbial potential, the production of biogas could become more efficient.

Many studies into anaerobic digestion microbial communities are based on 16S rRNA sequencing (Kirkegaard *et al*. 2017; McIlroy *et al*. 2017; Peces *et al*. 2018). The lack of genome datasets means that a lot of information present in the community is missing, including the metabolic information which is necessary to assign roles to parts of the AD process (Peces *et al*. 2018). Campanaro *et al*. (2019) produced >1,500 metagenome-assembled genomes (MAGs) of varying quality from publicly available metagenomic anaerobic digestion studies. They found that few taxa were shared between different AD systems - with most systems developing specialised microbial communities.

Assembly

Genome assembly is the process of joining the short fragments produced by DNA sequencing (sequencing reads) into long contiguous sequences known as contigs (Paszkiewicz and Studholme 2010).

Whilst Illumina sequencing revolutionised genome assembly due to the increased throughput, the length of the short-reads produced by Illumina are insufficient in length to resolve repeat regions present in many organisms (Treangen and Salzberg 2011). This has led to many studies releasing unfinished genome assemblies with low quality and low contiguity (Alkan *et al*. 2011). Illumina sequencing has an average read length of ~50-600bp (Weirather *et al*. 2017) which is much shorter than Oxford Nanopore Technologies sequencing (Nanopore), with an average read length of 1-100Kbp (Giordano *et al*. 2017). The longer read length of Nanopore sequencing makes genome assembly easier as it can provide clarity in areas of long repeats that were previously unpassable with Illumina sequencing (Koren and Phillippy 2015). This has brought definition to areas of genomes that have important functional roles, and which were previously challenging to sequence (Schmid *et al*. 2018), with Nanopore sequencing enabling the structure of an antibiotic resistance island in *Salmonella* Typhi to be resolved (Ashton *et al*. 2015).

Whilst the long-reads of Nanopore sequencing facilitates the resolution of repeat regions, the higher error rates of Nanopore sequencing, ~10-15% compared to ~1% for Illumina sequencing (Sović *et al*. 2016), proves challenging when it comes to genome assembly. Popular genome assembly tools for Illumina sequencing - such as `SPAdes` (Bankevich *et al*. 2012) - rely on algorithms which fail with an error rate above 10% (Lin *et al*. 2016) and are therefore unsuitable for Nanopore sequencing assembly. As assembly strategy varies depending on the sequencing technology used, new assemblers have been built which perform better with the error-prone long-reads of Nanopore, such as `Canu` and `Flye` (Koren *et al*. 2017; Kolmogorov *et al*. 2019). *De novo* Long-read Nanopore assembly is still far from perfect with sequencing errors often confounding the outcome (Fu *et al*. 2019) and therefore often requires polishing with Illumina data (Giordano *et al*. 2017). This method combines the ability of Nanopore long-reads to traverse regions of repeat with the more accurate Illumina sequencing (Weirather *et al*. 2017).

Metagenomic sequencing adds an extra level of complexity to the assembly of sequencing reads, due to the presence of multiple organisms in a metagenomics sequencing sample and the depth of coverage required to resolve many of them (Ayling *et al*. 2019). Most metagenomic datasets are large and diverse which presents a unique computational challenge when it comes to assembling these sequencing reads into contigs (Ayling *et al*. 2019). To produce high quality *de novo* genome assemblies from metagenomes, sequencing reads need to span both intragenomic and intergenomic repeats to prevent a highly fragmented assembly, the longer read lengths of Nanopore sequencing prove advantageous in this (Somerville *et al*. 2019). The higher error rate makes the assembly of Nanopore sequencing more challenging, with `Canu` taking twice as long to assemble a Nanopore dataset compared to a comparable PacBio (another method of long-read sequencing) dataset (Jain *et al*. 2018). To date no published tools are dedicated solely to the assembly of Nanopore long-read metagenomic data, however both `Canu` and `metaFlye` have been shown to deal well with Nanopore metagenomic datasets (Latorre-Pérez *et al*. 2019). Whilst Nanopore sequences are often polished using the raw signal to improve the accuracy, the large metagenomic datasets make this more challenging (Somerville *et al*. 2019). Somerville *et al*. (2019) assembled all dominant species of a small metagenome using a combination of Nanopore, Illumina and PacBio despite the challenge of *de novo* metagenome assembly, noting the importance of long reads and also the importance of polishing with short reads to achieve contiguous and highly accurate assemblies.

Clustering

Whilst it is possible to generate complete genomes from metagenomic assembly, many factors including community size and complexity prevent this generating highly fragmented assemblies (Alneberg *et al*. 2014). Because of this, it is necessary to group contigs into clusters based on shared characteristics in order to reconstruct genomes (Sieber *et al*. 2018). This process is known as binning and can be achieved using multiple methods. One such method is by using an element of sequence composition such as GC content or tetra-nucleotide identity. For example, the popular program `MaxBin` which utilises tetranucleotide frequencies along with one-sample coverage information to automatically bin contigs (Wu *et al*. 2014). Many microbial species have vast differences in their GC content (Reichenberger *et al*. 2015) which makes this a feasible method of clustering. Another common clustering method is the use of coverage information as different organisms are present in different quantities in an environmental sample (Alneberg *et al*. 2014) this can therefore be used to separate contigs. Two popular binning programs, `CONCOCT` and `MetaBAT`, combine coverage information and sequencing composition to reconstruct genomes (Alneberg *et al*. 2014; Kang *et al*. 2015). It has been noted that binning based on differential coverage information is more effective than binning based only on composition (Sieber *et al*. 2018).

The existence of the multitude of different metagenomic binning software utilising different binning techniques highlights the fact that metagenomic binning is far from perfect. The program `AMBER` (Meyer *et al*. 2018) exists to enable evaluation and comparison of different binning software. Meyer *et al*. (2018) found that `MetaBAT` recovered the most high-quality bins when using the datasets provided by the CAMI challenge (Sczyrba *et al*. 2017). However, as highlighted by Sieber *et al*. (2018), no single binning software produces optimal results on every metagenome consistently. The program `DASTool` (Sieber *et al*. 2018) provides a potential solution to this, enabling the combination of results from multiple different binning software to produce the most high-quality bins (Meyer *et al*. 2018). However, this requires multiple time consuming and computationally intensive programs to be run on one dataset and may not provide much overall improvement.

Benchmarking and comparison of metagenomic binning has long been focussed on the clustering of short-read assemblies and as outlined earlier, long-read metagenomic assembly faces separate issues to short-read metagenomic assembly. The benchmarking of metagenomic software has been achieved during the Critical Assessment of Metagenome Interpretation (CAMI) challenge (Sczyrba *et al*. 2017) which was based on short read metagenomic datasets and assemblies. Although a second challenge including long-read sequencing is currently in progress, this had not been completed at the time of writing.

Sequencing Classification

After the clustering of metagenomic bins it is necessary for them to be taxonomically classified in order to determine what organisms are present in the community. As this information is usually not known at the time of sequencing, sequencing classifiers, such as `NCBI BLAST` (Altschul *et al*. 1990), perhaps the most well-known method, classifies a sequence by finding other close aligning sequences from a large database. Whilst this method proves effective, it is extremely time-consuming and CPU intensive on large metagenomic datasets (Wood and Salzberg 2014). For this reason, much faster metagenomic classifiers requiring less computational power, such as `Kraken` (Wood and Salzberg 2014) or `Centrifuge` (Kim *et al*. 2016), can be used to accomplish this task. `Kraken` works by creating a Lowest Common Ancestor database of *k*-mers (i.e. a sequence of length *k*, the default length is 31bp). Any *k*-mers identified in the query sequence enables a path in the classification tree to be traversed, ultimately identifying the lowest common ancestor (LCA). The small database size and efficient classification strategy of `Kraken` allows rapid classification of large metagenomic datasets (Wood and Salzberg 2014). The program `Centrifuge` takes a similar strategy also resulting in a small database and fast classification of sequences (Kim *et al*. 2016). Both `Kraken` and `Centrifuge` however, were developed to determine species abundance information of raw sequencing reads and not assembled metagenomic bins, despite them being commonly used in this way (Wood and Salzberg 2014; Kim *et al*. 2016; Nicholls *et al*. 2019).

Most classification software is reliant on a database of reference sequences. Nasko *et al*. (2018) found that taxonomic classification by `Kraken` is strongly influenced by the database composition, with unknown species - i.e. those with no representation in the database - resulting in an analysis bottle-neck. They also highlighted the problem of contamination and misclassification in public databases which can cause errors due to the inconsistency of the database which underscores the fact that metagenomic sequence classification is still far from perfect.

Validation

Once produced, the metagenome bins must be validated to determine if the metagenome-assembled genomes are biologically real. If reference genomes are known for a community, validation becomes an easier task. For example, with `metaQUAST` which evaluates metagenomic assemblies using user-defined reference sequences (Mikheenko *et al*. 2016). However, given the nature of metagenomic sequencing the members of a community are often unknown. Whilst metaQUAST can identify related species through 16S rRNA sequences this only works for species with previously classified neighbours that are in the database (Mikheenko *et al*. 2016). The existence of the so-called microbial dark matter makes this challenging and therefore other methods of validation that are not based on reference sequences are necessary.

One such method used to validate the assembly is determining the number of genes present in an assembly or the coding density, i.e. the number of genes per 1Mbp of sequence, as this indicates the completeness of an assembly (Olson *et al*. 2017). Prokka is a popular software that performs rapid genome annotation by combining data from multiple different sources in order to predict and identify genes in a prokaryotic genome (Seemann 2014). This can then be extrapolated to determine the number of genes present in an assembly.

Gene presence or absence can also be used as a validation method. CheckM (Parks *et al*., 2015) is a popular metagenomic tool that utilises this method to validate the results of metagenomic binning. CheckM utilises 48 lineage-specific marker genes to place genome bins within a reference tree and to determine the completeness and contamination values of the genome bins given. However, as noted by Parks *et al*. (2015), eukaryotic and phage genomes and also plasmids will be reported as highly incomplete as the CheckM marker genes are only suitable to assess bacterial or archaeal genomes, and therefore must be analysed for completeness by another software.

Standards for reporting bacterial and archaeal genome sequences exist and for metagenomic binning the Minimum Information about a Metagenome-Assembled Genome (MIMAG) is relevant (Bowers *et al*. 2017). The difficulty of verifying assembly quality when there is a lack of ‘ground truth’ is highlighted by Bowers *et al*. (2017). It is therefore recommended to report basic assembly statistics that do not rely on reference genomes, including N50 length, total assembly size and maximum contig length. Completeness and contamination values calculated by a program such as CheckM are also an important metric with metagenome-assembled genomes (MAGs) >90% complete, <5% contaminated and that encode all rRNA genes and >18/20 tRNA genes which qualify them as “high-quality draft MAGs” using the Bowers *et al*. (2017) categorisation. The latter rRNA and tRNA completeness can be evaluated by a program like `Prokka`. Due to the small genomes of symbiotic bacteria discovered no minimum assembly size is suggested (Bowers *et al*. 2017).

Any errors in sequencing or assembly would impact both the coding density of a genome and the completeness and contamination values, as any errors could alter the gene sequence and lead to it not being recognised or would result in premature stop codons or frameshifts (Watson and Warr 2019). Watson and Warr (2019) highlights this issue especially when dealing with Nanopore sequencing, which has a higher occurrence of insertion or deletion errors when compared to Illumina sequencing.

Workflows

Because of the number of different tools available to analyse metagenomes, the speed at which they are updated and the increasing popularity of metagenomic sequencing it is necessary to develop methods of analysis that allow both comparison between metagenomic datasets and to prove that the results are themselves reproducible (Tamames and Puente-Sánchez 2018).

One such method to establish reproducible research during the data analysis stage is the use of workflows and package managers which allow easy distribution of the analysis steps and enable any variation between workstations e.g. operating system or different software versions to be dealt with (Visconti *et al*. 2018).

`Conda` (conda.io) is one such package manager that is popular in the biological sciences due in part to the channel `Bioconda` (bioconda.github.io) which carries many popular bioinformatics software. `Conda` enables the easy installation of packages and dependencies without administrative privileges which is an advantage when performing analysis within a high-performance computing (HPC) cluster environment (Grüning *et al*. 2018). `Docker`

(docker.com) is another platform that enables containerisation. Workflow managers are another useful addition to the bioinformaticians arsenal, as not only do workflow pipelines provide reproducible results but they also enable the easy analysis of alternate datasets.

The production of pipelines using `Perl` or the `UNIX` shell is common in bioinformatics. However, these do not allow the pipeline to be restarted from previous checkpoints or individual containerisation of different steps in the pipeline, something which modern workflow managers do allow (Leipzig, 2017). `Snakemake` (Köster and Rahmann 2012) and `Nextflow` (Di Tommaso *et al*. 2017) are two such modern workflow managers.

As outlined by Leipzig (2017), the choice of a workflow manager is primarily down to both user preference and which is most appropriate for the specific use case. `Snakemake` is built specifically for bioinformatics research and allows the integration of `Python` code directly into the pipeline and the use of different `Conda` environments for each step of the pipeline. This makes it easier to run different software that may require different dependencies (Köster and Rahmann 2012). Because of this `Snakemake` is popular with many bioinformatics tools utilising it for analysis such as `VIPER` (Cornwell *et al*. 2018).

CLUSTard

As outlined here, metagenomic assembly and analysis is a complex and time-consuming process made even more challenging by the large datasets. Metagenomic assembly and analysis can be achieved using many different strategies with a lot of third-party software available for each step. It is sensible to undertake any bioinformatics analysis with the best possible software for the specific user case. In the case of the datasets we produce as a group, which are large time-series datasets with a variety of different sequencing technologies, an assembly strategy that utilises the combination of long and short read sequencing and a binning strategy that utilises this and the time-series information would be most appropriate. Whilst binning tools that make use of time-series information are available as outlined earlier, they are far from perfect and it is not known how well they deal with a combination of sequencing technology.

To enable both the results themselves to be reproduced and to simplify this analysis on all appropriate datasets, this analysis should be made easily reproducible and portable by employing an analysis pipeline. Many workflow pipelines to enable reproducible metagenomic analysis have previously been released such as `Anvi’o` (Murat Eren *et al*. 2015), `SqueezeMeta` (Tamames and Puente-Sánchez 2018) and YAMP (Visconti *et al*. 2018). However, these pipelines are not appropriate for large scale metagenomic datasets due to in part to memory constraints and available pipelines either struggle with the large datasets, do not allow for the combination of sequencing technologies and do not allow the user much, if any, flexibility in programs or parameters used. The lack of available pipelines to correctly and efficiently analyse these large metagenomic datasets means that each dataset requires user intervention at each step of the analysis process.

To overcome these challenges, I first sought to establish a method of clustering that makes use of the time series information and then to integrate this method of clustering into a pipeline of further metagenomic analysis steps. Following this to both evaluate this method of clustering and also determine the parameters and assembly method which produce the best clustering results. Finally, I sought to use this pipeline to analyse a previously un-definable metagenomic dataset.

To bin contigs from a metagenomic assembly utilising time-series information, raw reads from each time point are mapped onto contigs to determine the abundance values of the contigs over time. Then using `Python` scripts, contigs with a similar abundance pattern, determined by using pairwise Pearson's correlation coefficient analysis, are clustered together if the correlation value is above a given threshold. This was all chained together using a `Snakemake` workflow and further analysis steps were added to determine the validity of the clusters produced and to provide biological context. This workflow enabled the pipeline to be re-run using different datasets and under different parameters in order to evaluate the binning method and also how well the downstream tools perform given these different situations. Once optimal conditions were determined, the `CLUSTard` pipeline enabled the biological classification of a metagenomic dataset.

Here, data from an 18-month metagenomic study of four parallel industrial-scale anaerobic digesters were investigated. The use of the `CLUSTard` pipeline enabled the community composition of this dataset to be determined which was previously un-resolvable due to its size. This provides an avenue for future research to link the community composition with metabolic function and process operational conditions to produce important information about how to maximise biogas production.

Methods

`CLUSTard` Pipeline

The CLUSTard pipeline was built using the workflow manager `Snakemake` (Köster and Rahmann 2012). Figure 1 shows the Snakemake pipeline diagram, the code is available in the Appendix and on `GitHub` (github.com/ac1513/CLUSTard).

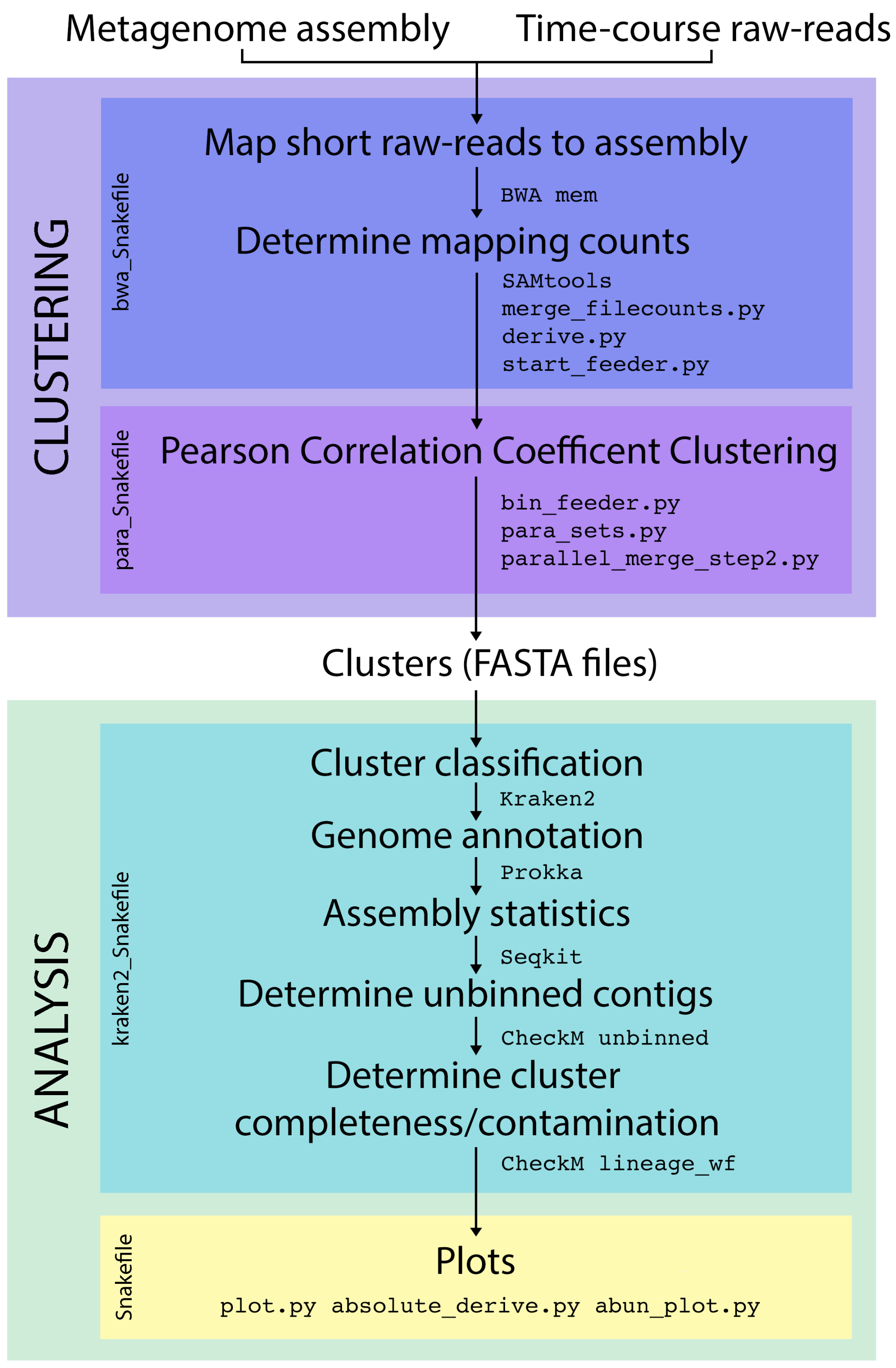


Figure 1: Workflow of the `CLUSTard` pipeline. Showing the two main elements of the pipeline: clustering and downstream analysis (outside boxes). The `Snakemake` files used in the pipeline in the inside boxes. With the programs or `Python` scripts used at each step shown in `this font`.

Clustering

The raw short-reads for each timepoint were mapped against the input assembly using `BWA mem (v.0.7.17)` (Li 2013). Mapped reads are then converted to counts using `SAMtools (v.1.9)` (Li *et al*. 2009) which determines the number of short reads at each timepoint mapping to each contig. A `Python3` script merges the counts for each timepoint into a single file (`merge_filecounts.py`), before `derive.py` normalises count values to the first timepoint providing an indication of relative abundance and calculates coverage values for each contig. This also filters out any contigs below the contig length threshold (suggested value is 1,000bp for long read assemblies). The next `Python3` script (`start_feeder.py`) calculates mean and standard deviations for the abundance values. The number of mapped reads at each timepoint for each contig were used to calculate the sample Pearson correlation coefficient by implementing Equation 1 in pairwise comparisons in the `Python3` script `bin_finder.py.`

$$r_{xy} = \frac{\sum_{i=1}^{n}\left(x_i - \bar{x}\right)\left(y_i - \bar{y}\right)}{\sqrt{\sum_{i=1}^{n}\left(x_i - \bar{x}\right)^2}\sqrt{\sum_{i=1}^{n}\left(y_i - \bar{y}\right)^2}}$$

Equation 1: The pairwise Pearson's correlation coefficient calculation, where n is the sample size, x_i and y_i are the individual sample points indexed with i, and $\bar{x}$ and $\bar{y}$ the sample mean.

Here a cut-off value of the sample Pearson correlation coefficient can be chosen for clustering. Prior to this step the count files are split into 10,000 contig long blocks, using the `Unix split` function to facilitate the parallelisation of the pairwise comparisons in order to speed up the correlation process. Here, the `Snakefile` in use changed to `para_Snakefile` as the number of files produced by the split would not be consistent between different `CLUSTard` runs. The separating of the pipeline was necessary as the *`dynamic`* function in `Snakemake,` which allows `Snakemake` to run rules that would produce an unknown number of output files, did not work in a computing cluster environment. The files produced in the parallel step are then merged using `para_sets.py`, `parallel_merge_step2.py` and `step2.py`.

Once contigs have been clustered a separate FASTA file containing the contigs for each cluster is produced using the script `file_parser.py`, the name of the largest contig in the cluster is subsequently used as the cluster identity for the plots, `csv` file and `FASTA` file produced.

Cluster Analysis

Once the cluster FASTA files have been produced they are passed into a separate `Snakefile` to undergo downstream analysis.

First, the clusters are each run through `Kraken` (v 2.0.7) (Wood *et al*. 2019) using a user-defined index for taxonomically classification. The top result at the user-defined taxonomic rank (e.g. genus) for each cluster is determined using the Unix command:

```
find -name '{JOBID}*_report_kraken.out' -type f -printf '\n%p\t' -exec
sh -c 'echo {} | sort -k1nr {} | grep -P "\t{params.level}\t" | head -n1
' \; > {JOBID}_{params.level}_top_kraken.out
```

where `{JOBID}` is the user defined `CLUSTard` prefix and `{params.level}` is the user defined taxonomic rank.

Each cluster is then run through `Prokka (v.1.11)` (Seemann 2014) for genome annotation. `Seqkit (v.0.10.1)` (Shen *et al.* 2016) is also run on each cluster with the parameter `-a` and `-T` to output tab-separated assembly statistics for each cluster. Finally, `CheckM (v. 1.0.13)` (Parks *et al.* 2015) is run on each cluster first with the parameter *`unbinned`* to determine the percentage of assembly contigs that have not been clustered. Then with the parameter *`Lineage_wf`*, to estimate genome completeness and contamination - important statistics for metagenomics. Whilst these steps (other than `Prokka`) are all utilised in the output of `CLUSTard` they also provide useful launchpads and information for further analysis by the user as is highlighted later in “Biological Analysis”.

The final step of the `CLUSTard` pipeline is to produce multiple plots. Each of the plots have user-defined parameters to allow for the simple production of meaningful plots. An overview plot is produced for each cluster (`plot.py`). These plots include information on size, completeness, contamination, and Kraken2 classification. This information is also saved to a csv file. Next, a plot is produced showing the size of contigs successfully clustered (`bin_plot.py`). Finally, the `Python3` script `abun_plot.py` produces a relative or absolute abundance profile of the clusters produced using the un-normalised short-read mapping counts and utilising the top `Kraken2` classification information at the user specified taxonomic rank. Here, the user can also specify if this plot should include all classification or only the 19 most abundant and 'other' as otherwise this plot would end up complex for large metagenomes.

How CLUSTard is run

The use of Snakemake as a workflow manager allows CLUSTard to be run with minimal user-input and despite entirely being run on the Unix command line requires minimal knowledge of the command line, minimal installation and no root access. If Snakemake is not already installed on the user's system, it should be installed following the steps in the Snakemake documentation.

First the code must be copied to the users working directory. As the code is available on the online repository GitHub all that is required is the command:

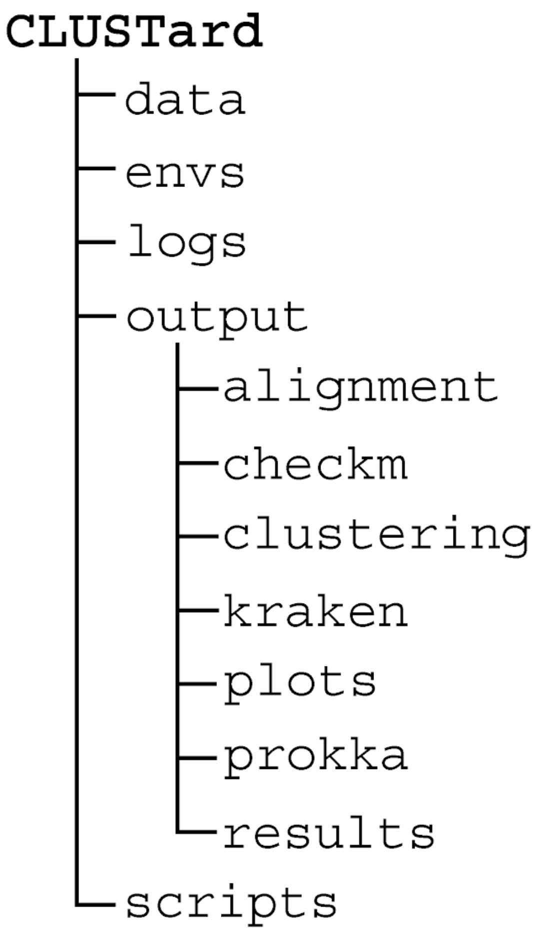


Figure 2: The directory organisation produced after a run of the CLUSTard pipeline has been completed.

```
git clone https://github.com/ac1513/CLUSTard.git
```

This will create a directory named *CLUSTard* and all the analysis is done within this directory. The data should be added to a subdirectory within the CLUSTard directory called *data*. This should contain the raw read samples and also the metagenome assembly. The sample order and grouping to be used for CLUSTard analysis is defined by the user in the *samples.tsv* file. The location of the input data is then defined by the user in the *config.yaml* file. Other parameters can also be defined by the user in this file including the Pearson's correlation coefficient threshold, options for the output plot including the taxonomic level of interest. Once the input files have been edited so that Snakemake is pointed to the correct dataset to be used Snakemake can then be run. If running on a computing cluster this would preferably be within Screen on Unix to allow Snakemake to continue sending jobs to the cluster without keeping a terminal window open. Due to limitations of Snakemake within a computing cluster it was necessary to separate the pipeline into three separate Snakefiles (bwa_Snakefile; para_Snakefile and kraken2_Snakefile) with a wrapper Snakefile which enables all steps to still be run consecutively from one command.

On a computing cluster CLUSTard can be run using the command

```
snakemake --use-conda --cluster "sbatch -t 48:00:00 --cpus-per-
task={threads}" -j 1000.
```

this runs the Snakemake pipeline using the specified Conda environments and runs each job on the cluster with a maximum time limit of 48hrs and the number of threads specified in the Snakefile(s), -j 1000 allows a maximum of 1000 cluster jobs to be sent to the queue at once. Once running Snakemake will catalogue its progress on the command line.

Directory Organisation

The output of CLUSTard is split into multiple directories. Figure 2 shows the outcome from a CLUSTard run from the CLUSTard directory which is produced when the git repository is cloned. *data* is a user generated directory containing the input data for the CLUSTard run, *envs* contains the information for Snakemake about the Conda environments and *scripts* contains the Snakefiles and Python3 scripts necessary to run the pipeline. *Logs* is a directory generated during a CLUSTard run, this is where all program and cluster logs are saved - which is useful to investigate if debugging.

The *output* directory is also automatically created during a CLUSTard run and contains the following subdirectories. The *alignment* subdirectory contains the alignment SAM files from BWA, *clustering* contains the intermediate files generated by CLUSTard and *results* contains the cluster

`FASTA` files and the corresponding csv file of abundance values that is used for the plot. The *`checkm`*, *`kraken`* and *`prokka`* directories contain the output created after the clusters are run through those programs. Finally, all plots produced by `CLUSTard` are saved in the *`plots`* directory.

Data Acquisition

The CLUSTard pipeline was developed and tested on a metagenomic dataset with both long Nanopore reads and a time-series of raw Illumina short reads.

Sampling and Sequencing

Treated sewage sludge samples were taken from four industrial-scale anaerobic digesters over a period of eight months. In total sampling from the digesters occurred 19 times, each approximately two weeks apart and the input feed was sampled five times towards the end of the sampling period (see Appendix Figure 1 for date breakdown). DNA from the samples was extracted using the DNeasy PowerSoil Kit. DNA from all 19 timepoints for all four digesters and the five timepoints from the feed underwent library preparation and sequencing on an Illumina HiSeq 3000 at either Novogene or Leeds Genomics.

DNA samples from timepoints 15 and 17 for all four digesters and feed were pooled and underwent standard Nanopore ligation library preparation to be sequenced on a PromethION at the University of York Genomics facility.

Assembly

The nanopore PromethION sequences were assembled and polished by the Bioinformatics Core at the University of York. First using `Canu (v.1.8)` (Koren *et al.* 2017) on `Google Cloud` to assemble the raw-reads. The assembly was then polished using the Nanopore `FAST5s` by `Nanopolish (v.0.11.0)` (Simpson *et al.* 2017), then polished using pooled Illumina raw-reads from timepoints 15 and 17, first by `Pilon (v.1.23)` (Walker *et al.* 2014) and finally polished by `Racon (v.1.3.3)` (Vaser *et al.* 2017) over four iterations.

The Illumina short reads were first adaptor trimmed using cutadapt (v2.3) (Martin 2011) with the adaptor sequence AGATCGGAAGAG. The Illumina-only assembly was then assembled using all of the trimmed short-read sequencing data by MEGAhit (v 1.1.3) (Li *et al*. 2015) using the parameter *--presets meta-Large* and utilising the paired-end information. MEGAhit was used over SPAdes (v.3.13.1) (Bankevich *et al*. 2012) as it was not possible to run SPAdes on this dataset within the computing cluster time and memory constraints.

SeqKit (v.0.10.1) was then used to determine assembly statistics with the parameter *stats -a*. BWA mem (v.0.7.17) was used to map the raw-short reads back onto all assemblies with the parameter *mem* and the percentage sequence mapping was calculated using SAMtools (v.1.9) flagstat. Prokka (v.1.11) was used to determine the presence of complete or partial 16S sequences.

Cluster Analysis

Pearson Correlation Coefficient Threshold

In order to investigate the optimal Pearson correlation coefficient (Pcc) threshold to run the CLUSTard pipeline, CLUSTard was repeated using the polished nanopore assembly (NAB LR-pol) and trimmed short-reads at four different Pcc thresholds: 0.97; 0.99; 0.997 and 0.999. The threshold 0.99 was initially determined systematically, then threshold values were chosen around it. The threshold 0.90 was also chosen but timed-out during the bin_feeder.py step after reaching the maximum time on the Viking computing cluster.

Other than the differing Pcc thresholds the same parameters were used for all other steps of each run. As the alignment of short reads onto the long read contigs would not be altered by a different Pcc threshold and to allow a comparison that would not be impacted by the mapping success of BWA, the same alignments were used for all CLUSTard runs. After the first `CLUSTard` run was completed the alignment directory was copied over into the directories of the other runs which allowed `Snakemake` to skip the step as the output files already existed.

After each `CLUSTard` run was completed `SAMtools` *`flagstat`* was used on the resulting `SAM` file to determine the percentage of the assembly that was captured in the clustered contigs. A script was written to parse the output of `Prokka` to determine the presence of tRNA and rRNA sequences in each cluster, which is used to determine the completeness of metagenomic bins and a required metric for high quality MAGs (Bowers *et al*. 2017).

Further Validation

Due to the lack of known truth for this dataset and the lack of time-series metagenome datasets that utilise nanopore sequencing, validation of `CLUSTard's` clustering method was not straightforward.

First, it was deemed appropriate to bin the NAB LR-pol assembly using established binning software. Two of the most popular binning software in use are `CONCOCT` (Alneberg *et al*. 2014) and `MetaBat2` (Kang *et al*. 2015). Unfortunately, there were issues installing `MetaBat2` on the Viking computing cluster so only `CONCOCT` could be run on this dataset.

CONCOCT

`CONCOCT` (v.1.1.0) was used to bin the NAB LR_pol, NAB SR, and NAB LR assemblies. First the short reads were mapped against the assembly contigs then sorted and indexed using `SAMtools` using the parameters *`sort`* and *`index`*. Then the basic usage steps from the CONCOCT documentation (github.com/BinPro/CONCOCT) were followed.

Once the clustering had completed, the clusters were run through `SeqKit` with the parameter *`stats -a`* to determine the binning statistics. Then `CheckM`, first with the parameter *`unbinned`* to determine how many of the input contigs were binned and then with the parameter *Lineage_wf* to determine the completeness and contamination of the bins. Kraken2 was also run with the same parameters as the CLUSTard NAB runs to taxonomically classify the clusters. GC±SD of the clusters was determined using a custom python script (gc_count.py). Prokka was run on the clusters to determine the number of predicted genes and the presence or absence of rRNA sequences and tRNA sequences to allow the classification of certain MAGs as high-quality. The 16S sequences identified by Prokka from the MAGs that were classified as high-quality were then run through Silva (Quast *et al*. 2013) to determine the 16S classification.

Sharon dataset analysis

Another time-series metagenomic dataset of an infant faecal microbiome from Sharon *et al*. (2013) was chosen to be binned by CLUSTard in order to determine how successfully a small metagenome would be binned and also allow the comparison of the binning to a well characterised metagenome. This time-course was specifically chosen as it has also previously been used to validate CONCOCT clustering, which enables a further comparison between CLUSTard and CONCOCT. Although only 11 timepoints were mentioned, 18 libraries of Illumina short reads were downloaded from SRA (SRA052203). Seven of these were re-sequenced samples as the samples in the first run did not provide enough data (Alneberg *et al*. 2014) however, no information about which samples these were, or the sampling order was given. Whilst, a correct order is not necessary for CLUSTard's clustering it is necessary to interpret the results in a biological way - which was not the ultimate goal for the clustering of this dataset.

Whilst an assembly already existed for this dataset it was necessary to reassemble the raw-reads as metagenomic assemblers have developed rapidly since 2013. The short-reads were first analysed by FastQC (v.0.11.8) (Andrews 2010) to check for the absence of adaptor sequences and then assembled using SPAdes with the parameter *--meta* and assembly statistics calculated using SeqKit *stats -a*. The presence of complete/partial

16S rRNA sequences was determined using `Prokka.` The `CLUSTard` pipeline (including the analysis and plotting steps) was then run at a threshold of 0.997 and a contig cut-off value of 1,000bp.

Kraken Databases

Due to the success of the `Kraken2` identification with the well characterised Sharon *et al*. (2013) dataset it was deemed prudent to investigate alternate classification methods. As `Kraken2` can accept user-defined index databases, the GTDB_r89_4k database (dated 23/07/19) as outlined in Méric *et al*. (2019) was downloaded and used to classify the clusters produced by the NAB LR-pol 0.997 `CLUSTard` run. This database (Kraken_GTDB) is built from many MAGs using the taxonomic system from Genome Taxonomy Database (GTDB) (Parks *et al*. 2018). To maintain consistency the database based on the NCBI taxonomic system would have been preferable, however at the time of writing the NCBI version of the `Kraken2` database is missing the taxonomic information so the GTDB database was used instead. GTDB taxonomy can be converted to NCBI taxonomy on the GTDB website.

Biological Analysis

Relative abundance plots were produced using the `Python3` script `abun_plot.py` with the `Kraken2` output produced by Kraken_GTDB. A plot of the top 19 genera for all digesters and feed was created and because much of the feed was classified as ‘other’ in this plot a separate plot was created using only the data from the feed. The 20 high-quality MAGs that were identified from the NAB LR-pol 0.997 CLUSTard run were further investigated by comparing the size and GC content to the top classified species from both the Kraken_DB and Kraken_GTDB indexes. The size and GC content for the reference species from Kraken_DB was determined from the NCBI `RefSeq` genome browser (O'Leary *et al*. 2016), the reference sequence taken from Kraken_GTDB was determined from the GTDB website. If more than one genome was available, the most complete and/or the genome marked as ‘representative’ was chosen. The 16S rRNA sequences were identified by `Prokka` and then run through `Silva` to determine if the 16S sequence matched the other taxonomy information.

All 20 MAGs were also run through `autoMLST` (Alanjary *et al*. 2019) which detects closely related genomes to the user-inputted query sequences to place them within a reference tree. First all 20 MAGs were run at once with the options: *`select default nearest organisms`* and *`concatenated alignment`* selected. As `autoMLST` only outputs 50 leaves the tree it meant that in-depth context for each MAG was missing, therefore each of the 20 MAGs were also run through `autoMLST` separately. The trees were then viewed in iTOL (Letunic and Bork 2019).

Results

Data Summary

DNA extracted from each of the 19 timepoints were sequenced separately by Illumina HiSeq, producing 198GB of raw sequencing data (351,735,338,778 bp). DNA extracted from all four digesters and input feed at timepoints 15 and 17 were pooled and sequenced on a PromethION using Oxford Nanopore Technology sequencing producing 55GB of raw sequencing data (61,874,783,098 bp).

Assembly Statistics

Assembly statistics for all three assemblies produced from the NAB dataset - Illumina only (NAB SR), Nanopore only (NAB LR), and Nanopore assembly polished with Illumina raw reads (NAB LR-pol) - can be seen in Table 1.

Briefly, the NAB SR assembly produced over 17 million sequences (contigs) with an average length of 802bp and an N50 length of 983bp. 96.79% of raw Illumina short reads mapped back to this assembly. The NAB LR assembly produced 78,439 contigs with an average contig length of 21,544bp and an N50 length of 38,925bp. 77.39% of raw short reads mapped back to this assembly. After the Nanopore assembly was polished the average contig length increased with to 21,895bp and the N50 length also increased to 39,579bp with no change in the number of contigs. The percentage of raw short reads mapping also increased to 78.36%.

Table 1: Summary statistics for the three assemblies produced from the NAB dataset. Illumina only (NAB SR), Nanopore only (NAB LR) and Nanopore assembly polished with Illumina short-reads (NAB LR-pol).

	Number of sequences	Total length (bp)	Mean contig length (bp)	Maximum contig length (bp)	N50 length (bp)	Raw SR mapping to asm (%)	Complete 16S sequences (partial)
NAB SR	17,833,222	14,294,830,142	802	688,912	983	96.79	140 (1489)
NAB LR	78,439	1,689,879,228	21,544	1,406,516	38,925	77.39	590 (301)
NAB LR-pol	78,439	1,717,415,419	21,895	1,420,398	39,579	78.36	2556 (91)

In order to easily determine how the coding density between the three assemblies differed the number of complete and partial 16S sequences were investigated. In the Illumina assembly there were 140 complete 16S sequences and 1489 partial 16S sequences. 590 complete 16S sequences were identified with the nanopore assembly, increasing to 2556 complete 16S sequences when the assembly is polished. Partial 16S rRNA sequences decreased from 301 to 91 when polished.

Parameter Optimisation

Pcc Threshold

To determine the optimal threshold value of the Pearson correlation coefficient (Pcc) for clustering, the `CLUStard` pipeline was repeated using the polished nanopore assembly at four different Pcc thresholds of 0.97; 0.99; 0.997; and 0.999.

Table 2 shows the binning statistics of the clusters produced at all four thresholds. As seen in the table, as the Pcc threshold is increased the number of clusters produced decreased as does the total size of the clusters.

Table 2: Binning statistics for the four NAB LR `CLUSTard` runs, at a Pcc threshold of 0.97, 0.99, 0.997 and 0.999.

	Pcc Threshold			
	0.97	**0.99**	**0.997**	**0.999**
No. of clusters	931	594	327	153
Total size of clusters (bp)	1,155,485,978	801,125,977	383,836,160	186,596,516
Percentage contigs binned (%)	35.44	16.7	4.51	1.63
Percentage bases binned (%)	67.28	46.65	22.35	10.86
MAGs >90% complete	98	68	30	7
MAGs <5% contaminated	847	530	310	149
No. of high-quality draft MAGs[A]	32	24	20	4
Total size high-quality clusters (bp)	106,972,458	76,215,271	62,149,967	11,240,922
No. MAGs with a genome quality ≥50 [B]	133	155	93	49
No. of clusters with a complete 16S	206	215	138	72
Predicted genes	210,018	216,734	132,318	65,784

A. As defined in Bowers *et al*. (2017) a high-quality draft metagenome-assembled genome (MAG) is classified as over 90% complete and under 5% contaminated and should also encode 16S, 5S and 23S rRNA genes as well as the tRNAs of 18 of the 20 amino acids.

B. As defined in Parks *et al.* (2017), genome quality of a MAG is completeness minus 5x contamination

It is also relevant to determine how much of the input assembly is actually captured by the clusters produced at different thresholds. In order to determine this the original assembly - in the case of Table 2, the Nanopore polished assembly (NAB LR-pol) - was mapped back onto the clustered contigs. As seen in Table 2, both the percentage of contigs that are binned, and the percentage bases binned decreased as the Pcc threshold increased. At a Pcc threshold of 0.97 the percentage of bases binned is 67.28% and the percentage of actual contigs binned is 35.44%. But at a threshold of 0.999 the percentage of bases binned is 10.86% and the percentage of contigs binned is 1.63%.

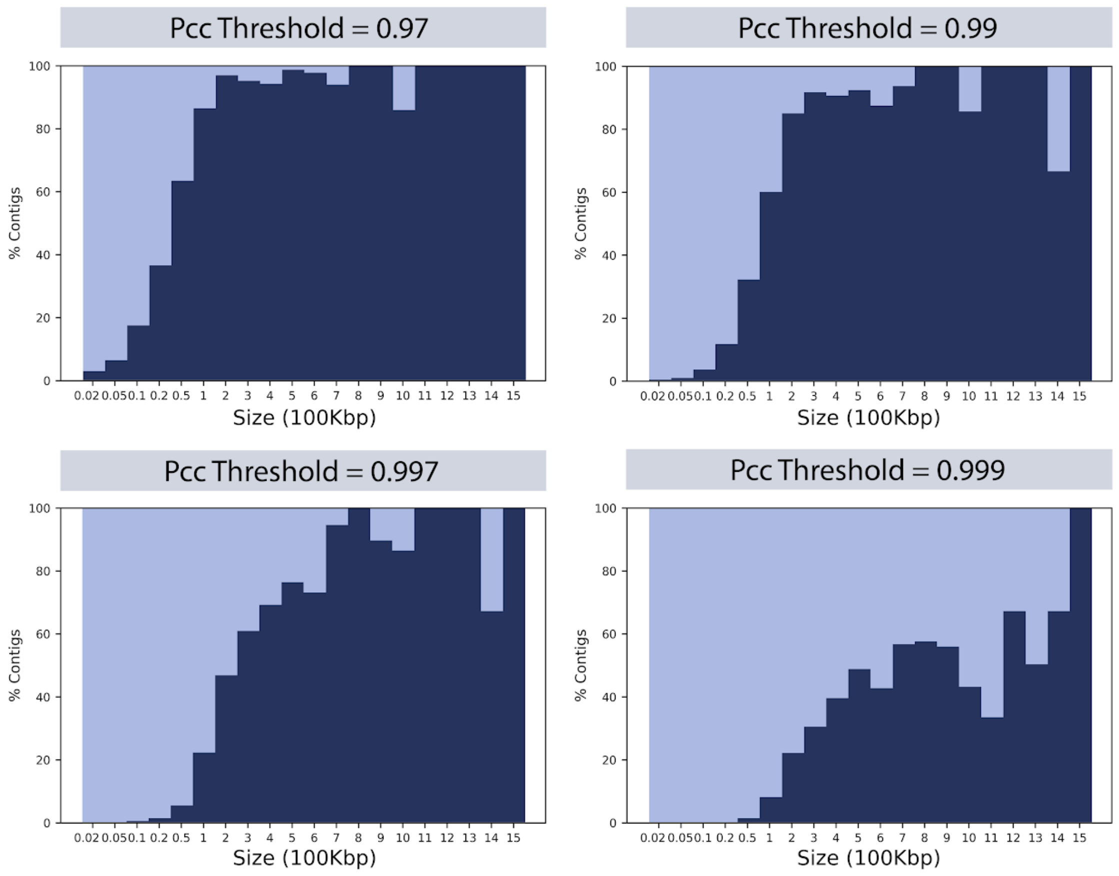


Figure 3: The percentage of different length contigs binned (dark blue) **or unbinned** (light blue) at the four Pcc thresholds; 0.97, 0.99, 0.997, and 0.999.

To further investigate the impact the change of Pcc threshold has on the results from the `CLUSTard` pipeline it is necessary to look at which contigs are clustered. Figure 3 shows the percentage of contigs clustered at the four different Pcc thresholds. The percentage of unbinned contigs increases as the Pcc threshold increases. At all thresholds a lower percentage of contigs <100kbp were binned when compared to contigs ≥100kbp. A reduction in the percentage of contigs ≥100kbp binned is seen as the threshold is increased however the biggest reduction in contigs binning is seen between a threshold of 0.997 and 0.999.

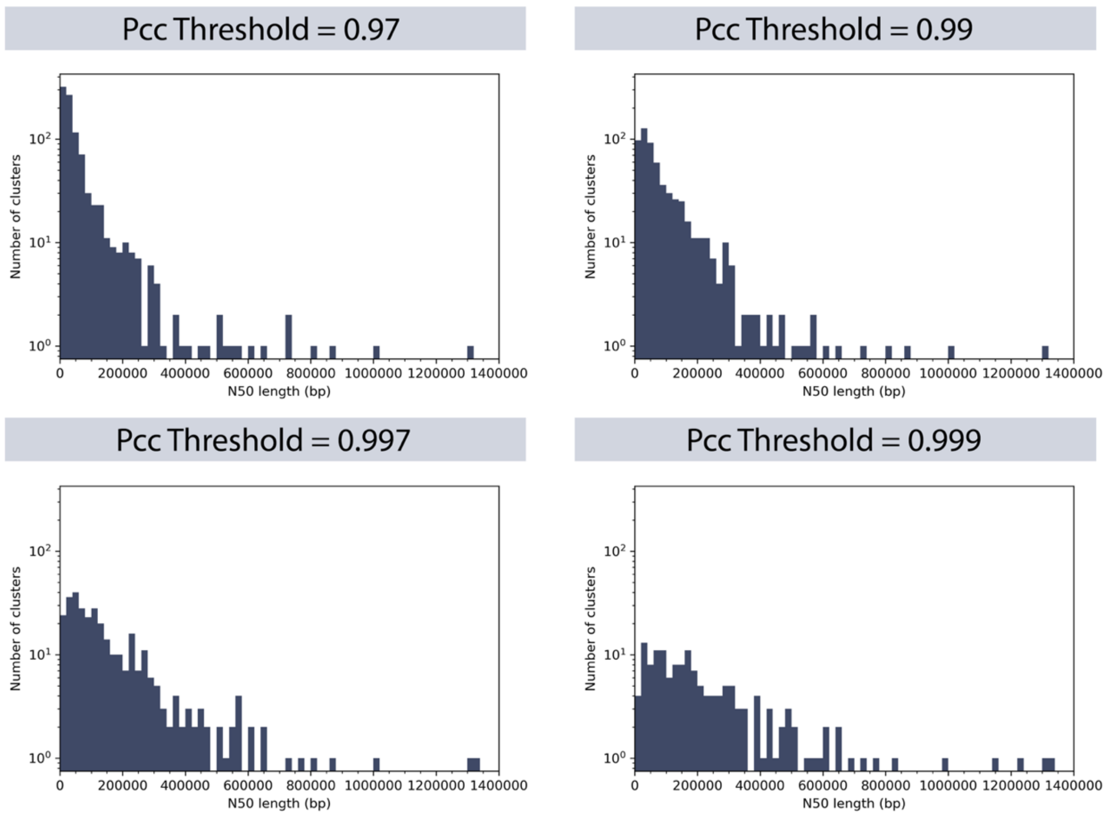


Figure 4: Distribution of the N50 length of the clusters produced at each of the four Pearson's correlation coefficient thresholds; 0.97, 0.99, 0.997 and 0.999.

In order to determine the success of binning at each threshold the N50 length of each cluster produced was investigated. A threshold of 0.97 produced the largest number of clusters (931) but as seen in figure 4, the N50 length distribution is skewed to the left, meaning that the vast majority of clusters have an N50 length under 200,000bp (874/931 clusters). As the threshold increases the N50 lengths become less skewed to the left. With 0.999 having the highest proportion of clusters with an N50 length of over 200,000bp (66/153 clusters).

In order to determine the effect that a changing threshold has on the quality of binning it is necessary to use a variety of different binning metrics due to the lack of a known ground truth for this dataset.

As defined in (Bowers *et al*. 2017) a high-quality draft metagenome-assembled genome (MAG) is classified as over 90% complete and under 5% contaminated and should also encode 16S, 5S and 23S rRNA genes as well as the tRNAs of 18 of the 20 amino acids. The number of clusters passing this metric that can subsequently be classified as high-quality draft MAGs decreased as the threshold increased - from 37 clusters passing this metric at a threshold of 0.97, to four clusters passing this metric at a threshold of 0.999. However, at a threshold of 0.997 the clusters classified as high-quality draft MAGs made up 16.19% of the total bases binned compared to 9.51% at a threshold of 0.99, 9.26% at a threshold of 0.97, and 6.02% at a threshold of 0.999.

In order to explore the effect that a changing Pcc threshold has on the gene completeness of clusters the presence of complete 16S sequences in clusters was investigated. As the threshold increased, the proportion of clusters with a complete 16S also increased from 22% of clusters at 0.97 to 47% of clusters at 0.999.

Genome quality is an additional metric that has been proposed by Parks *et al*. (2017). Quality is defined as completeness minus 5x contamination (both calculated by the program `CheckM`) and only genomes with a quality score ≥50 were kept for additional analysis. Here, the proportion of clusters with a quality score ≥50 increases as the threshold increases. With 133 (14%) of cluster at a threshold of 0.97 passing this criterion and 49 (32%) clusters at a threshold of 0.999 passing this criterion.

The variation seen in GC content within contigs in a cluster is another metric that can be used for clustering validation. The largest variation in GC was seen at a threshold of 0.99 at ±44.8%, at 0.97 the largest variation was ±17.5% and the largest variation seen at 0.997 was ±5.0% followed by 4.9% at 0.999. The average standard deviation was also impacted as the threshold changed, with 0.97 having an average SD of ±2.37, 0.99 an average SD of ±3.09, 0.997 an average SD of ±1.40 and 0.999 an average SD of ±1.13.

Figure 5 shows the percentage classification from `Kraken` at genus-level at each of the four thresholds. Whilst the classification percentages follow the same profile, 0.997 (yellow) has the most clusters classified to 100% identity (63, when compared to 32 at 0.97, 44 at 0.99, and 38 at 0.999).

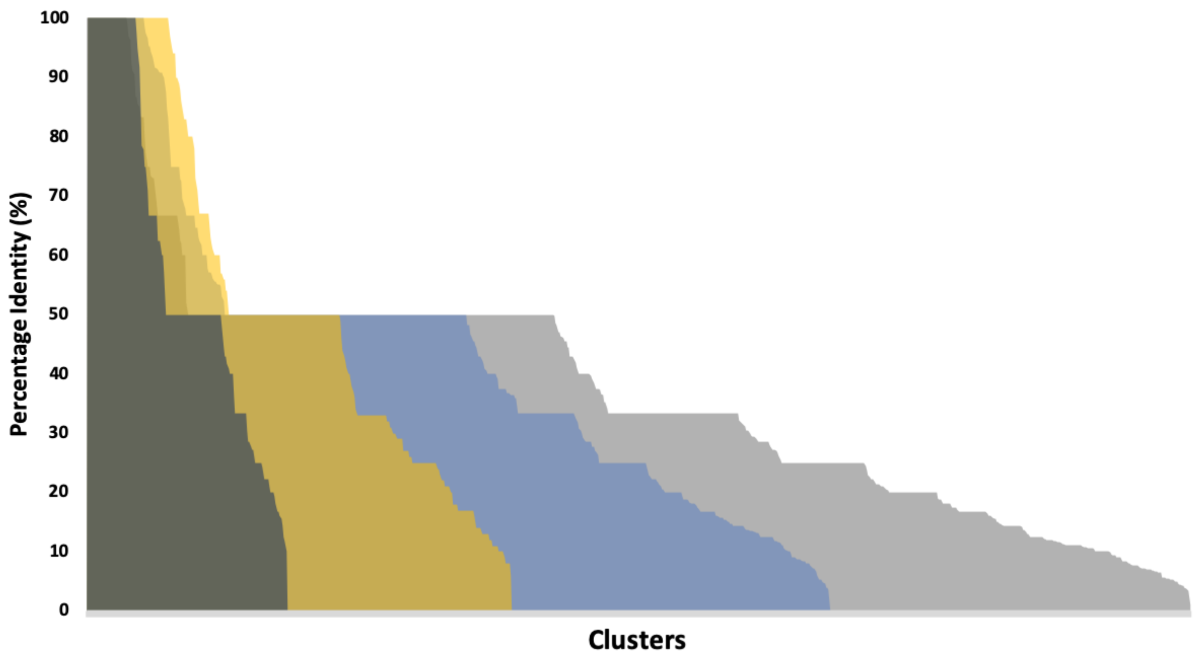


Figure 5: Percentage taxonomic identity for each of the clusters as defined by `Kraken`. At different Pcc thresholds; 0.999 (dark grey), 0.997 (yellow), 0.99 (blue) and 0.97 (light grey).

Figure 6 shows the visual output from `CLUSTard` for the largest four clusters at each of the four thresholds. Included in the visual output (Fig. 6) are other metrics that show changes as the threshold increases. At the lowest two thresholds (0.97 and 0.99) the largest cluster for each contains thousands of contigs with lower N50 lengths than the largest cluster for each of the two higher thresholds. At 0.97 the largest cluster has 13,578 contigs with a total size of 385Mbp and an N50 length of 40,307bp. The largest cluster for 0.99 is made up of 3,623 contigs with a total size of 110Mbp and an N50 length of 41,756bp. In contrast, at a threshold of 0.997 the largest cluster is made up of 21 contigs, at a total size of 6Mbp with an N50 length of 448,341bp and at a threshold of 0.999 the largest cluster is made up of 14 contigs at a total length of 4.9Mbp with an N50 length of 386,138bp. At lower thresholds a greater variation is seen in the abundance values (grey area) than when the Pcc threshold is increased.

The largest clusters for both 0.97 and 0.99 also have much lower coverage, 66-fold (±47) and 80.2-fold (±41.1) respectively, than the largest clusters for 0.997 and 0.999 at 723.8-fold (±111.8) and 748.3-fold (±85.7) respectively.

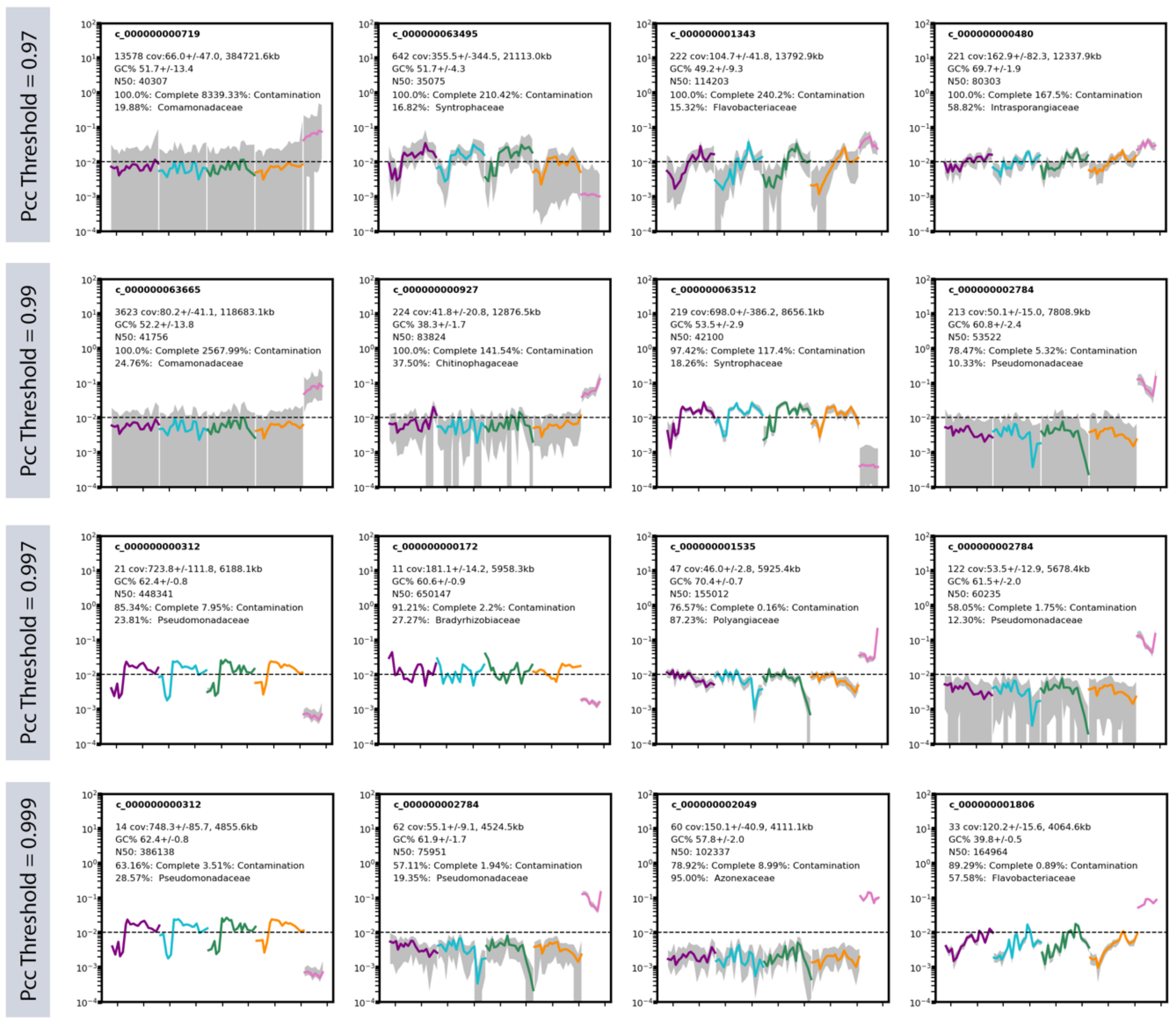


Figure 6: The output plots created by the `CLUSTard` pipeline for the largest four clusters produced at each of the four Pcc thresholds. The first line on each plot is the cluster identity. The second line shows the number of contigs clustered, the coverage (±1SD) of the cluster and the size of the cluster in Kbp. The third line shows the GC content of the cluster (±1SD) and the fourth line N50 length. Then the completeness and contamination values are on the next line and the top Kraken identity on the final line. The plot themselves shows the relative abundance of the cluster across all time points, here the colours correspond to the four digesters (1= purple, 2=blue, 3=green, 4=yellow) and the feed (pink). The grey area seen on the figures show the range in abundance values.

As the input data was used for each CLUSTard run the names of the contigs that remain consistent meaning it is possible to compare clusters between thresholds.

For example, in Figure 6 the cluster c_002784 appears in 0.99, 0.997, and 0.999 but it does not appear in 0.97. However, all contigs that are in Cluster_c_002784 at the threshold 0.99 are present in Cluster_c_00719 at 0.97 (this inconsistency is due to the fact that the clusters are named based on the longest contig present).

These clusters are directly compared in Table 3. The number of contigs in the cluster and the total size of the cluster decreased as the threshold increased and N50 length increased as the threshold increased. The variation seen in the fold-coverage decreased as the threshold increased as did the variation in GC content. Both completeness and contamination decreased as the threshold increased. The number of predicted genes decreased as the threshold increased.

Table 3: Cluster statistics for four comparable clusters over the four Pcc thresholds.

Pcc	Cluster ID	# Contigs	Size (Mbp)	N50 (bp)	Fold-coverage	GC-content (%)	Completeness (%)	Contamination (%)	Predicted genes
0.97	c_000719	13,578	385	40,307	66.0 (±47.0)	51.7 (±13.4)	100.00	8339.33	5528
0.99	c_002784	213	7.8	53,522	50.1 (± 15.0)	60.8 (±2.4)	78.47	5.32	1397
0.997	c_002784	122	5.7	60,235	53.5 (±12.9)	61.5 (±2.0)	58.05	1.75	1144
0.999	c_002784	62	4.5	79,591	55.1(±1.7)	61.9 (± 1.7)	57.11	1.94	1013

Using different raw reads

To determine the impact that different sequencing technologies could have on the result of clustering. The CLUSTard pipeline was then repeated using three different input assemblies from the NAB dataset (NAB SR, NAB LR and NAB LR-pol) at a Pcc threshold of 0.997, with the same time-series Illumina raw-reads used for abundance clustering. Table 4 shows the binning statistics for the clusters produced from the clustering of each different input assembly.

Table 4: Cluster statistics for the `CLUSTard` run with three different assemblies, NAB Illumina assembly (NAB SR), NAB Nanopore assembly (NAB LR) and NAB Nanopore assembly polished with Illumina reads (NAB LR-pol).

	Assembly		
	NAB SR	**NAB LR**	**NAB LR-pol**
No. of clusters	2184	331	327
Total size of bins (bp)	1,155,485,978	322,246,267	383,836,160
Percentage contigs binned (%)	0.04	3.62	4.51
Percentage bases binned (%)	0.85	19.07	22.35
MAGs >90% complete	0	0	30
MAGs <5% contamination	2184	323	310
No. of high-quality draft MAGs^{A}	0	0	20
Total size high-quality clusters (bp)	0	0	62,149,967
No. MAGs with a genome quality ≥50 ^{B}	7	2	93
No. of clusters with a complete 16S	7	113	138
Predicted genes	45,052	93,505	132,318

A. As defined in Bowers *et al*. (2017) a high-quality draft metagenome-assembled genome (MAG) is classified as over 90% complete and under 5% contaminated and should also encode 16S, 5S and 23S rRNA genes as well as the tRNAs of 18 of the 20 amino acids.
B. As defined in Parks *et al.* (2017), genome quality of a MAG is completeness minus 5x contamination

The Illumina-only (NAB SR) `CLUStard` run produced the greatest number of clusters (2184) but binned a small percentage of total sequences (0.04%) and total bases (0.85%). All clusters from the short-read only assembly had under 5% contamination, but none of the clusters reached >90% completeness, meaning no cluster would pass the criteria necessary to be classified as a high-quality MAG. However, seven clusters (0.32%) passed the Parks *et al*. (2017) criteria.

The nanopore-only assembly (NAB LR) `CLUSTard` run produced 331 clusters, binning 3.62% of the total input sequence and 19.07% of total bases. After the nanopore assembly is polished (NAB LR-pol) there are fewer total clusters but there was a slight increase in the number of sequences (4.51%) and bases binned (22.35%). None of the LR clusters reached >90% complete and therefore none could be classified as high-quality MAGs. After polishing, 20 clusters can be classified as high-quality draft MAGs. With the LR unpolished assembly two clusters passed Parks *et al*. (2017) criteria increasing to 93 clusters after the assembly is polished. When polished the number of clusters with a complete 16S rRNA gene increases from 113 to 138.

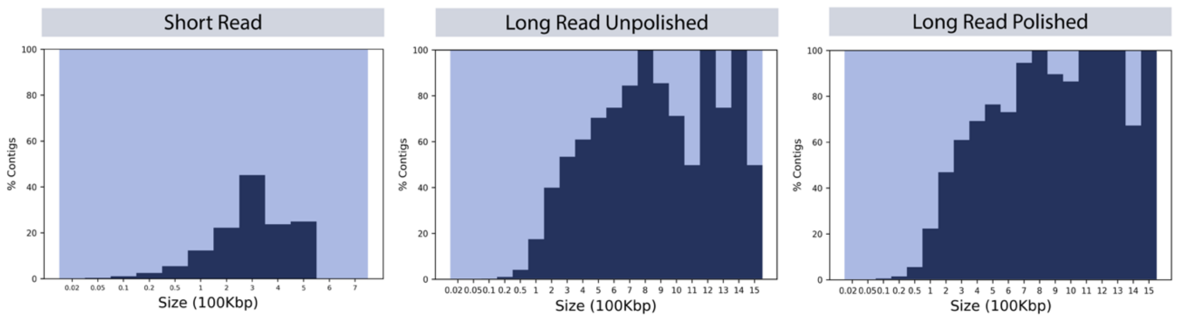


Figure 7: The percentage of different length contigs binned (dark blue) **or unbinned** (light blue) **for each of the `CLUSTard` runs with the three assemblies**.

Fig. 7 shows the percentage of contigs that have been binned for each of the assemblies. Although due to the large proportion of contigs in the SR assembly below 2000bp (16,877,845) it was necessary to increase the minimum contig length used for clustering to 2000bp for the SR assembly due to the computational challenge performing pairwise analysis for this many contigs. The `CLUSTard` run with the short-read assembly clustered a similar percentage of reads >50kbp as the other assemblies, but a much lower percentage of contigs ≥100kbp. Both long-read unpolished and polished assemblies have similar clustering success at lengths of ≥100kbp. However, when the long-read assembly is polished, contigs over 700kbp have clustered more successfully.

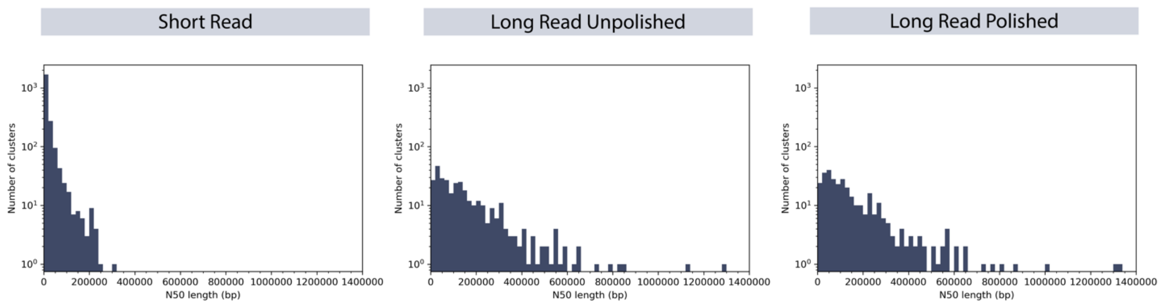


Figure 8: Distribution of the N50 length of the clusters produced by `CLUSTard` with each of the three assemblies.

The distribution of the N50 length for all clusters produced by each `CLUSTard` run can be seen in Fig. 8. The short-read assembly `CLUSTard` run produced clusters with a lower N50 length when compared to the long-read assembly `CLUSTard` runs both before and after polishing. A slight increase in N50 length is observed after the long-read assembly has been polished.

The short-read `CLUSTard` run produced 18/2184 clusters with at least one complete 16S rRNA gene. The nanopore-unpolished run produced 116/331 clusters increasing to 140/327 clusters with at least one complete 16S rRNA gene when the assembly is polished.

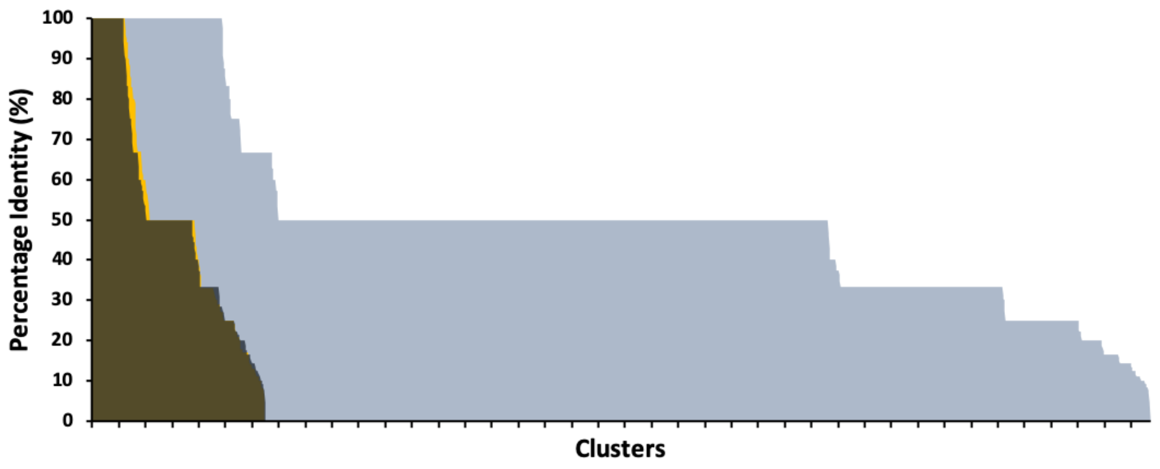


Figure 9: Percentage taxonomic identity at genus level for each of the clusters as defined by `Kraken`. From each of the three NAB LR (dark grey), NAB LR-pol (yellow) and SR (blue).

Figure 9 shows the genus-level `Kraken` identity for each of the three assemblies. After the NAB LR assembly was polished there was a slight increase in the `Kraken` classification, from 59 to 63 clusters at 100% identity. In contrast, 244 of the NAB SR clusters were classified to 100% identity by `Kraken`.

Figure 10 shows the largest four clusters produced after each of the three assemblies were run through the `CLUSTard` pipeline. The largest cluster produced by the short-read `CLUSTard` run (SR) is made up of 22 contigs with a total size of 3.03Mbp and an N50 length of 153,660bp, with a mean coverage of 17.5-fold (+-1x). The largest long-read unpolished cluster is made up of 104 contigs and a total size of 6.95Mbp and an N50 of 131,201bp with an average coverage of 134.6-fold (+-49.9). When the assembly is polished the largest cluster is made of 21 contigs with a total length of 6.19Mbp and an N50 length of 448,341bp with an average coverage of 723.8-fold (+-111.8).

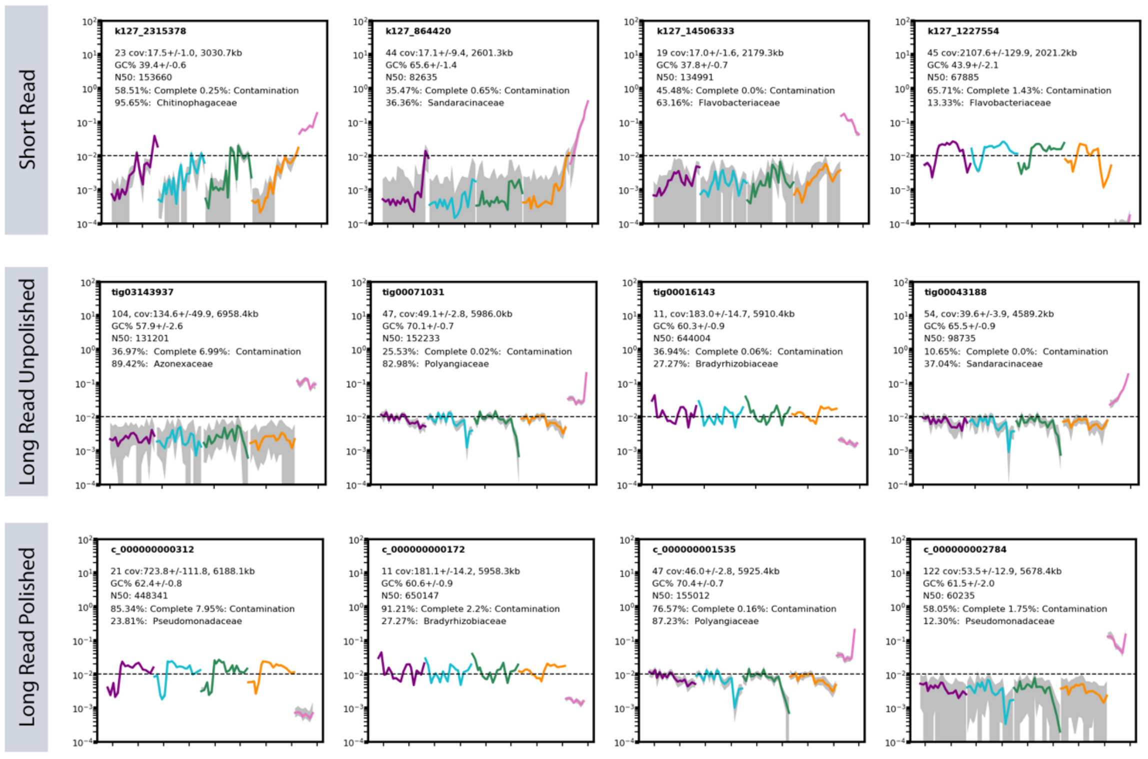


Figure 10: The output plots created by the `CLUSTard` pipeline for the largest four clusters produced with each of the assemblies. The first line on each plot is the cluster identity. The second line shows the number of contigs clustered, the coverage (±1SD) of the cluster and the size of the cluster in Kbp. The third line shows the GC content of the cluster (±1SD) and the fourth line N50 length. Then the completeness and contamination values are on the next line and the top Kraken identity on the final line. The plot themselves shows the relative abundance of the cluster across all time points, here the colours correspond to the four digesters (1= purple, 2=blue, 3=green, 4=yellow) and the feed (pink). The grey area seen on the figures show the range in abundance values.

As three different assemblies have been used as the input for each CLUSTard run the lack of coherence in contig names between each assembly makes the comparison between the clusters challenging. However, based on the similar GC-content, the similarity of the abundance change profiles and the similar Kraken identity seen in figure 10 it is highly likely that tig016143 and c_00172 contain much of the same sequence and as do tig071031 and c_01535. After polishing the tig00016143/c_0172 pair increases from 36.94% complete to 91.21% complete. With an increase in contamination from 0.06% to 2.20%. The total length also increases from 5,910,400bp to 5,958,300bp and the N50 length also increases from 644,004bp to 650,147bp although the number of contigs does not increase from 11. When polished the tig000071031/c_01535 pair increases from 25.53% to 76.57% complete and from 0.02% to 0.16% contaminated. With a decrease in the total length from 5,986,000bp to 5,925,400bp and an increase in N50 length from 152,233bp to 155,012bp. Although both are made up of 47 contigs.

Binning Validation

Comparing to other clustering software

In order to validate the clustering results of the CLUSTard pipeline, the NAB LR-pol assembly was run through CONCOCT - a popular program used for metagenomic assembly clustering. As seen in table 5, CONCOCT successfully binned all but one contig from the LR Pol assembly - producing 368 clusters with a total size of 1,717,414,419bp. However, only eight clusters passed the criteria necessary in order to be classified as a high-quality MAG at a total size of 32,950,345bp.

Table 5: Cluster statistics for the `CONCOCT` run with the NAB LR-pol assembly

	`Concoct` NAB LR-pol
Number of clusters	368
Percentage contigs binned (%)	100
Total size of bins (bp)	1,717,414,419
MAGs >90% complete (%)	161
MAGs <5% contamination (%)	145
High-quality draft MAGs[A]	8
Total size high-quality draft MAGs	32,950,345
No. MAGs with a genome quality ≥50 [B]	51
% Assembly mapping back to clusters	100
Number of clusters with a complete 16S	258
Predicted genes	355,597

A. As defined in Bowers *et al.* (2017) a high-quality draft metagenome-assembled genome (MAG) is classified as over 90% complete and under 5% contaminated and should also encode 16S, 5S and 23S rRNA genes as well as the tRNAs of 18 of the 20 amino acids.
B. As defined in Parks *et al.* (2017), genome quality of a MAG is completeness minus 5x contamination

The N50 length for `CONCOCT` clusters can be seen in Figure 11. When compared to the CLUSTard_997 run on the same assembly (see Fig. 8) `CONCOCT` produced more clusters with a lower N50.

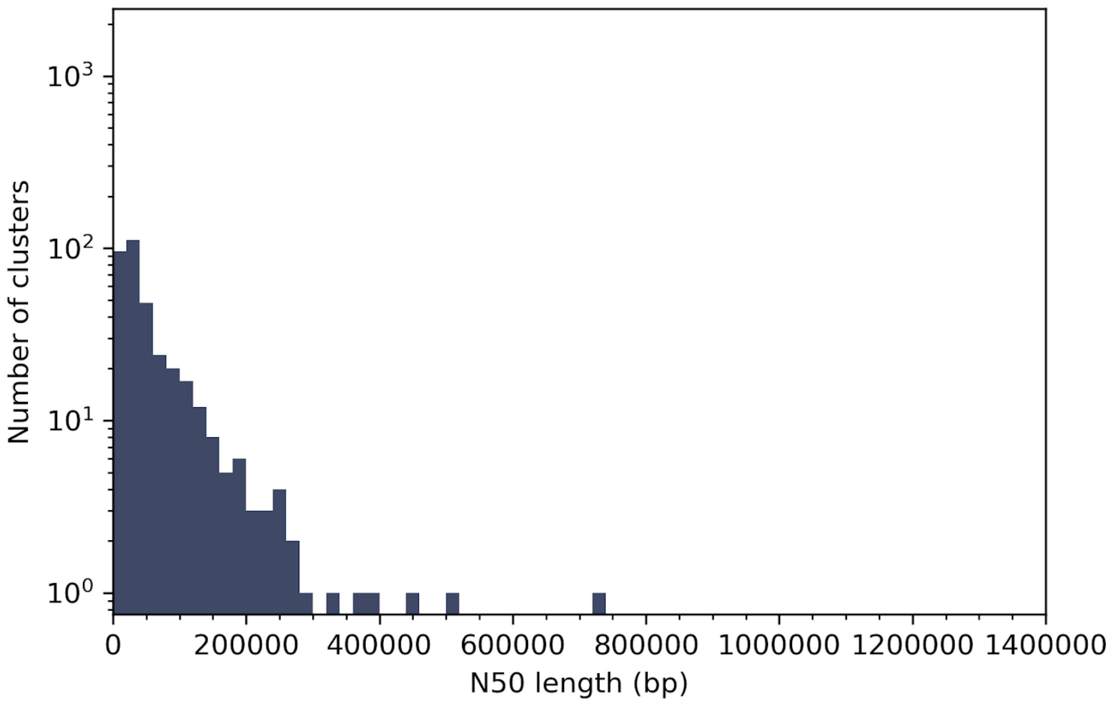


Figure 11: Distribution of the N50 length of the clusters produced by `CONCOCT` with the NAB LR-pol assembly

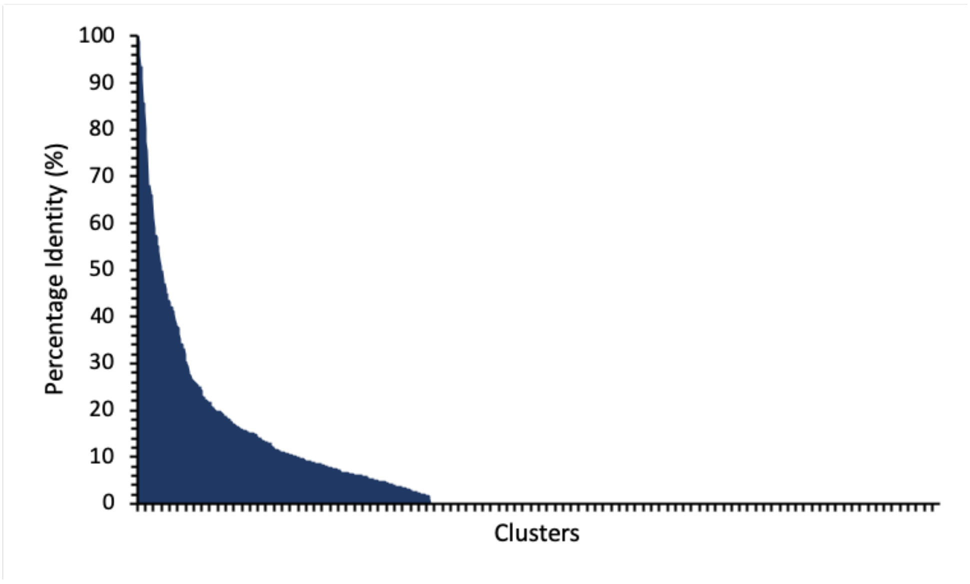


Figure 12: Percentage taxonomic identity at genus level for each of the clusters as defined by Kraken for the CONCOCT NAB LR-pol analysis.

The percentage identity of the Kraken classification at genus level from each cluster produced by CONCOCT can be seen in Fig. 12. Only three clusters were classified to a 100% identity and only 33/368 clusters were classified at ≥50% identity.

Statistics for the eight clusters classified as high-quality MAGs can be seen in Table 6. In comparison to the high-quality MAGs produced by CLUSTard, CONCOCT produced fewer high-quality MAGs eight compared to 20. The majority of these clusters have many more contigs than the MAGs produced by CLUSTard - and the only two clusters with a comparable number of contigs are significantly smaller in size. While the majority of the clusters are made up of more contigs than the CLUSTard run no outstanding difference is seen in the total size of clusters, but a decrease is seen in the N50 size. These MAGs are also less tight in GC content ranging from ±1.36% to ±4.83% SD when compared to ±0.5-2.2%. CONCOCT MAGs were also identified at family level to a lower percentage than CLUSTard MAGs - with only one cluster (129) reaching over 90% identity.

Table 6: Genome statistics of the eight high-quality metagenome-assembled genomes (MAGs) produced in the CONCOCT run of the NAB LR-pol assembly

Clust ID	No. Contigs	Size (bp)	N50 (bp)	GC-content (%)	Completeness (%)	Contamination (%)	Kraken Top Genus ID	Pred. genes	16S match
129	28	3,837,032	231,831	39.92 ±1.35	96.58	0.26	85.71% Petrimonas	1070	94.24% Dysgonomonadaceae
195	117	5,828,401	79,800	60.81 ±1.86	96.02	2.27	4.27% Streptomyces	1221	91.19% SGB-4
244	55	3,411,358	120,192	54.25 ±4.64	95.45	2.73	20.00% Anaerolinea	1122	92.31% Anaerolineaceae
276	20	3,193,484	242,872	52.09 ±1.78	95.00	2.42	15.00% Christensenella	1085	94.94% Christensenellaceae
331	54	5,703,507	250,817	65.92 ±2.17	98.13	4.37	25.93% Rubrivivax	1899	98.15% Burkholderiaceae
368	39	5,048,258	202,618	53.96 ±4.83	92.82	3.37	10.26% Pseudomonas	1318	89.96% Leptospiraceae
42	24	2,616,471	234,423	56.00 ±3.90	91.59	3.03	20.83% Dehalococcoides	975	93.95% Christensenellaceae
59	44	3,311,834	125,022	64.30 ±3.49	92.32	4.60	15.91% Streptomyces	1116	91.88% Spirochaetaceae

Clustering another dataset

To allow the comparison of CLUSTard's clustering to a 'known truth' a well characterised a time-series of a small metagenome dataset was chosen. This dataset from Sharon *et al.* (2013) (Sharon_DS) is made up of an 11 timepoints of short-read sequencing of an infant faecal microbiome. The raw-reads were reassembled using SPAdes (v 3.13.1) - assembly statistics can be seen in table 7.

Table 7: Metagenome assembly statistics for the Sharon *et al*. (2013) short read assembly

	Number of sequences	Total length (bp)	Mean contig length (bp)	Maximum contig length (bp)	N50 length (bp)	% Raw SR mapping	Complete 16S sequences (partial)
Sharon_DS	42,330	45,307,024	1,070.3	1,069,435	16,398	94.27	0 (11)

The assembly was run through `CLUSTard` at a Pcc threshold of 0.997 with an estimation of the sampling order (based on the abundance plots).

Table 8: Binning statistics of the Sharon *et al.* (2013) dataset when run through the `CLUSTard` pipeline.

	Sharon DS
Number of clusters	79
% binned sequences	5.32
% binned bases	64.46
Total size binned (bp)	29,203,531
MAGs > 90% complete	7
MAGs < 5% contam	77
High-quality draft MAGs[A]	0
No. MAGs with a genome quality ≥50 [B]	7
Number of clusters with complete 16S	0
Total no. predicted genes	9931

A. As defined in Bowers *et al*. (2017) a high-quality draft metagenome-assembled genome (MAG) is classified as over 90% complete and under 5% contaminated and should also encode 16S, 5S and 23S rRNA genes as well as the tRNAs of 18 of the 20 amino acids.
B. As defined in Parks *et al.* (2017), genome quality of a MAG is completeness minus 5x contamination

The binning statistics for the Sharon *et al*. (2013) dataset can be seen in table 8. In total 79 clusters were produced, binning 5% of sequences and 64.46% of all bases for a total binned size of 29Mbp. The distribution of contigs binned and the N50 distribution of the contigs can be seen in Appendix Fig. 2 and 3 respectively. In total seven MAGs pass the completeness and contamination criteria but not the gene completeness for a high-quality metagenome.

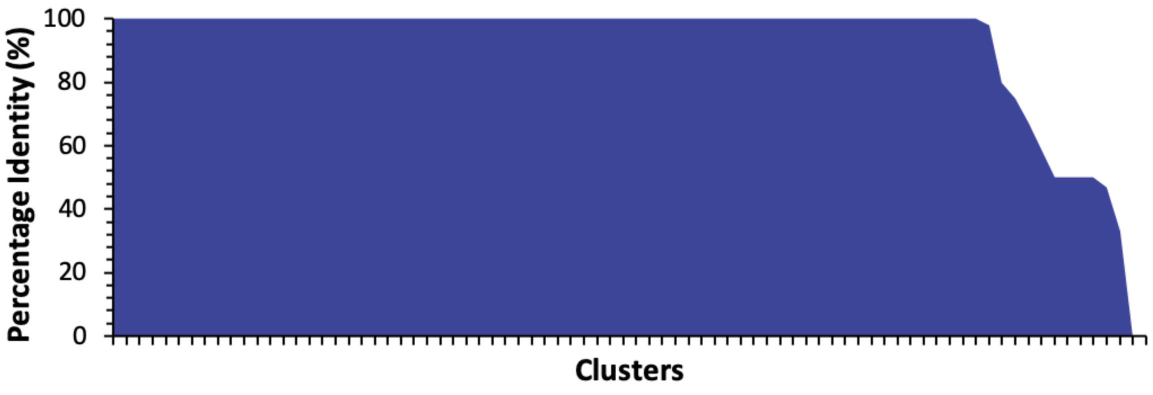


Figure 13: Percentage identity at genus level of the Sharon_DS `CLUSTard` clusters as determined by `Kraken`

Fig. 13 shows the percentage identity of the `Kraken` classification at genus level of the Sharon_DS clusters. With the Sharon_DS `CLUSTard` run, `Kraken` was more successful, at genus level classifying 76 contigs at ≥50% identity, 67 of these contigs at 100% identity.

The output for the largest four clusters produced by the Sharon `CLUSTard` run can be seen in Appendix Fig. 4. The seven clusters with `CheckM` completeness >90% and contamination <5% can be seen in fig. 14. The seven clusters range in size between 1.80 to 2.86Mbp. All but one was classified to 100% identity at species level by `Kraken`. The abundance variation (grey area) is high in some clusters NODE_75, NODE_161, NODE_220 and NODE_342 in particular.

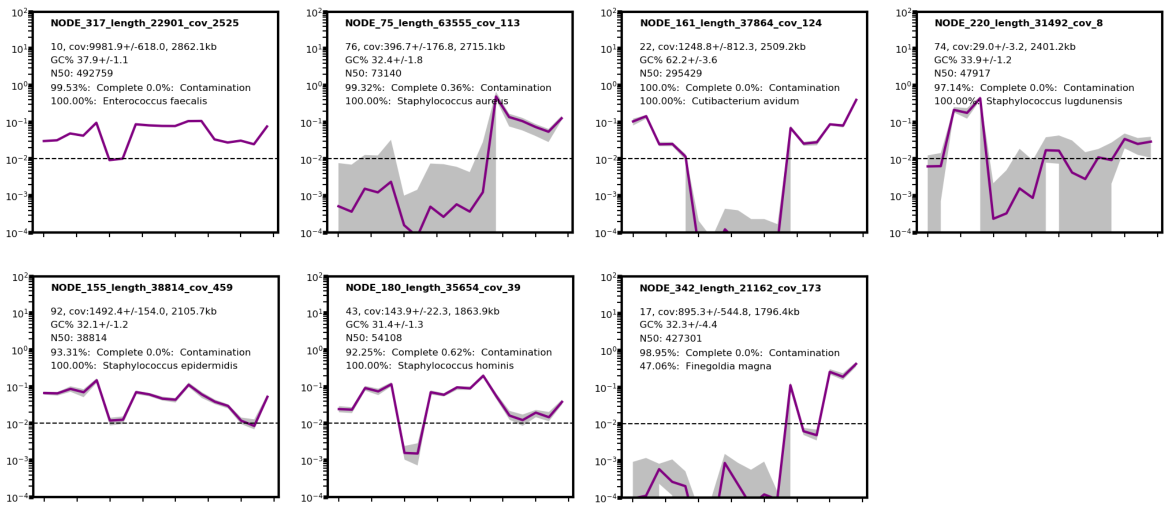


Figure 14: The seven highly complete clusters produced during the Sharon_DS `CLUSTard` run. The first line on each plot is the cluster identity. The second line shows the number of contigs clustered, the coverage (±1SD) of the cluster and the size of the cluster in Kbp. The third line shows the GC content of the cluster (±1SD) and the fourth line N50 length. Then the completeness and contamination values are on the next line and the top Kraken identity on the final line. The plot themselves shows the relative abundance of the cluster across all time points. The grey area seen on the figures show the range in abundance values.

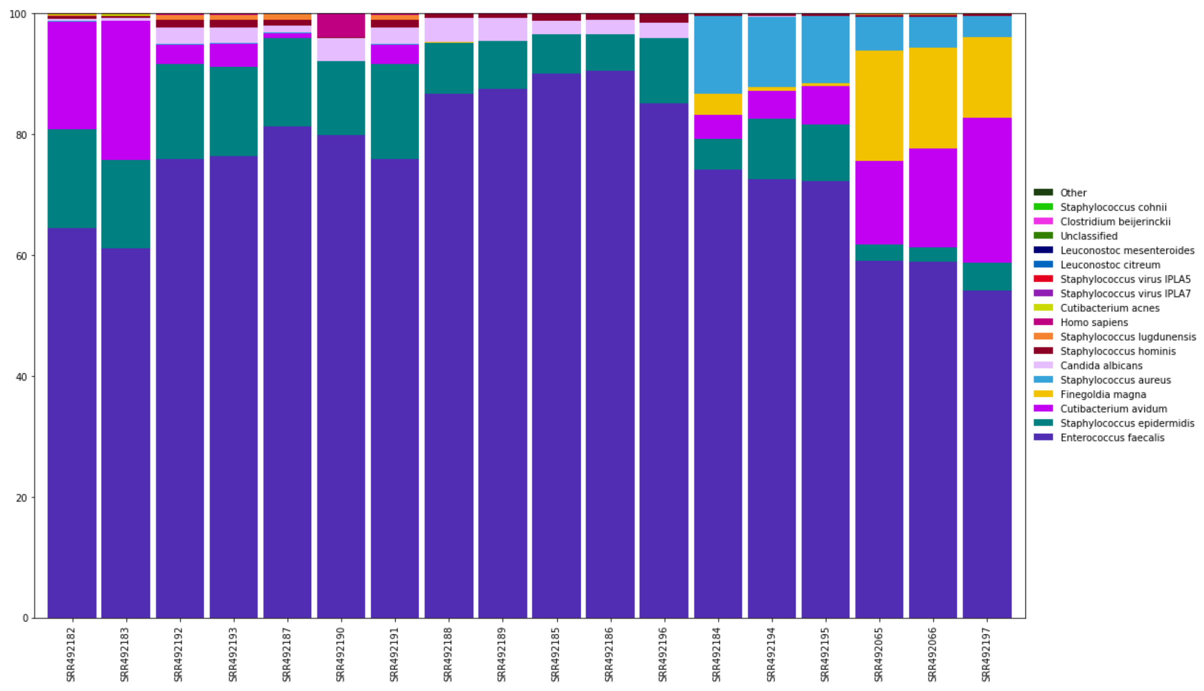


Figure 15: The relative abundance profile of the clusters produced from the Sharon_DS `CLUSTard` run, with the species identity determined by `Kraken`.

The abundance profile for the timepoints at species-level can be seen in figure 15, this follows a similar pattern to what is seen in Figure 2 from the Sharon *et al*. (2013) paper. Although the order of the samples for the `CLUSTard` run were estimated, some species follow a similar abundance profile to those in the Sharon *et al*. (2013) paper. With *Enterococcus faecalis* (purple) remaining largely abundant throughout the experiment, *Staphylococcus aureus* (light blue) increasing in abundance at the end of the experiment and *Cutibacterium avidum* (likely labelled as *Propionibacterium* Carrol in the Sharon el al. (2013) paper) being abundant at the start of the experiment, decreasing in the middle days and increasing at the end. However, *Finegoldia magna* is shown to be much more abundant in the output from the `CLUSTard` run than in Sharon *et al*. (2013). Of the abundant and rare species outlined in Sharon *et al*. (2013), 10/13 species were present in the `CLUSTard` dataset.

`Kraken` Databases

As it is possible to use a different database with `Kraken` and due to the difference in `Kraken's` classification success between the datasets, and also the fact that many genomes identified in metagenomes remain un-characterised. `Kraken` was run to identify clusters with both the original `Kraken` index and a database that includes many MAGs identified from environmental samples (here, termed `Kraken_GTDB`) (Méric *et al*. 2019).

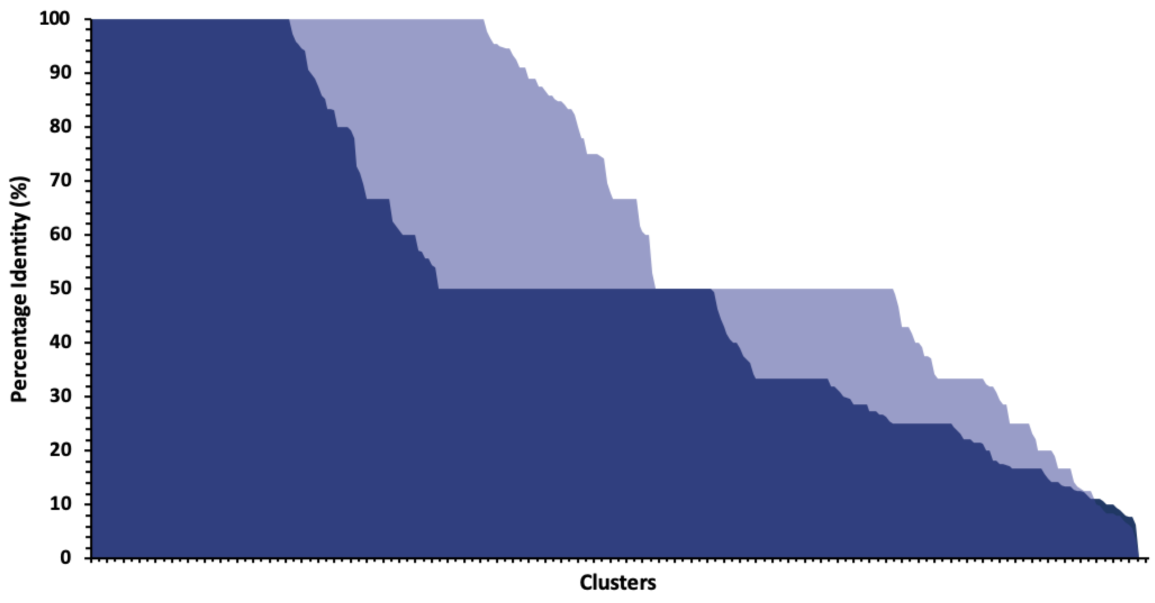


Figure 16: The percentage identity at genus-level of the NAB LR-pol 0.997 clusters after classifying with `Kraken` using both the Kraken_DB (dark blue) **and the Kraken_GTDB** (light blue) **indexes.**

As shown in Fig. 16 Kraken_GTDB classified more clusters at 100% identity (122) than Kraken_DB (62) and also has classified many clusters at a higher percentage identity than Kraken_DB, with 248 clusters at ≥50% identity compared to 92 clusters with Kraken_DB. Kraken_GTDB was more successful at all taxonomic levels, classifying 165 clusters to 100% identity at family level compared with the 72 identified by Kraken_DB.

Biology

One of the main goals of the `CLUSTard` pipeline was to allow the quick and easy analysis of large metagenomic datasets. Here, the `CLUSTard` pipeline enabled the NAB dataset to be further analysed. Based on the earlier metrics the NAB LR-pol assembly that was clustered with a Pcc threshold of 0.997 and classified by Kraken_GTDB was chosen to continue analysis with.

Abundance Changes

In order to investigate the community dynamics as a whole, the relative abundance of each timepoint was explored. After the NAB_pol 0.997 clusters were classified to genus level by Kraken using the Kraken_GTDB index the relative abundance changes of the 20 most abundant genera at each timepoint were plotted and can be seen in Figure 17. 169 different genera were identified by Kraken_GTDB in the NAB dataset in total.

Over time, a change in the abundance profile can be seen – this pattern is highly similar between the four digesters. However, the abundance profile of the feed is vastly different – with the top 20 most abundant genera overall not being overly abundant in the feed, only *Flavobacterium* is present in a high level and this is not present to a particular high degree within the digesters. As the top twenty abundant genera differ for the feed a second abundance plot was produced with only the feed samples in order to accurately show what was there, this can be seen in Appendix Fig. 5. In the feed only minor differences in abundance were seen over time.

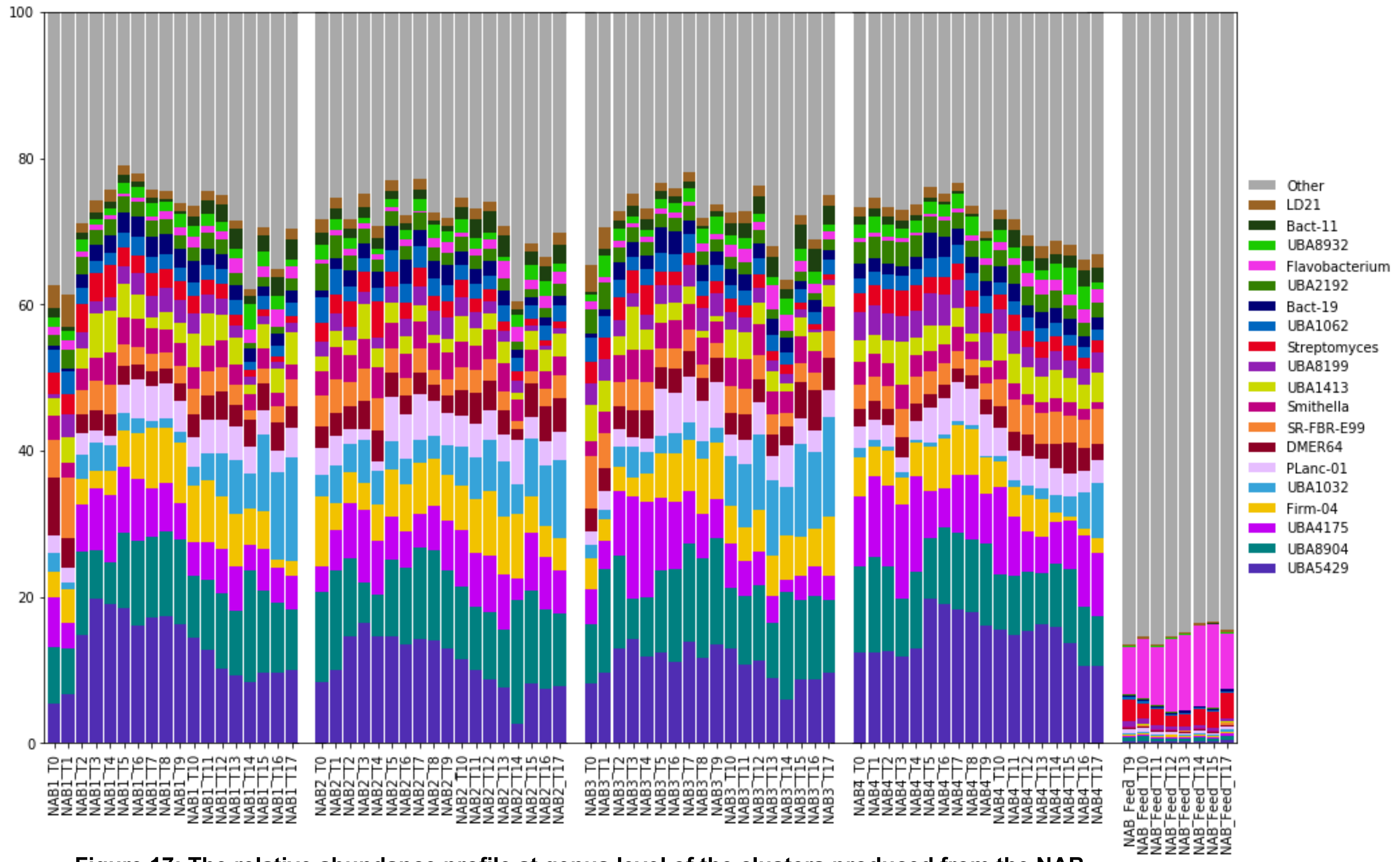


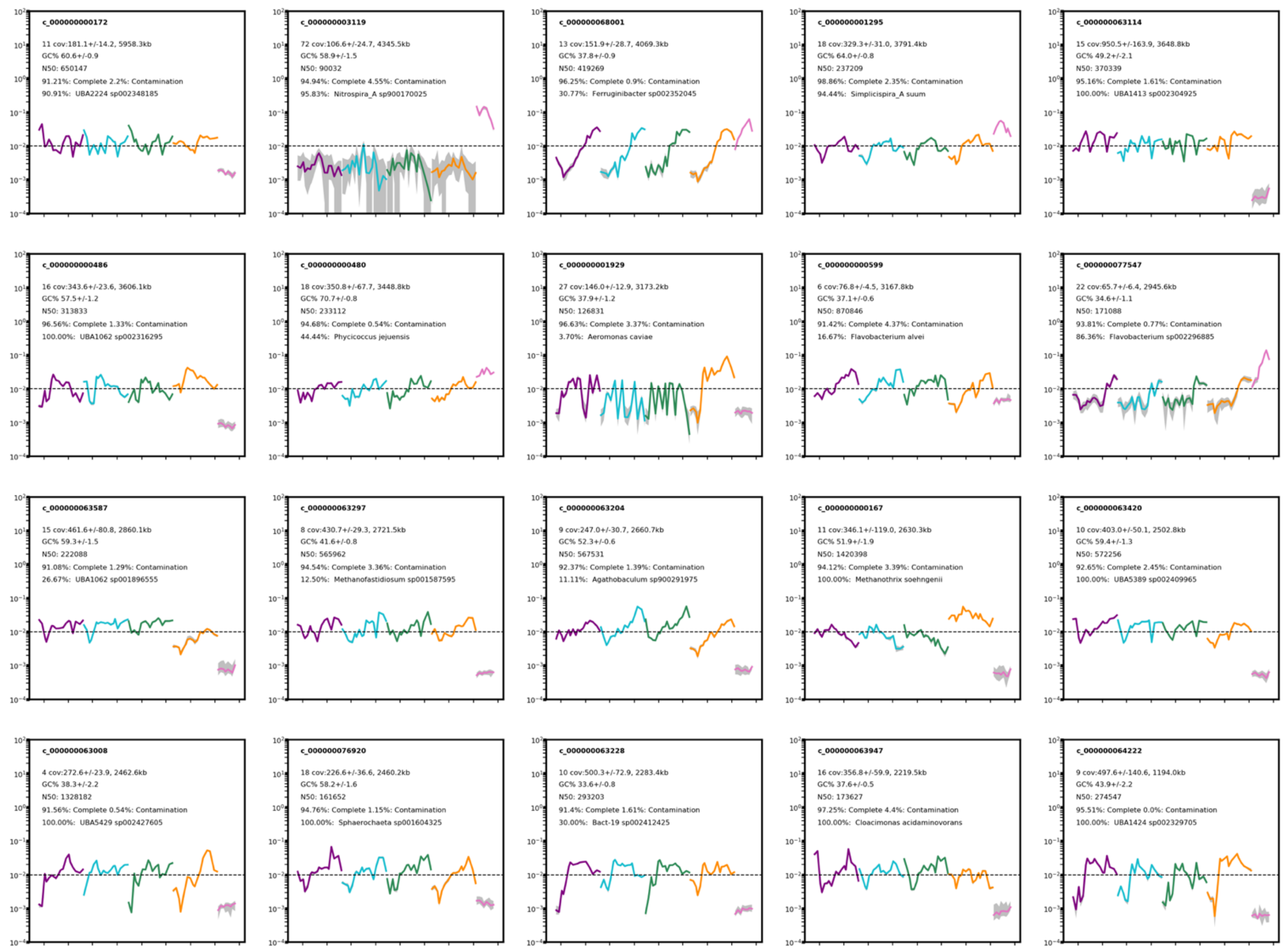
Figure 17: The relative abundance profile at genus level of the clusters produced from the NAB LR-pol `CLUSTard` run, with the genera determined by `Kraken` using the database Kraken_GTDB. The top 19 most common genera across all samples are shown in the plot, with all other genera contained in the *other* group (grey).

High-quality MAGs

In order to investigate community dynamics and to determine the success of clustering, the 20 high-quality MAGs produced by the CLUSTard 0.997 run (shown in Figure 18) were further investigated by comparing the cluster size, GC content and 16S rRNA sequences to related genomes - defined by the Kraken_DB and Kraken_GTDB classification. The outcome of which can be seen in Appendix table 1.

The clusters c_03119, c_01295, c_63114, c_00486, c_000167, c_63008, and c_63947 all have genome sizes within ±0.5Mbp of the Kraken_GTDB-identified related genome and the related genome has a GC content that falls into ±1 SD of the MAG GC-content.

Figure 18: The output plots for the 20 high-quality metagenome-assembled genomes (MAGs) identified from the NAB LR-pol `CLUSTard` run. First line on each plot is the cluster identity. The second line shows the number of contigs clustered, the coverage (±1SD) of the cluster and the size of the cluster. The third line shows the GC content of the cluster (±1SD) and the fourth line N50 length. Then the completeness and contamination values are on the next and the top Kraken identity on the final line. The plot themselves shows the relative abundance of the cluster across all time points, here the colours correspond to the four digesters (1= purple, 2=blue, 3=green, 4=yellow) and the feed (pink). The grey area seen on the figures show the range of abundance values.



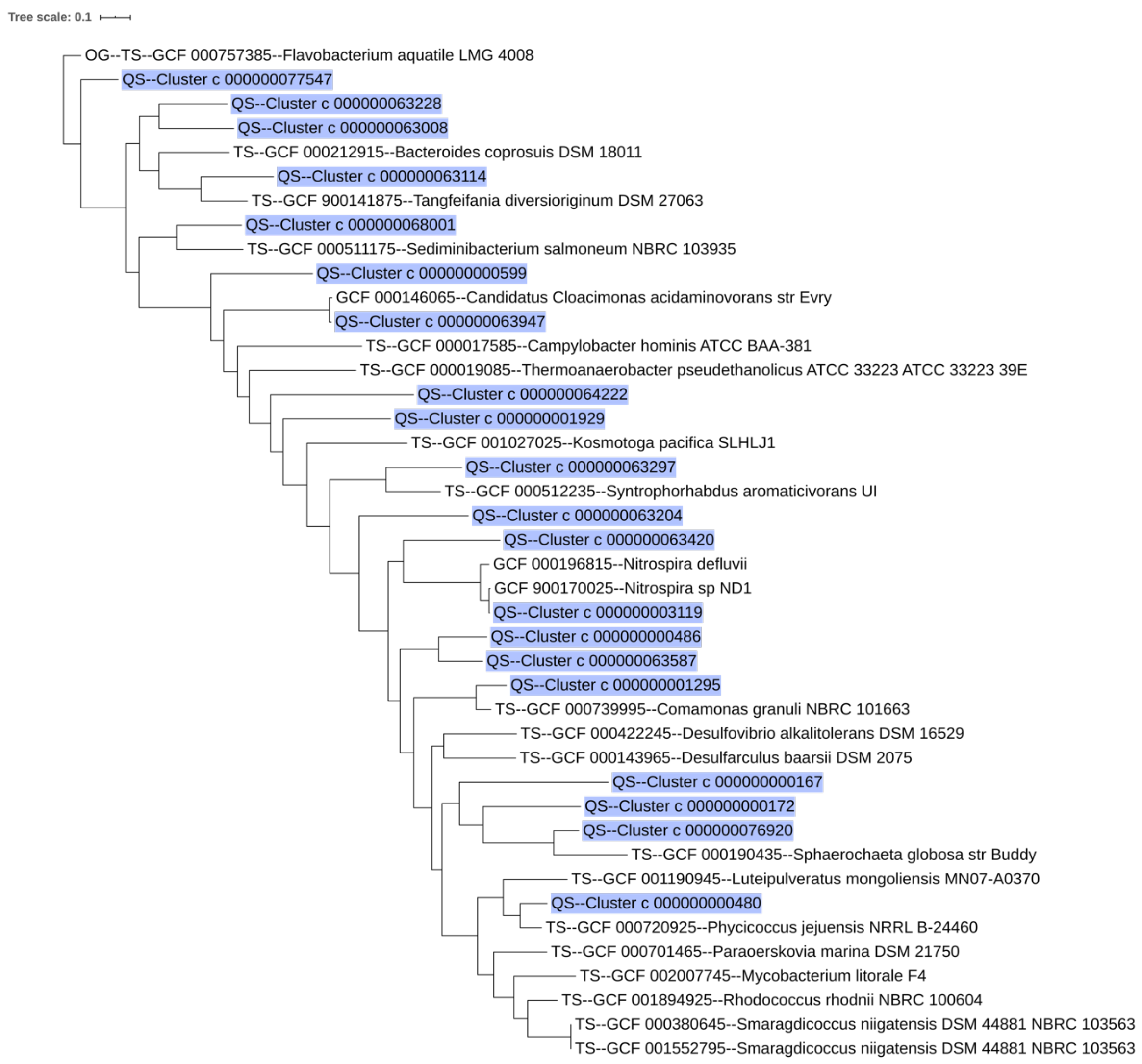

Tree scale: 0.1
OG--TS--GCF 000757385--Flavobacterium aquatile LMG 4008
QS--Cluster c 000000077547
QS--Cluster c 000000063228
QS--Cluster c 000000063008
TS--GCF 000212915--Bacteroides coprosuis DSM 18011
QS--Cluster c 000000063114
TS--GCF 900141875--Tangfeifania diversioriginum DSM 27063
QS--Cluster c 000000068001
TS--GCF 000511175--Sediminibacterium salmoneum NBRC 103935
QS--Cluster c 000000000599
GCF 000146065--Candidatus Cloacimonas acidaminovorans str Evry
QS--Cluster c 000000063947
TS--GCF 000017585--Campylobacter hominis ATCC BAA-381
TS--GCF 000019085--Thermoanaerobacter pseudethanolicus ATCC 33223 ATCC 33223 39E
QS--Cluster c 000000064222
QS--Cluster c 000000001929
TS--GCF 001027025--Kosmotoga pacifica SLHLJ1
QS--Cluster c 000000063297
TS--GCF 000512235--Syntrophorhabdus aromaticivorans UI
QS--Cluster c 000000063204
QS--Cluster c 000000063420
GCF 000196815--Nitrospira defluvii
GCF 900170025--Nitrospira sp ND1
QS--Cluster c 000000003119
QS--Cluster c 000000000486
QS--Cluster c 000000063587
QS--Cluster c 000000001295
TS--GCF 000739995--Comamonas granuli NBRC 101663
TS--GCF 000422245--Desulfovibrio alkalitolerans DSM 16529
TS--GCF 000143965--Desulfarculus baarsii DSM 2075
QS--Cluster c 000000000167
QS--Cluster c 000000000172
QS--Cluster c 000000076920
TS--GCF 000190435--Sphaerochaeta globosa str Buddy
TS--GCF 001190945--Luteipulveratus mongoliensis MN07-A0370
QS--Cluster c 000000000480
TS--GCF 000720925--Phycicoccus jejuensis NRRL B-24460
TS--GCF 000701465--Paraoerskovia marina DSM 21750
TS--GCF 002007745--Mycobacterium litorale F4
TS--GCF 001894925--Rhodococcus rhodnii NBRC 100604
TS--GCF 000380645--Smaragdicoccus niigatensis DSM 44881 NBRC 103563
TS--GCF 001552795--Smaragdicoccus niigatensis DSM 44881 NBRC 103563

Figure 19: Phylogenetic tree generated by `autoMLST` and visualised in `iTOL`, placing all 20 high-quality metagenome assembled genomes (highlighted in blue) into phylogenetic context.

All 20 MAGs were run through `autoMLST` to place them within a reference tree which can be seen in Figure 19. As `autoMLST` only outputs 50 leaves, each of the 20 were separately run through `autoMLST` to place the MAGs within a wider tree. This highlighted other MAGs of interest – for example c_77547 which did not match any species at a high degree of classification for either `Kraken` database and differed in GC content to the related species but has been placed in the middle of a *Flavobacterium* tree (Appendix Fig. 6), with *Flavobacterium aquatile* as the closest related species. This also allowed the cluster c_03119, which matched the size and GC content of the *Nitrospiria* species identified by the

`Kraken` databases very closely, to be seen in a tree of closely related *Nitrospiria* species in Appendix Fig. 7. For certain MAGs `AutoMLST` was not particularly successful in placing them in trees, this often matched the MAGs that Kraken_DB failed to highly classify, for example the cluster c_00172, whose `AutoMLST` tree can be seen in Appendix Fig. 8.

Discussion

Assembly

The assemblies required for this project proved to be a computational challenge not usual for metagenomic assemblies, which are challenging often due to the large dataset sizes and diverse population. Multiple challenges were faced by the Bioinformatics Core when assembling and polishing the long-read nanopore assembly. The first challenge faced was with the assembly software `Canu` (Koren *et al*. 2017) which requires a lot of memory and would often time out on the computing cluster. Whilst this issue was circumvented by completing the assembly on `Google Cloud`, this highlights the issue of memory when assembling metagenomes and especially when dealing with long-read sequencing. This area of bioinformatics is seeing rapid development, the software `metaFlye` (Kolmogorov *et al*. 2019) has since been released and is reportedly 10 to 300-fold faster, with an increase in assembly contiguity to `Canu` and is also capable of dealing with 150GB sequencing runs (Kolmogorov *et al*. 2019).

Problems were also encountered when assembling the short reads, `metaSPAdes` is purported to be the best option for short read metagenomic assembly (Vollmers *et al*. 2017). However, it was not possible to run `metaSPAdes` successfully on this dataset as it either ran out of memory or timed out before reaching even the first checkpoint. For this reason, the NAB SR assembly was completed using `MEGAHIT`, which is recommended by Vollmers *et al* (2017) where computational resources are limited. Although `MEGAHIT` completed the assembly without issue, it is highly probable that the assembly is sub-optimal as `MEGAHIT` has been reported to be biased towards lower abundant organisms (Vollmers *et al*. 2017). Whilst other short-read metagenome assemblers exist it seems to be that this area is under less active development than that of long-read metagenomics.

After all three assemblies; short-read; long-read; and long-read polished, had been completed it enabled the comparison between them. Although the short-read assembly

(NAB SR) produced a larger assembly it was less contiguous and therefore made up of more contigs of a shorter size – which is to be expected with short-read only assemblies as they cannot span regions of repeat (Goldstein *et al*. 2019). The total size of the Illumina assembly is around eight times more than the long-read assembly (NAB LR) of the same metagenome. This is probably because of both the increase in total size of the input raw-reads and the sub-optimal assembly leading to redundancy within the contigs, because the increase in size makes it much harder for the assembly software to assemble. As both `MEGAHIT` and `Canu` are each specialised for short-reads and long-reads respectively, they cannot be easily compared to each other. However, the overlap error correction step of `Canu` (Koren *et al*. 2017) increases confidence in the results from `Canu` as there is no such pre-processing step present in `MEGAHIT` (Li *et al*. 2015). However, it could be that the Nanopore assembly is missing contigs or organisms that appear in the SR data and further research should be done to either prove or disprove this.

The percentage of raw short-reads mapping back to the assemblies also indicates that data could be missing. Unsurprisingly, as the assembly was produced using them, 96.79% of the raw short-reads mapped back to the NAB SR assembly. However, even after the NAB LR assembly was polished by the raw short-reads (NAB LR-pol), only 78.36% of the raw short-reads mapped back to the assembly. There are many reasons that could cause this. As mentioned earlier it could be that the NAB LR and LR-pol assemblies are not capturing the entirety of the community. Only the later time points (T15 and T17) were long read sequenced so it is possible that an assembly of these timepoints does not capture species present earlier on in the time course. Another reason may be that some of the raw-reads did not match closely enough to the nanopore assembly. Nanopore sequencing is much less accurate than Illumina sequencing and prone to homopolymers (Goldstein *et al*. 2019). These sequencing errors may remain in the assembly and decrease the percentage of reads mapping back, especially as `BWA` is not built to deal with the increased error rates of Nanopore sequencing (Li and Durbin 2009).

Increased accuracy of the nanopore assembly was attempted by polishing the long-read assembly with the raw short-reads. Due to computational limitations only the short reads from time points 15 and 17 were used to polish this assembly. It may be possible to increase the accuracy of the sequence and therefore the number of reads mapping back by polishing the assembly with all available short reads.

16S sequences were used to represent gene completeness in the assemblies as these sequences are well conserved across all bacterial groups (Janda and Abbott 2007) and therefore easy to identify. The short-read assembly had fewer complete 16S sequences when compared to the long-read assembly both before and after polishing, but many more partial sequences. This indicates that the gene completeness of the Illumina sequence is likely to be much worse which potentially caused issues downstream in the clustering pipeline. This lack of gene completeness in the short-read assembly is in line with Goldstein *et al*. (2019) and is likely caused by the more fragmented assembly.

Snakemake

Snakemake allowed the easy chaining together of different programs and python scripts to build the `CLUSTard` pipeline. The use of `Snakemake` enabled the pipeline to be run with minimal supervision and was relatively painless over the seven separate times it was run for this project. Only one command was needed to run the entire pipeline once the configuration file had been altered to show `CLUSTard` where to look for the input files.

`Snakemake` also enabled different steps to be run as a job on the cluster in parallel or locally. This vastly decreased the real-time the pipeline took to run as certain steps in the process took only seconds to run with minimal memory requirements. They could be classified as local jobs and run locally without the need to wait in the computing cluster queue which often makes up the majority of the analysis time. The option of running jobs on the cluster in parallel and the fact that cluster jobs can be given different parameters for the computing cluster further reduced the amount of downtime waiting in the computing cluster queue.

The use of Conda environments for containerisation in Snakemake meant that no installation by the user was required for any of the packages used in the pipeline other than the Snakemake package itself (which can also be installed via Conda, although this would require user input) and any dependencies were also dealt with.

The pipeline was easy to re-run on a different dataset, or under different parameters as Snakemake only runs jobs if the modification time of the input files is newer than the output files, or if the output files are not present (Köster and Rahmann 2012).

As Snakemake is a Python-based workflow manager it provides an easy entry point. However, as the pipeline got increasingly complex so did Snakemake. In comparison to many programming languages and bioinformatics software Snakemake is currently lacking a large online community or tutorials outside of the official documentation, which made problem solving of issues encountered during the pipeline development challenging. One such issue was related to the *dynamic* function in Snakemake which can be used when the number of output files expected is not known before the job is run, for example the number of FASTA files produced during the clustering. This feature could not be used in this pipeline due to a file locking issue on our local computing cluster, which lead to the pipeline having to be split up into multiple different "sub-workflows".

Benchmarking the CLUSTard pipeline using these datasets would have been beneficial, however how to benchmark within the Snakemake wrapper remains elusive. Snakemake does have its own in-built benchmarking function but this was unhelpful as only wall-clock time was given and not CPU hours which is much more useful when benchmarking jobs that can run in parallel or across multiple cores on the computing cluster. It also was not apparent if this included time spent waiting in a computing cluster queue or not.

Another issue faced was with certain programs that require a set-up step when run for the first time. For example, when Snakemake first initiated the Conda environment for CheckM it prompted the user to set up a database which could not be done whilst in Snakemake. Therefore, in this pipeline a local installation of CheckM was used to circumvent this issue. A workaround for this issue should be implemented into the `CLUSTard` pipeline to enable `CheckM` to be run inside a `Conda` environment and thereby allow it to be portable to any system.

Pearson’s correlation coefficient threshold

As part of the testing process for `CLUSTard` an optimal threshold value for the Pearson's Correlation Coefficient (Pcc) value needed to be chosen. This threshold meant that only contigs with a correlation value above this were clustered together. It was, therefore, hypothesised that a lower Pcc threshold would allow less stringent clustering, inevitably clustering more contigs but potentially producing lower quality bins. Contrary to this, it was hypothesised that too high a threshold would be too stringent and potentially filter out contigs that should be clustered. Ultimately, of the four thresholds tested, the value of 0.997 was chosen as a compromise between minimising data loss and the overall quality of clusters. Although this threshold value did not produce the largest number of clusters (327 compared to 931 at a threshold of 0.97) those clusters that it did produce were higher in quality as 6% of clusters were classified as high-quality metagenome-assembled genomes (MAGs).

At all Pcc thresholds a small percentage of short contigs (<1Kbp) were successfully clustered. This is probably because of the reduced number of short read sequences mapping back to the contig and these smaller numbers cause greater variation between abundance values impacting the correlation value. Larger contigs do appear to cluster better than shorter contigs at all thresholds which is probably due to increased number of raw reads mapping to the contigs meaning that less noise is affecting the correlation value. However, a decrease in the percentage of larger contigs clustering is seen at a threshold of 0.999. This may be because these longer contigs would only need a couple of bases different to other contigs in order to no longer be clustered together at this threshold. These different bases or regions could be caused by the high error rate of nanopore sequencing, an error in the assembly or inherent differences within the population. Also, any other related contigs could be having the same problems hence compounding the issue.

Another difference seen between the thresholds is the N50 length of the clusters, as the threshold increased so did the average N50 length. This is probably because of the decrease in the number of smaller contigs being clustered thereby increasing the overall N50 length. This increased N50 length means that the clusters will be less fragmented which indicates that the assembly software has performed better on the dataset. But ultimately N50 length only gives a somewhat biased representation especially when the estimated genome size of the clusters is not known as it does not indicate the actual quality of the assembly as long contigs containing erroneous sequences could be confounding this value (Castro and Ng 2017). However, the N50 length can be compared here as it's all the same dataset.

Another metric that changed as the threshold increased was the percentage classification achieved by `Kraken`. At a threshold of 0.997 the greatest number of clusters were classified to 100% genus level. This indicates that higher quality clusters were produced at this threshold as any erroneous contigs in the cluster would pull the `Kraken` classification value down.

The variation seen in GC-content within a cluster is also a good metric for clustering validation. Whilst prokaryotic GC content varies massively the GC-content, between 15-75% (Reichenberger *et al*. 2015), within a genome less so – although there are still regions with very diverse GC content the overall variation within a genome is unlikely to be that high (Bohlin *et al*. 2010). The average standard deviation in GC-content reduced from ±3.09 at a threshold of 0.99 to ±1.40 at 0.997, reducing even further to ±1.13 at 0.999. This indicates that a high threshold produced clusters of contigs with closer GC content probably due to less contamination from sequences from different species. Although it is possible some of this variation exists because of differing GC content across the genome, the high variation seen within the GC content of some clusters produced at the thresholds 0.97 and 0.99 was due to the presence of contigs with large regions of homopolymers in the cluster.

In order to directly compare the different thresholds, the clusters c_002784 (at 0.99, 0.997, and 0.999) and c_000719 in 0.97, which were identified as highly similar in composition and therefore likely to be clusters of the same organism, were compared to each other. Ultimately, similar trends were seen between these clusters as were seen between all the clusters - as the threshold decreased the N50 length increased, the variation in GC content decreased and completeness and contamination also decreased. The less contaminated clusters meant that analysis was easier - allowing `Kraken` to classify the clusters to a higher degree which therefore meant that further analysis into the biological nature of certain clusters would prove easier and more informative.

This cluster at a threshold of 0.97 (c_000719) is made up of 13,578 contigs at a size of 385Mbp with a `CheckM` contamination value of 8339%, which indicates that many different species or strains are probably present in this cluster. This cluster seems to act as a catch-all for many contigs that are clustered elsewhere when a more stringent threshold is applied. These large 'catch-all' clusters seem to be common at lower thresholds with nine clusters at 0.97 and three clusters at 0.99 over 7.5Mbp with contamination values >100%. These clusters also have a large grey area on the output plots produced meaning that they have a large variation within the abundance of contigs.

Overall, the choice of a Pcc threshold is ultimately a balance between the number of useful clusters and the lack of contamination in the clusters produced. Although a higher proportion of high-quality MAGs were produced at 0.997, it is possible that the 12 more high-quality MAGs produced at the threshold of 0.97 could prove to be biologically meaningful clusters that were not seen at 0.997. For this reason, those additional 12 high-quality MAGs should also be investigated to determine if they hold any biological importance.

Whilst multiple Pcc thresholds were investigated it was by no means comprehensive, thresholds should have been chosen stochastically over a wider range of values. However, when the sPcc threshold of 0.90 was chosen, it timed out on this dataset before completing the clustering step, showing this could end up with diminishing returns. Whilst, with more time a result may be achieved at this threshold it is unlikely to cluster any better than 0.97 with a significant increase in time and computational power used. This may not be true for all datasets, a smaller dataset or a better initial assembly may allow for clustering at lower thresholds. It is also possible that other datasets may be clustered better at a different threshold depending on the diversity of the metagenome and a variety of other factors. It would be wise to do the sPcc threshold analysis with another dataset - preferably one with a known ground truth as the analysis done here was compounded by not knowing this.

Sequencing Technology

As this dataset was sequenced using both long-read sequencing and short read sequencing it allowed the investigation into the effects that the type of sequencing technology had on the `CLUSTard` pipeline and furthermore, the determination of which sequencing technology would provide the best clustering results. Of the three assemblies available for this dataset (SR, LR, and LR-pol), the assembly LR-pol was ultimately chosen to continue analysis with. Comparison of the clustering results between the sequencing technologies was hindered by the lack of coherence in contig names between different assembly software, indicating the need for more consistency in contig naming methods between assembly software.

Long-read polished versus unpolished

Initially, the LR-pol assembly was chosen over the unpolished assembly because of the increase seen in the number of clusters with a complete 16S sequence and an increase in the number of predicted genes in the clusters. This was not surprising as the number of complete 16S sequences in the original assembly increased after polishing (from 590 to 2556). This indicates that genes can be more precisely predicted in the more accurate polished assembly. As done with this dataset, polishing of Nanopore data with both short and long raw reads is commonly done in order to overcome the higher error rate of Nanopore sequencing whilst also maintaining the longer read length (Goldstein *et al.* 2019).

Similarly, the `CheckM` completeness values were poor prior to polishing with no clusters reaching >90% completeness - meaning that no clusters could be classified as high-quality draft MAGs. After polishing 30 clusters reached >90% complete. The reason behind this is also probably due to the increased error rate of the unpolished assembly as CheckM completeness is calculated based on a series of marker genes. The errors present in the assembly meant that the genes were not identified in the clusters.

Despite the increase in predicted genes and the decrease in sequence error after polishing, only a minor increase was seen in Kraken classification value of 59 to 63 clusters classified as 100% at genus level after polishing. This combined with Kraken's success with the short reads indicates that the algorithm for detecting the lowest common ancestors in Kraken still struggles even with the reduced error present in the Nanopore polished assembly.

An increased number of longer contigs were successfully clustered after the Nanopore assembly was polished. Before polishing, the longer contigs were likely to contain more sequencing errors. The reduction in the number of short raw reads mapping to the contig would cause the contig to drop below the sPcc threshold required for clustering. The reduction of sequence errors after polishing would therefore increase the likelihood of a long contig being clustered. The increase in longer contigs being clustered together with the overall increase of contig length in the polished assembly (the N50 increased from 38,925bp to 39,579bp after polishing) meant that the cluster N50 lengths also increased after polishing which produced more contiguous clusters.

Whilst the overall quality of the cluster increased after polishing, this increase was mainly caused by the higher-quality input assembly that was achieved after polishing. As the accuracy of Nanopore sequencing is likely to increase it is possible that high-quality assemblies can be produced without requiring any short-read polishing, it is likely the same could be possible with CLUSTard – requiring less data to produce an accurate assembly. As the short reads already existed for this dataset no extra experimentation was required to polish the assembly with short reads.

The sequencing technology used for the time-course data was not investigated here as only short-read time series data were available for this dataset. Theoretically, the length of the reads used for the time course should not impact the clustering algorithm. However, the increased error rate of the raw nanopore sequences would probably cause problems. A different mapping software would have to be used (`Minimap2` instead of `BWA`) in order to deal with the higher error rates (Li 2017). Even with the different mapping software the increased length of the reads would probably slow down mapping and the increased error rate would probably still prevent the most accurate mapping, hindering the clustering process. The reduced number of raw-reads produced with nanopore sequencing would also impact the clustering process. Due to the amount of sequencing necessary for a time-course, cost could also be a limiting factor as currently Illumina sequencing is less expensive than Nanopore (De Maio *et al*. 2019).

The dataset available for the NAB metagenomic community combined both Illumina and nanopore sequencing technologies. This combination of sequencing technologies could be having unexplored effects on the efficiency of clustering. Nanopore and Illumina sequencing have different error profiles (De Maio *et al*. 2019) and whilst the Nanopore assembly had been polished by the short reads, meaning the error will have reduced, it is possible that any error still present in the assembly will cause fewer short reads to be successfully mapped onto the contigs which may confound the clustering of contigs. An advantage to the mix of short and long reads used for clustering is cost. At this point in time Illumina sequencing is cheaper than Nanopore sequencing (De Maio *et al*. 2019) and therefore a time course of short-reads rather than long Nanopore reads would be lower in price. Due to the reduced clustering success seen with the NAB SR `CLUSTard` run it was hypothesised that the biggest factor in the success of clustering is in fact the quality of the assembly and not the sequencing technology used. However, different combinations of sequencing technologies should be investigated to determine the impact the choice of sequencing technology has on clustering.

Long-read versus short-read

The long-read polished assembly was chosen over the short-read assembly as the clusters produced were deemed higher quality. Whilst there were fewer clusters produced with the LR-pol assembly (327 compared to 2184 with the short reads) those that were produced were a much higher quality. No clusters with a completeness over 90% were produced by the short-read assembly. This incompleteness is mainly because the short-read clusters are much smaller in size than the LR-pol clusters, with 41 clusters over 500Kbp in length compared to 179 of the LR-pol clusters. The N50 lengths of the SR clusters are also lower when compared to the N50 lengths of the LR-pol clusters, indicating much more fragmented clusters. This low N50 length was seen in the input SR assembly and fragmented assemblies are a common issue with short-read assemblies (De Maio *et al*. 2019). The largest cluster of the short-read assembly was 3.03Mbp compared to 6.19Mbp from the long-read polished assembly and, although there is no set microbial 'genome length' (Bowers *et al*. 2017) it is likely that there would be some organisms with genomes larger than the largest cluster. This, along with the large number of clusters overall, indicates that many organisms may be split over multiple clusters.

`Kraken` has identified many more clusters to a higher degree at genus level with the SR assembly, with 244 clusters classified to 100% compared to the LR-pol assembly where 63 clusters were classified to 100% a genus level. This indicated that either the short-read clusters are being binned more successfully or the `Kraken` classification level is not a good indication of binning quality. The latter is more probable as the fewer errors in the SR sequences will allow more accurate identification of lowest common ancestor *k*-mers present in a cluster. The *k*-mer sequence is short which decreases computational time but means that any error seen in the *k*-mer sequence will have a large impact (Nasko *et al*. 2018). In the short-read assembly `CLUSTard` run issues with gene completeness have propagated from the assembly to the clusters - with only seven of 2184 clusters containing a complete 16S sequence. Although Illumina sequencing is ultimately more accurate, fewer genes were identified, probably due to the assembly being more fragmented. Therefore, the gene sequences were split across multiple contigs and hence cannot be identified. It is possible that these contigs have ultimately clustered together but have not been identified as genes due to this fragmentation.

Despite the larger number of clusters produced by the short-read CLUSTard run, a lower percentage of the input assembly and overall sequences were captured by the binning (0.04% of contigs were binned and 0.85% of bases were binned). This indicated that much of the population may be missing from the bins. This is backed up by the small clusters produced - rather than being highly fragmented they could be missing a lot of genetic information that was not clustered. Much of these missing data was probably caused by the requirement to only cluster contigs that were >2000bp in length, as due to the number of contigs that fell into this category (16,877,845) and therefore the number of pairwise comparisons necessary, clustering could not be completed because this proved to be a computational bottleneck. Although contigs of this size did not cluster with much success in the LR-pol run it is likely that a large amount of data is still being lost due to this.

Despite what was seen during the LR-pol assembly clustering, longer contigs were not binned to the same degree with the short-read assembly run. The short-read assembly is made up of fewer longer contigs overall but the majority of contigs over 500kbp were not clustered successfully which although there were fewer contigs of this size, is surprising.

Despite the issues seen in the clustering of the NAB short-read assembly, as seen with the Sharon dataset clustering, it is possible to get clusters which provide some biological information from short reads. It is likely that the Sharon dataset clustered well because it was a less complex metagenome with an increased depth of sequencing coverage. As a result, the dataset was much smaller hence a more contiguous assembly was produced using SPAdes, with an N50 length of 16,398bp when compared to the NAB SR dataset with an N50 length of 983bp. This, and the fact that the long-read polished genome clustered more successfully than the unpolished indicates that the quality of the input assembly is the most limiting factor on the quality of the clusters produced.

In this dataset the assemblies were challenging to produce, mainly because of the size and complexity of the metagenome. Here, in order to produce a better short-read assembly to ensure better clustering, random subsets of the short-reads available for this dataset should have been run through the assembler SPAdes, then the separate assemblies could have been merged and then de-replicated to remove any duplicate information. This division of the input data would have enabled SPAdes to be successfully run, probably producing a more contiguous assembly than the one produced by MEGAHIT and therefore allow more contiguous clusters to be produced. The long-read assembly could be further improved by polishing with all available short-reads. This should improve the sequence accuracy of the clusters produced and therefore enable CheckM, Prokka and Kraken to produce more accurate results.

Overall, a highly contiguous and relatively accurate assembly is required to properly utilise this method of binning. Not everyone will have an assembly that fits the bill with an associated time course but for those that do, this method of binning will be a fast and effective way of dealing with the vast amounts of data.

CONCOCT

In order to compare the clustering results of CLUSTard to popular software the NAB LR-pol assembly was run through CONCOCT - a well-used metagenomic binner that utilises sequence composition and coverage over multiple samples to cluster contigs (Alneberg *et al*. 2014). CONCOCT proved to be relatively easy to install as an up-to-date version is available through Conda. However, although the installation proved to be simple running the software was more complex with six different steps required. This does not include the necessary step of mapping the raw-reads onto the assembly as no instructions for how to do this is given.

Here, the SAM files produced by BWA when the short reads were mapped to the NAB LR-pol assembly were used for the coverage information. CONCOCT binned all contigs which had the benefit of retaining data. This could be seen with the increase of clusters with a complete 16S between the CLUSTard NAB LR-pol run at 42% (138/327) to 70% (258/368) with

CONCOCT. As CONCOCT clustered every contig this appeared to reduce the quality of clusters as the CONCOCT clusters had lower N50 lengths than the CLUSTard NAB LR-pol run because all of the smaller contigs were also clustered bringing down the N50 length.

Only eight high-quality draft metagenome-assembled genomes were produced compared to 20 in the CLUSTard run with the same input data. This lower number of high-quality draft MAGs is probably due to CONCOCT clustering everything, including low quality contigs, which increased both the completeness and contamination values of the clusters. The completeness increased because more of the sequence has been captured and therefore the clusters will be more complete. The contamination increased because the inclusion of low quality and possibly erroneous contigs will increase the number of sequences labelled as contaminants by CheckM. Similarly, to this the Kraken classification values of clusters is low, with three clusters reaching 100% classification at genus level and 33 clusters reaching 50% classification, whilst the increased error of the long-reads will probably have caused some of the reduction in Kraken classification values. It is likely that the lower quality contigs that have been clustered will confound the Kraken classification as they will have errors in them preventing them from being classified to a high degree, or they have been classified or even clustered erroneously. In the CONCOCT pipeline contigs under a certain length can be filtered out but although the authors recommend filtering out contigs <1000bp, no mention of it is made in the basic usage. The filtering out of shorter contigs would improve the N50 lengths of the clusters. However, even when filtering out the smaller contigs the issue of CONCOCT clustering of all contigs regardless of quality still remains because longer contigs are not necessarily any more accurate than short read contigs in Nanopore assemblies.

Another potential cause of the lack of high-quality draft MAGs produced is that short reads were used for the mapping information and a long-read assembly was binned overall. CONCOCT may not be able to deal with a mix of the two technologies or the long-read assembly itself - which is more error prone than the short-read assemblies this software was built for (Alneberg *et al*. 2014) - and therefore produce clusters of lower quality.

Despite being from the same input dataset as the NAB LR-pol `CLUSTard` run the eight high-quality draft MAGs created by `CONCOCT` could not be compared to the MAGs produced by `CLUSTard`. This issue was compounded by the relatively poor Kraken identification of the bins produced by `CONCOCT`. Comparison between different binning software is complex but a tool such as `AMBER` (Meyer *et al*. 2018) could be used to simplify this comparison. Overall, the fact that the clusters produced by `CONCOCT` were more contaminated and as there were fewer overall high-quality draft metagenome assembled genomes in comparison to the clusters produced by CLUSTard indicates that CLUSTard was more successful at binning this dataset.

Sharon Dataset

To investigate how successful `CLUSTard` clusters other datasets and in order to validate clustering by comparing the clusters produced to a known community raw Illumina sequencing data from Sharon *et al*. (2013) was downloaded, assembled and the resultant assembly run through `CLUSTard`. Although a time-course of raw-reads with associated nanopore sequences would have been ideal, a dataset that fills these requirements was not available at the time of writing. An issue with this dataset is the availability of 18 sequencing runs for 11 timepoints, seven of the sequencing runs were re-sequenced although no information was given about which ones or the order of the sampling (Alneberg *et al*. 2014). For this reason, an estimation of the sequencing order was taken as clustering can be completed regardless of the order of the samples which is only important if further analysis based on the community dynamics was desired which was not in this case.

Given the lack of success of the NAB short-read clustering it was expected that Sharon dataset would be similarly challenging to cluster. However, the `CLUSTard` run proved to be somewhat successful. Similar to the other `CLUSTard` runs, the shortest contigs did not cluster to as high degree as the longer contigs. However, the Sharon dataset contigs of <50,000bp did cluster more successfully than those of that length in the other `CLUSTard` runs. This increased success in binning the short reads is probably due to the more contiguous assembly which was in turn due to the smaller, less complex metagenome thus enabling SPAdes to be run successfully. Also, the higher depth of coverage in this dataset which meant that even the shorter contigs have many raw-reads mapping back to them.

With this dataset Kraken classification values had increased when compared to the other completed CLUSTard runs with the NAB dataset. Here the majority (67/79) of clusters were classified to 100% at genus level. As this is in line with the increase in Kraken identity seen with the NAB SR, it is likely that Kraken performs better with the more accurate short reads. It is also probable that Kraken classification performs better when faced with better characterised communities (Almeida *et al*. 2019) as the species in these communities are more likely to be present in the Kraken database.

Overall, the Sharon dataset was binned well, nine clusters were produced with a total length >500kbp. Of these, seven clusters were classified as >90% complete by CheckM and under 5% contaminated. However, none of these clusters passed the rRNA gene completeness required to be classified as a high-quality MAG. This is an issue that has propagated from the assembly to the clusters, with a low number of predicted genes identified and is related to the increased fragmentation of the short-read assembly (Denton *et al*. 2014).

These seven clusters match to species identified in Sharon *et al*. (2013), with five of these clusters corresponding in classification to all the “abundant species” and the other two clusters corresponding to “rare species” identified, however some species have since been reclassified and therefore are named differently. Although there are clusters that correspond to the other rare species identified, these clusters are not particularly complete which is probably because the sequencing depth was not high enough to capture the rare species and therefore not enough data were available to assemble and subsequently bin the rare species. In Sharon *et al*. (2013), four genomes were classified as “essentially complete” although no information is given about the completeness of tRNA sequences which is necessary to classify these as high-quality draft MAGs. When this dataset was subsequently run through CONCOCT (Alneberg *et al*. 2014) six “pure and complete genomes” were identified. The seven high-quality MAGs identified by the CLUSTard run were either comparable in size, coverage and N50 length or exceeded the genomes identified by Sharon *et al*. 2013. Alneberg *et al*. (2014) did not provide statistics for the “pure and complete genomes” produced. The abundance profile produced in the CLUSTard run was similar to the one produced in Sharon *et al*. (2013). This indicates that CLUSTard has captured the species diversity present in the metagenome in the clusters that have been produced.

Many small clusters have been produced by the Sharon CLUSTard run (68 clusters <100,000bp) and there are many reasons for this. The first being that they could be a genetic element e.g. plasmid at a different copy number to the rest of the genome. Or secondly the cluster is actually one of the phage identified in Sharon *et al*. (2013). As the software used for the downstream analysis of the CLUSTard clusters (i.e. CheckM, Prokka and Kraken) are not built for detection of phage it is probable that these would be missed.

It is also possible that these small clusters correspond to other clusters produced but they have not successfully been clustered together (i.e. the binning is fragmented). The reason for this could be the large drop in abundance at certain time points seen in the output plots for many of these clusters. This reduced abundance could correspond to the samples that were subsequently re-sequenced as “they did not provide enough data” (Alneberg *et al*. 2014) and as no information was given as to which of the samples these were, they were included in the CLUSTard run. This reduction in raw sequencing data at certain time-points could have caused these contigs to have a lower correlation with each other leading to them being clustered separately. This issue could be rectified by decreasing the Pcc threshold thereby enabling contigs with a lower correlation to cluster together.

The run of the Sharon dataset through CLUSTard not only shows that CLUSTard produces similar results to other metagenome binning software but also highlights that the quality of the assembly is the important factor for binning in CLUSTard. The short-read assembly of

Sharon dataset was binned much more successfully than the NAB SR assembly mainly due to the quality of the metagenome assembly. An interesting avenue of investigation would be to run the metagenome assembly from Sharon *et al*. (2013), which was assembled using outdated tools, through the `CLUSTard` pipeline to allow comparison of the binning success between a good and a sub-optimal short-read assembly.

Software Issues

Kraken Database

Due to the comparable success of the `Kraken` classification on the Sharon dataset when compared to the classification of the NAB dataset, `Kraken` databases were investigated.

The standard `Kraken` database is built with complete genomes from NCBI `RefSeq` using NCBI taxonomy (Wood *et al*. 2019). Whilst this may work for many well characterised microbial communities, in waste water AD many species remained either uncharacterised or unculturable - so called "microbial dark matter" (Kirkegaard *et al*. 2017). These species are therefore missing from the Kraken database which contains only complete genomes. These missing genomes meant that much of the NAB dataset remained poorly characterised by `Kraken`. Méric *et al*. (2019) outline this problem, proposing a purpose-built index database containing many MAGs to increase the classification power of Kraken.

This custom index database was downloaded and run on the clusters produced in the CLUSTard NAB LR-pol run. The database based around NCBI taxonomy would have been more consistent and would have allowed better comparison between the default Kraken2 database (Kraken_DB) and the custom one. Unfortunately, as the file containing the taxonomy information is missing, the database based on (GTDB) taxonomy had to be downloaded. However, Méric *et al*. (2019) report that the number of classified reads increased by using the "phylogenetically coherent" taxonomy of GTDB.

Here, the use of the custom index (Kraken_GTDB) over the default database (Kraken_DB) resulted in almost double (1.97x) the number of clusters being classified to 100% at genera level. The number of clusters being classified to over 50% at genera level had a 2.70-fold increase after Kraken_GTDB was used. This increase in classification is similar to what was seen in Méric *et al*. (2019), with a 2.2-fold increase in classified reads from soil-metagenome which are similar in community complexity to sewage-sludge (Frisli *et al*. 2013).

Although the use of Kraken_GTDB resulted in better classification for the NAB dataset, this may not be the case for all datasets run through `CLUSTard` - depending on the community of the metagenome. Well characterised metagenomes may be captured in the default `Kraken` index database. As the use of the Kraken_GTDB requires the maintenance of an external website in order to download the custom database, it will not be integrated into the pipeline. It will, however, be easy for the user to integrate a different index database and this should be communicated in the documentation.

Whilst the use of a custom `Kraken` index database meant that clusters were classified to a higher degree, it is likely that `Kraken` is still struggling to deal with the error rates currently intrinsic to nanopore assemblies. `Kraken` was initially built to deal with highly accurate raw short-reads, although it is now used in many metagenomic studies to classify clustered contigs (Nicholls *et al*. 2019), often due to its speed when dealing with large datasets. The *k*-mer based method of classification `Kraken`, the very reason for its speed, will likely struggle to correctly classify error prone sequences. As these short sequences (the default length is 31bp) have less room for sequencing error meaning that even a single base change will easily confound the results.

Further research into the impact of long error-prone reads and contigs on the successful classification of `Kraken` would be beneficial. Here, the clusters produced after the NAB SR `CLUSTard` run should be re-classified using Kraken_GTDB to see if further increase in classification is seen between the short read and long read assemblies. A custom database containing MAGs identified from anaerobic digestion metagenomes such as those from Campanaro *et al*. (2019) could further increase classification accuracy. Méric *et al* (2019) also built the database for the popular classification software `Centrifuge`. Therefore, this database (Centrifuge_GTDB), along with the default index database for `Centrifuge`, which is similarly fast and has low memory requirements of `Kraken` and was built specifically for metagenomic studies (Kim *et al*. 2016), should be compared to the `Kraken` results to determine if `Centrifuge` is better able to deal with noisy long-read assemblies. In Kim *et al*. (2016) 17% of nanopore raw-reads were successfully classified, however this number may increase after assembly and polishing.

Whilst easy and accurate classification is desirable in the pipeline, Kraken is primarily used in the `CLUSTard` pipeline to give the user an indication into the taxonomy of the cluster so that they can identify families or clusters of interest for in-depth and specialised downstream analysis.

CheckM

The issues with the noisy long-read assemblies could have also caused a problem with the `CheckM` assessment of `CLUSTard's` bins as `CheckM` utilises lineage-specific collocated marker genes to assess the quality of genomes (Parks *et al*. 2015). Again, although `CheckM` is ubiquitous in long-read metagenomic studies no investigation or benchmarking into how well `CheckM` deals with the increased error long-reads has been done. The high contamination and low completeness seen in many of the clusters produced from long read assemblies could in fact be because of the inherent noise present in the assembly (Watson and Warr 2019), not an issue with the `CLUSTard` binning as a similar issue is seen in the clusters from the long-read assembly with `CONCOCT`. The noise present may either lead to marker genes not being identified and therefore reducing the completeness value of the cluster or marker genes being incorrectly identified as a different lineage and therefore increasing the contamination of the cluster. `CheckM` should also be benchmarked against both short-read and long-read assemblies to determine the effect, if any, long-read assemblies have on the CheckM results.

Binning Issues

Perhaps the most important avenue for further research would be the clustering of a synthetic or known dataset by CLUSTard. This would enable easy analysis into the success and accuracy of the clustering algorithm. This was not done due to time restraints and the challenge of producing a synthetic metagenomic dataset that follows a complex microbial community over time. One experimental method of creating a known dataset to fill these requirements would be to use a mock microbial community, such as ZymoBIOMICS Microbial Community standards, to produce a time course with species at known abundance and then sequence. As the ZymoBIOMICS standards have already been sequenced by both Nanopore and Illumina sequencing (Nicholls *et al*. 2019), an easier and cheaper method would be to use this publicly available dataset and build a synthetic time-series with it.

Not only would this allow the CLUSTard results to be easily validated against known genomes and abundance levels, it would also enable further investigation into the effect different sequencing technologies have on the success of the clustering of CLUSTard. As only ten microbial species are present in the ZymoBIOMICS standard it is a small metagenome (Nicholls *et al*. 2019). It would be beneficial to also run CLUSTard on a large metagenome with a known truth to determine if the size of a metagenome has any effect on the clustering of CLUSTard. A large synthetic metagenome should be built in order to do this which could be produced using simulated reads from well characterised microbial genomes created by software such as DeepSimulator (Li *et al*. 2018) for Nanopore sequences and ART (Huang *et al*. 2012) for Illumina sequences.

The production of a synthetic metagenome would also enable analysis into other factors that would impact the success of clustering. One such thing would be the optimal number of timepoints necessary for meaningful results. Whilst CLUSTard has been run on fewer timepoints than the 80 available for the NAB dataset (for example the 18 timepoints used for the clustering of the Sharon *et al*. (2013) dataset), it would be beneficial to determine the minimum number of timepoints necessary for accurate clustering and also if there is a maximum number of timepoints before the clustering becomes too computationally intensive, or the results become confounded.

CONCOCT attempts to bin all input sequences given (Alneberg *et al*. 2014). Not only does this increase the computational challenge due to the larger dataset but it also results in lower quality clusters as erroneous or lower quality contigs are clustered with them. In CLUSTard, contigs are filtered out based on a given size threshold (recommended 1,000bp with a long-read assembly) and if they do not match to another contig. With the NAB dataset, the long-read polished CLUSTard run at a threshold of 0.997 binned only 4.51% of the contigs from the input assembly but 22.35% of the bases. This reduced dataset meant that the overall computation power necessary for downstream analysis was reduced whilst still capturing much of the diversity present in the assembly.

Due to the challenge of assembling large metagenomic datasets it is likely that many metagenomic assemblies are suboptimal, containing redundant or erroneous sequences. This means that clustering could be just as, if not more, effective when part of the assembly is excluded.

Metagenomic assemblies will not always contain redundancies and as metagenomic assemblers become more efficient the accuracy of the assembly will increase and become less redundancy. Along with the increase in the accuracy of contigs the length should decrease, this would reduce the number of contigs which would decrease overall computational time and increase the accuracy of clusters.

Whilst this method of clustering seems to ultimately be successful, some issues in the pipeline remain. First is the issue of ‘singletons’ - i.e. a single large contig that encompasses an entire genome. As the genome has been completely assembled into one contig there are no other sequences to correlate with. Therefore, despite its size this single sequence would not be recognised as a cluster and would remain unbinned by `CLUSTard` even though it could potentially be a complete genome. This appears to be the case in the NAB dataset with a contig of 1,303,796bp remaining unclustered. Steps should be integrated into the `CLUSTard` pipeline to account for this. As all unbinned sequences are outputted to a `FASTA` file by `CheckM`, it would be relatively simple to pull out all unbinned contigs over a certain size (say 500kbp) and treat these "singletons" as their own cluster. Then they could be run through the further analysis steps used in the pipeline (e.g. `Kraken` and `Prokka`) to determine if they are of biological importance and if so, what organisms they are. The abundance information that corresponds to these singletons could also be integrated with the further analysis information in order to include these singletons in the output plot. It may also be prudent to run these contigs through NCBI `Blast` (Altschul *et al*. 1990) to determine if they match to any known species. Some of these long unclustered contigs could be caused by long homopolymers which are a common issue in nanopore assemblies (Rang *et al*. 2018). If this was the case this would become evident in the analysis steps.

Another issue seen with the `CLUSTard` pipeline is the poor clustering of short reads, perhaps caused by the fact that they have fewer time-course raw reads mapping to them, but there's no obvious work around for this. This is not as much of an issue with long-read assemblies as less of the assembly will be in short contigs, however this would remain an issue with short-read input assemblies as the assembly is likely to be more fragmented (Goldstein *et al*. 2019). This links back to the earlier point - the quality of the assembly is the limiting factor in the quality of the bins.

The presence of small potentially fragmented clusters indicates that `CLUSTard` may not be clustering to the highest efficiency. This could be caused by any number of reasons, such as those previously outlined; poor quality assembly, the increased error in the contigs and the mix of sequencing technologies. Once those issues have been resolved or limited, any small clusters remaining should be investigated to determine if they are biologically relevant. It is possible that they are viral or phage genomes, which are diverse in size (Hatfull and

Hendrix 2011; Campillo-Balderas *et al*. 2015). To determine if this is the case it may be as simple as using another `Kraken` database that includes viral (especially phage) genomes in the pipeline. Alternatively integrating other software into the pipeline such as `MARVEL` (Amgarten *et al*. 2018) to identify clusters containing potential bacteriophage and then `PHANOTATE` (McNair *et al*. 2019) to annotate genes in any phage clusters. It is also possible that these small clusters are other DNA molecules, such as plasmids - which are clustering separately due to a difference in copy number. To ascertain this, any identified open reading frames (ORFs) in the clusters it should be investigated to determine if they are plasmid ORFs.

If, even after clustering as efficiently as possible and further analysis the small clusters prove not to be biologically important, it would be possible to merge clustering bins that share sequence characteristics, such as GC content or taxonomic classification, to obtain meaningful clusters.

Biology

Twenty high-quality draft MAGs were produced from the NAB dataset after it was run through `CLUSTard`. These 20 Mags encompass a diverse range of prokaryotic phyla. Some of these MAGs appear to be close in composition to other defined species yet some do not. Here, due to time and space constraints, only the results of a selection of clusters with well-defined related species and with genome characteristics (i.e. GC-content and size) that matched these related species are reported here. When closely related species are not known, any further analysis becomes much more complex.

Here, cluster c_03119 was investigated. This cluster was identified both by Kraken_GTDB and Silva as belonging to the *Nitrospira* genus, and the genome characteristics also matched those of the *Nitrospira* species identified. This cluster was then run through `autoMLST` in order to determine where the cluster sits within a *Nitrospira* tree. Here, the closest related genome was identified as *Nitrospira* sp. strain ND1. Here this cluster is much more abundant in the feed timepoints but not in the digesters. This is to be expected as while *Nitrospira* sp. strain ND1 had previously been identified in activated sludge as *Nitrospira*, it is nitrite-oxidising which is an aerobic process (Ushiki *et al*. 2017) and therefore unlikely to survive in an anaerobic environment.

Another cluster investigated was c_77547 which was identified as *Flavobacterium* by both Kraken and Silva, however the exact species remained undefined. This cluster was also run through `autoMLST` to determine related species, identifying *Flavobacterium aquatile* as the closest relative, however, it is possible that this cluster is a previously undefined species. The final cluster investigated in depth was c_63947, which was identified as *Candidatus Cloacimonas acidaminovorans* by both `Kraken` and `Silva`. *Candidatus Cloacimonas acidaminovorans* is thought to be widely present in many in anaerobic digesters, potentially decreasing the methane produced (Solli *et al*. 2014).

A lot of useful biological information remains untapped in the NAB dataset and should be further investigated in order to make sense of the microbial community. Much of this can be undertaken in the same way as the analysis of the 20 high-quality draft MAGs, identifying closely related genomes through `Kraken`, `Silva` and `autoMLST`. A further step would be to align the related genome and the cluster together to determine the evolutionary relationship between genomes at the nucleotide level. Where the identity of a cluster remains inconclusive further steps should be undertaken. It is possible that the closest species are not present in the databases used for `Kraken`, `Silva`, or `autoMLST`. This may be rectified by using a custom `Kraken` database or by using the feature that allows user defined sequences that are not present in the `autoMLST` database to be included in an `autoMLST` tree, for example MAGs identified by Kraken_GTDB.

A further step to undertake would be the production of "finished" MAGs - defined as "Single contiguous sequence without gaps or ambiguities with a consensus error rate equivalent to Q50 or better" (Bowers *et al*. 2017). This could be achieved by mapping the raw long reads back onto clusters, then pulling out the raw reads that map and only assembling them. This would reduce the computational demand on the assemblers and hopefully produce a better-quality assembly however would be a time-consuming process.

Whilst `CLUSTard` has produced 20 high-quality draft MAGs it has also facilitated the analysis of the microbial community as a whole. The composition of the microbial community remained very similar across all four digesters. As seen in figure 17, each of the four digesters had a very similar abundance profile. This relatively stable population is not unexpected, Kirkegaard *et al*. (2017) observed similarity in the microbial community across 32 full-scale anaerobic digesters over a six-year period. However, with the NAB dataset, the sampling strategy and lack of sequencing depth lends itself to capturing stable populations as it would miss any short-term population booms and rare species.

Overall the clusters recovered from the NAB dataset are very diverse, with 227 different species from 170 different genera present when GTDB taxonomy is used, or 202 different species from 139 genera when NCBI taxonomy is used.

The difference in the community make-up between the feed and the digesters is stark, sharing only *Flavobacterium* and *Streptomyces* to a high degree. This indicates that the clustering is robust even with large changes of community between the samples given. However, the low abundance of otherwise common organisms may be introducing noise into the clustering.

Kirkegaard *et al*. (2017) also found that much of the stable population seen in the anaerobic digesters is due to potentially inactive populations immigrating with the feed. This does not appear to be the case with the NAB dataset as the feed has very a different abundance pattern to that of the digesters, with the most abundant genera in the feed (*Dechloromonas*) not similarly abundant in the digesters. There are many things that could be causing this - including the fact that these anaerobic digesters were initially seeded using inoculum from an up-and-running AD community and that it is unlikely for the populations in the feed to displace these established communities. However, it has been shown that process operational conditions are the strongest driver of microbial communities over the initial inoculum (Peces *et al*., 2018). It is also likely that the feed contains aerobic species that would not survive in an AD environment or species that are not well adapted to the selective conditions present within the digesters, as was seen in cluster c_03119 which was identified as *Nitrospira*.

Whilst the community composition remains relatively stable, certain species do experience changes in abundance both over time and between digesters. The genus *Candidatus Cloacimonas* (UBA1032) experiences a large increase over time across all four digesters but is less abundant in digester four (NAB4). Digester four is the most divergent of the four digesters, although there is no obvious cause of this. Ideally, the changes in the biological community could be linked to changes in quantifiable physical properties such as gas production, gas composition, temperature or ammonia concentration, as these have been shown to be important factors in the composition of microbial communities and also indicate the health of the digesters (Kirkegaard *et al*. 2017). Process data of the anaerobic digesters is available for this dataset however there is a lot of missing data and gaps. Whilst filling these gaps using modelling would be possible it is beyond the scope of this project.

Despite the promising results seen in the NAB LR-pol clustering it is likely that the clusters produced have only captured the most abundant data. This dataset only has limited depth of sequencing with both the short reads and the long reads. This means that the lower abundant species will have fewer sequencing reads corresponding to them, therefore any sequence errors will be more likely to remain after assembly reducing the precision of the clustering for rare species (Sims *et al*. 2014). The fewer short reads mapping to contigs of rare species will also add challenges to the clustering. Along with this low depth of coverage, the long-read assembly was produced from only two time-points (T15 and T17). As these were late in the time-course it stands to reason that the CLUSTard run will only be capturing the species that are abundant late in the time-course and probably not any species that were abundant at the start of the time-course and then died out. This issue would be resolved by sequencing some earlier time points with nanopore technology and then re-assembling with all the data.

Conclusions

This research aimed to produce a simple method to both cluster large time-series metagenome datasets and to undertake reproducible further analysis. The `CLUSTard` pipeline produced is a fast metagenomic binning and analysis pipeline that is comparable in results to another popular metagenomic binning software. The pipeline allows for easy and reproducible metagenomic binning and analysis, requiring minimal user input as all software installation is handled by the workflow manager and the pipeline only requires one command to run all steps on both a local machine and a computing cluster. This workflow makes it simple for steps or for the whole pipeline to be reproduced. Ultimately this pipeline plays to the strength of the datasets our research group produces by utilising both the time-series information already present and also integrating both long and short raw reads when binning metagenomic-assembled genomes. The `CLUSTard` pipeline is quick to run even on large metagenomic assemblies, not only when producing metagenome bins but also when completing further analysis steps and producing summary plots and files. This further analysis undertaken enables a quick entry point to determine the composition and dynamics of the community. Whilst `CLUSTard` was ultimately produced to be a tool used in-house, it may be beneficial for other research groups to get the most out of their large time-series metagenomic datasets. The use of a workflow manager which enables the easy portability to other systems helps towards this goal.

The optimal Pearson's Correlation Coefficient value for a cut-off threshold for clustering was found to be 0.997. This threshold was ultimately a compromise between minimising data loss, as a higher threshold meant that many contigs did not have a high enough Pcc value and were excluded from clusters, and the overall quality of clusters, as a lower threshold meant that more contigs were erroneously clustered together.

After investigating the use of different input assemblies in `CLUSTard` it became apparent that the most important factor governing clustering success was the quality of the input assembly, with longer more accurate contigs having the most success being clustered together. Because of this it is to be recommended that `CLUSTard` is run with the best possible assembly available for the dataset. If only Illumina short-reads are available, then an assembly produced by `SPAdes` is the recommended input in order to increase the contiguity of the assembly. However, as seen with the NAB dataset used here, this may not always be possible due to computational restraints. If only Nanopore long reads are available, then an assembly produced by either `Canu` or `metaFlye` is recommended followed by consensus sequence polishing, e.g. by `Medaka (github.com/nanoporetech/medaka)` or `Nanopolish`. If both long and short reads are available, a long-read assembly produced by `Canu` or `metaFlye` which is then consensus polished by both the long reads (`Medaka`/`Nanopolish`) and the short-reads (`Pilon`) is recommended. This latter situation is the ideal assembly for `CLUSTard` - combining the long read length of the nanopore sequencing and the higher accuracy of the Illumina sequencing. As the nanopore technology develops and the error rate reduces further to be comparable with the accuracy of Illumina sequencing, it is probable that the need for consensus polishing will also reduce. As the optimum assembly method is specialised depending on the input data and due to the assembly process being computationally intensive the steps required for assembly will not be integrated into the `CLUSTard` pipeline.

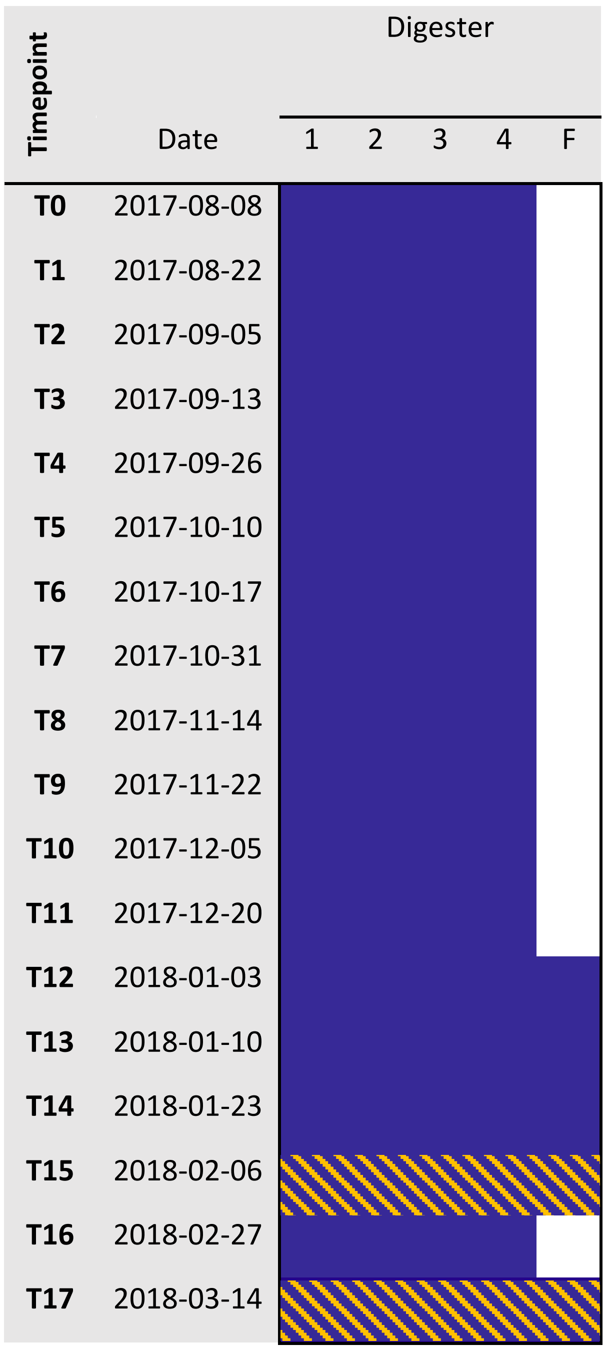
This pipeline enabled the definition of a large metagenomic dataset, determining the dynamics of the community and the production of many metagenome assembled genomes (MAGs), including 20 high-quality MAGs which could, with a little effort, become finished MAGs. Further research should link the community composition with both the metabolic function of the anaerobic digestion community with the process operation conditions to produce important information about how to maximise biogas production by harnessing the power of the microbial community.

Many of the improvements mentioned here are geared towards improving the quality of the genome bins produced. Although it is useful to gain high-quality MAGs for previously un-categorised organisms or those that prove a cultivation challenge, this is not the be all and end all of metagenomic studies. For many, the aim is to determine what is present in the community and sometimes what happens to the community over time. The production of the community relative abundance and cluster level abundance plots within the `CLUSTard` pipeline facilitates the analysis of time-course sequencing projects easily and rapidly.

At the time of this project, the lack of comparison or benchmarking studies of many of the common metagenomic or bioinformatic tools was a big issue. This was especially the case in regard to how well these tools deal with Nanopore sequencing, as this presents its own set of unique problems compared to Illumina sequencing (De Maio *et al*. 2019). This information is lacking for many of the common tools, including those used here such as `CheckM`, `Prokka` and `Kraken`, as Nanopore was not a popular sequencing technology at the time of their release and few benchmarking studies have been done since. Further studies should be done to validate the common metagenomic tools with nanopore sequencing. This along with a comparison of tools would be beneficial to researchers building analytical pipelines and the standardisation of metagenomic analysis as a whole.

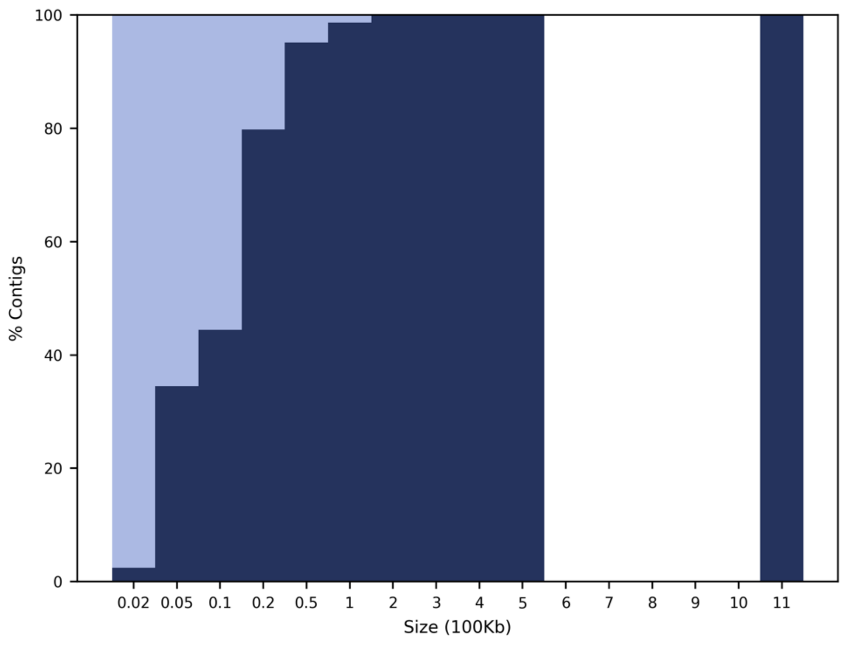
Appendices

Figures

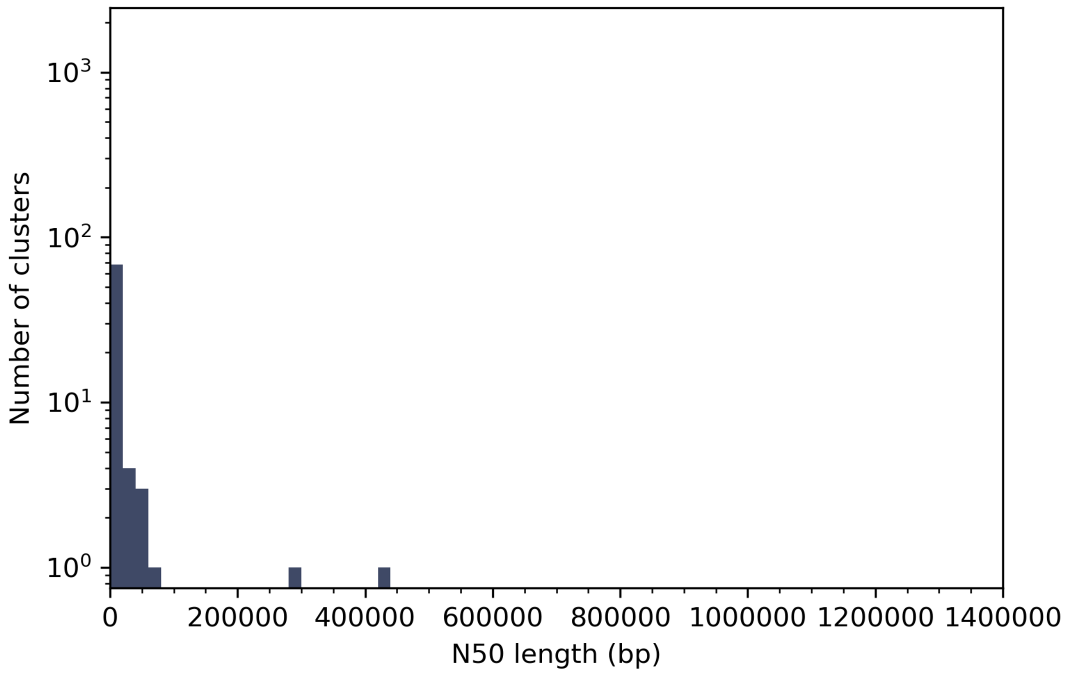


Timepoint	Date	Digester 1	Digester 2	Digester 3	Digester 4	Digester F
T0	2017-08-08	🟪	🟪	🟪	🟪	
T1	2017-08-22	🟪	🟪	🟪	🟪	
T2	2017-09-05	🟪	🟪	🟪	🟪	
T3	2017-09-13	🟪	🟪	🟪	🟪	
T4	2017-09-26	🟪	🟪	🟪	🟪	
T5	2017-10-10	🟪	🟪	🟪	🟪	
T6	2017-10-17	🟪	🟪	🟪	🟪	
T7	2017-10-31	🟪	🟪	🟪	🟪	
T8	2017-11-14	🟪	🟪	🟪	🟪	
T9	2017-11-22	🟪	🟪	🟪	🟪	
T10	2017-12-05	🟪	🟪	🟪	🟪	
T11	2017-12-20	🟪	🟪	🟪	🟪	
T12	2018-01-03	🟪	🟪	🟪	🟪	🟪
T13	2018-01-10	🟪	🟪	🟪	🟪	🟪
T14	2018-01-23	🟪	🟪	🟪	🟪	🟪
T15	2018-02-06	▨	▨	▨	▨	▨
T16	2018-02-27	🟪	🟪	🟪	🟪	
T17	2018-03-14	▨	▨	▨	▨	▨

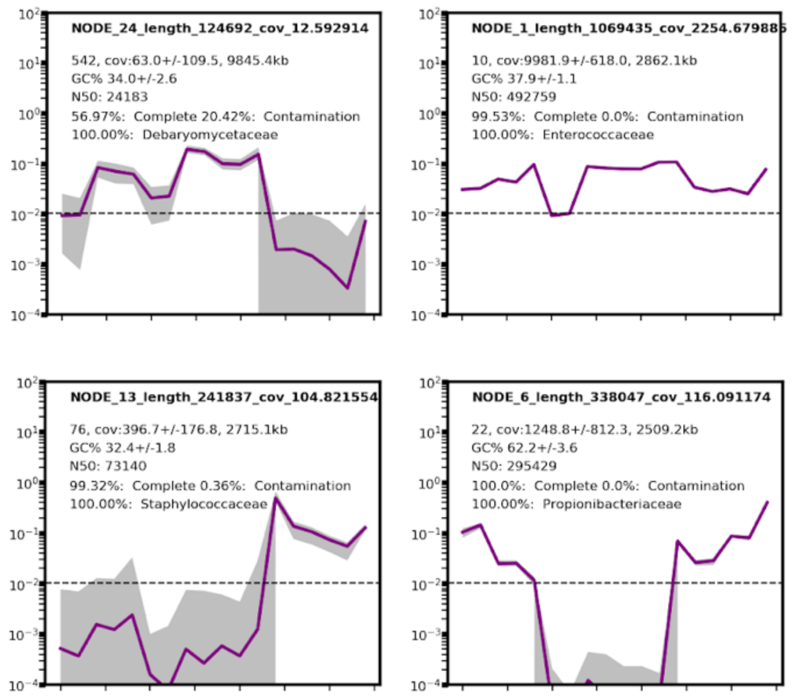
Ap. Figure 1: Diagram of the sampling strategy for the NAB data. Showing timepoints and the corresponding dates that samples were taken and from which digesters (1, 2, 3 or 4) or feed (F). Purple filled squares indicate an Illumina sequencing sample and diagonal orange lines over a purple filled square indicates both an Illumina and a Nanopore sequencing sample.



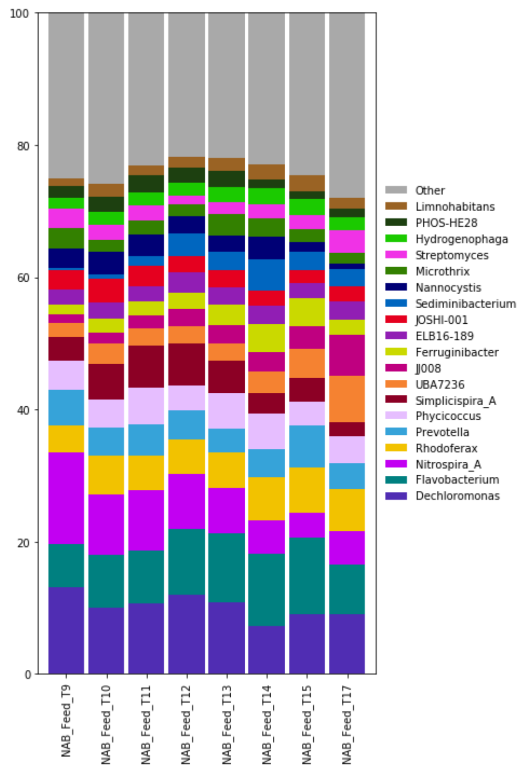
Ap. Figure 2: The percentage of different length contigs binned (dark blue) or unbinned (light blue) after the Sharon *et al*. (2013) dataset was run through `CLUSTard`. The gap between 600-1,000kbp indicates that there were no contigs between these lengths.



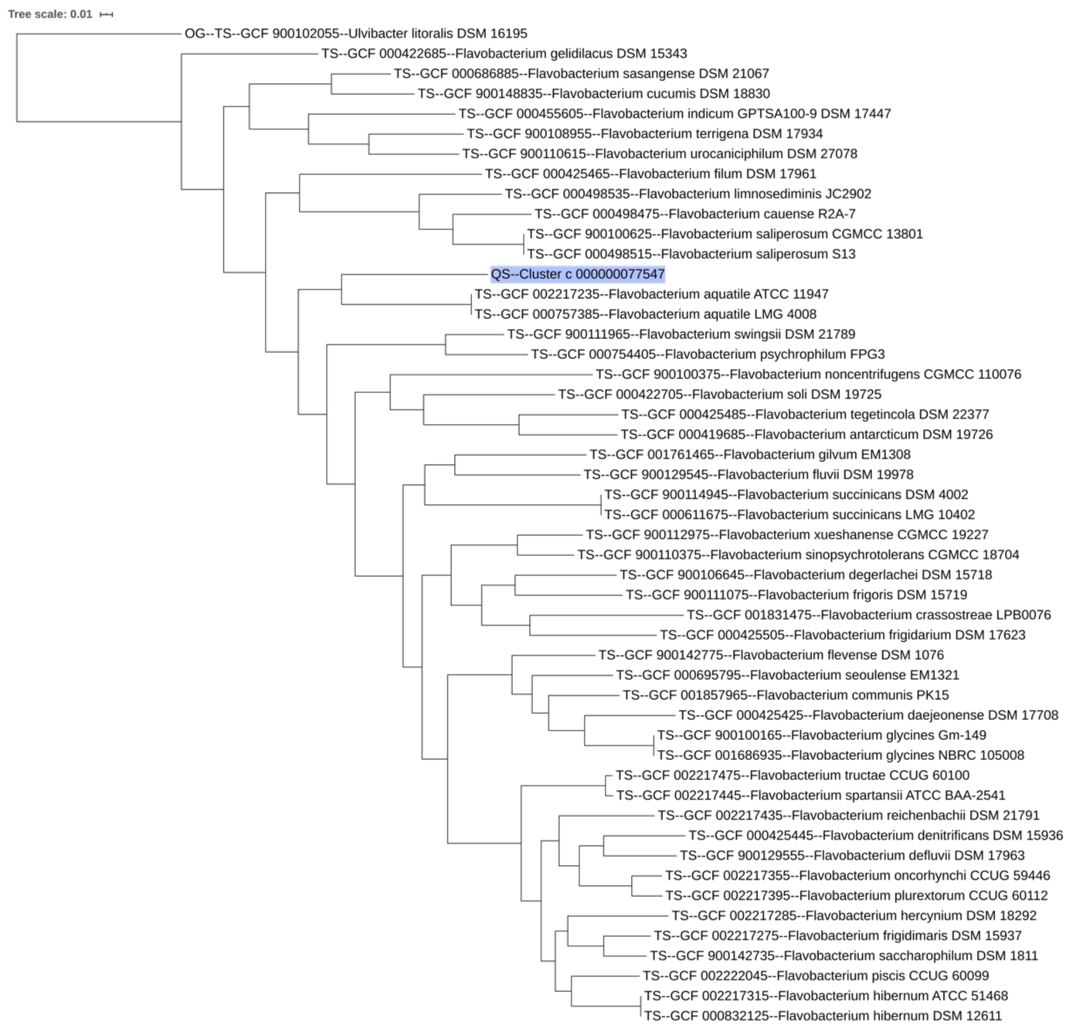
Ap. Figure 3: The distribution of the N50 length of clusters produced when the Sharon *et al*. (2013) dataset was run through `CLUSTard.`



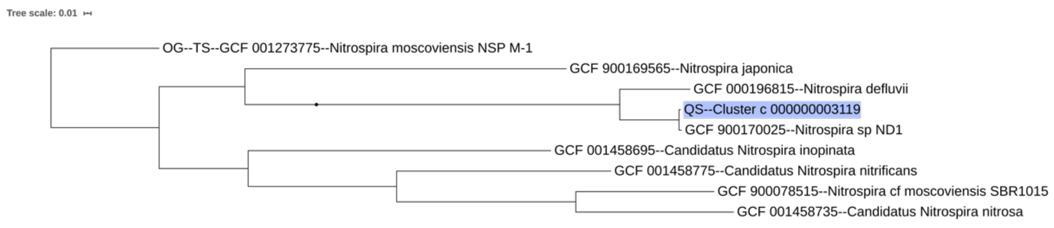
Ap. Figure 4: Overview plot of the four largest clusters produced when the Sharon *et al*. (2013) dataset was run through the CLUSTard pipeline. The first line on each plot is the cluster identity. The second line shows the number of contigs clustered, the coverage (±1SD) of the cluster and the size of the cluster in Kbp. The third line shows the GC content of the cluster (±1SD) and the fourth line N50 length. Then the completeness and contamination values are on the next line and the top Kraken identity on the final line. The plot themselves shows the relative abundance of the cluster across all time points. The grey area seen on the figures show the range in abundance values.



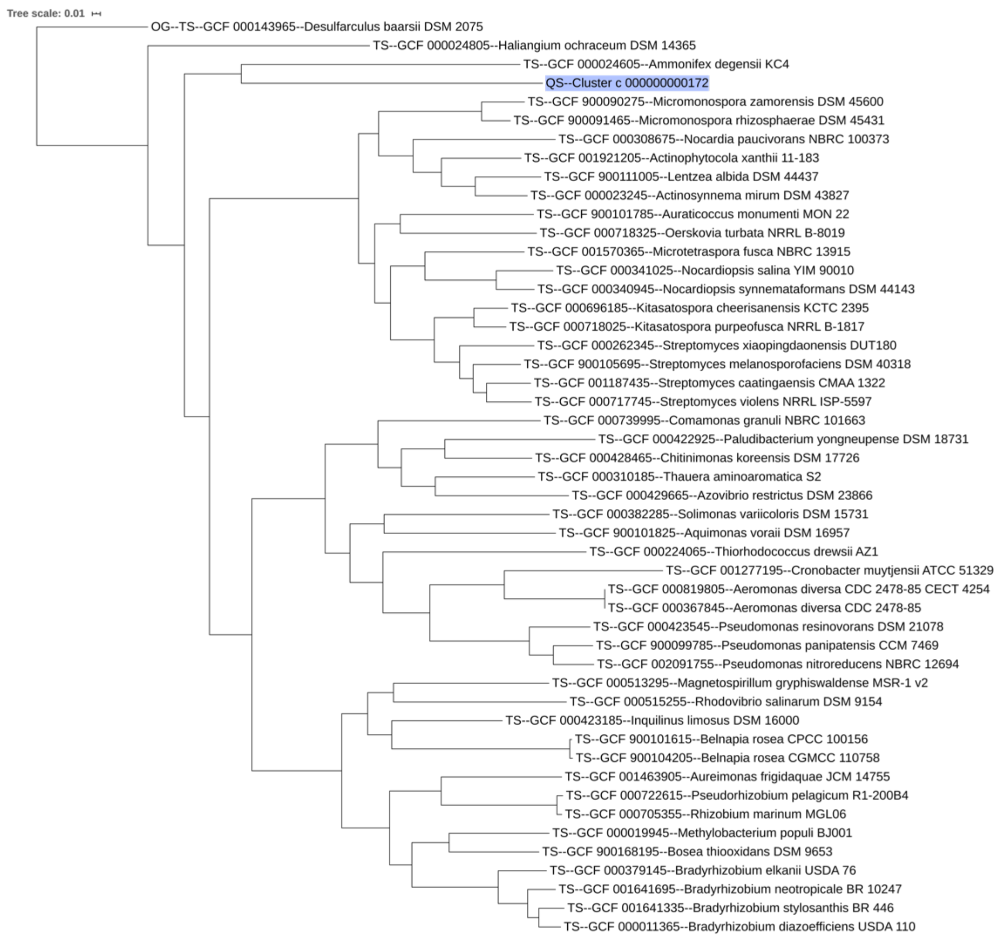
Ap. Figure 5 (right): Relative abundance plot of the feed only from the CLUSTard NAB LR-pol 0.997 run. The top 19 most common genera in the feed are shown in the plot, with all other genera contained in the *other* group (grey).


Tree scale: 0.01
OG--TS--GCF 900102055--Ulvibacter litoralis DSM 16195
TS--GCF 000422685--Flavobacterium gelidilacus DSM 15343
TS--GCF 000686885--Flavobacterium sasangense DSM 21067
TS--GCF 900148835--Flavobacterium cucumis DSM 18830
TS--GCF 000455605--Flavobacterium indicum GPTSA100-9 DSM 17447
TS--GCF 900108955--Flavobacterium terrigena DSM 17934
TS--GCF 900110615--Flavobacterium urocaniciphilum DSM 27078
TS--GCF 000425465--Flavobacterium filum DSM 17961
TS--GCF 000498535--Flavobacterium limnosediminis JC2902
TS--GCF 000498475--Flavobacterium cauense R2A-7
TS--GCF 900100625--Flavobacterium saliperosum CGMCC 13801
TS--GCF 000498515--Flavobacterium saliperosum S13
QS--Cluster c 000000077547
TS--GCF 002217235--Flavobacterium aquatile ATCC 11947
TS--GCF 000757385--Flavobacterium aquatile LMG 4008
TS--GCF 900111965--Flavobacterium swingsii DSM 21789
TS--GCF 000754405--Flavobacterium psychrophilum FPG3
TS--GCF 900100375--Flavobacterium noncentrifugens CGMCC 110076
TS--GCF 000422705--Flavobacterium soli DSM 19725
TS--GCF 000425485--Flavobacterium tegetincola DSM 22377
TS--GCF 000419685--Flavobacterium antarcticum DSM 19726
TS--GCF 001761465--Flavobacterium gilvum EM1308
TS--GCF 900129545--Flavobacterium fluvii DSM 19978
TS--GCF 900114945--Flavobacterium succinicans DSM 4002
TS--GCF 000611675--Flavobacterium succinicans LMG 10402
TS--GCF 900112975--Flavobacterium xueshanense CGMCC 19227
TS--GCF 900110375--Flavobacterium sinopsychrotolerans CGMCC 18704
TS--GCF 900106645--Flavobacterium degerlachei DSM 15718
TS--GCF 900111075--Flavobacterium frigoris DSM 15719
TS--GCF 001831475--Flavobacterium crassostreae LPB0076
TS--GCF 000425505--Flavobacterium frigidarium DSM 17623
TS--GCF 900142775--Flavobacterium flevense DSM 1076
TS--GCF 000695795--Flavobacterium seoulense EM1321
TS--GCF 001857965--Flavobacterium communis PK15
TS--GCF 000425425--Flavobacterium daejeonense DSM 17708
TS--GCF 900100165--Flavobacterium glycines Gm-149
TS--GCF 001686935--Flavobacterium glycines NBRC 105008
TS--GCF 002217475--Flavobacterium tructae CCUG 60100
TS--GCF 002217445--Flavobacterium spartansii ATCC BAA-2541
TS--GCF 002217435--Flavobacterium reichenbachii DSM 21791
TS--GCF 000425445--Flavobacterium denitrificans DSM 15936
TS--GCF 900129555--Flavobacterium defluvii DSM 17963
TS--GCF 002217355--Flavobacterium oncorhynchi CCUG 59446
TS--GCF 002217395--Flavobacterium plurextorum CCUG 60112
TS--GCF 002217285--Flavobacterium hercynium DSM 18292
TS--GCF 002217275--Flavobacterium frigidimaris DSM 15937
TS--GCF 900142735--Flavobacterium saccharophilum DSM 1811
TS--GCF 002222045--Flavobacterium piscis CCUG 60099
TS--GCF 002217315--Flavobacterium hibernum ATCC 51468
TS--GCF 000832125--Flavobacterium hibernum DSM 12611

Ap. Figure 6: Phylogenetic tree generated by `autoMLST` and visualised in `iTOL`, placing cluster c_077547 (highlighted in blue) into phylogenetic context.


Tree scale: 0.01
OG--TS--GCF 001273775--Nitrospira moscoviensis NSP M-1
GCF 900169565--Nitrospira japonica
GCF 000196815--Nitrospira defluvii
QS--Cluster c 000000003119
GCF 900170025--Nitrospira sp ND1
GCF 001458695--Candidatus Nitrospira inopinata
GCF 001458775--Candidatus Nitrospira nitrificans
GCF 900078515--Nitrospira cf moscoviensis SBR1015
GCF 001458735--Candidatus Nitrospira nitrosa

Ap. Figure 7: Phylogenetic tree generated by `autoMLST` and visualised in `iTOL`, placing cluster c_03119 (highlighted in blue) into phylogenetic context.


Tree scale: 0.01
OG--TS--GCF 000143965--Desulfarculus baarsii DSM 2075
TS--GCF 000024805--Haliangium ochraceum DSM 14365
TS--GCF 000024605--Ammonifex degensii KC4
QS--Cluster c 000000000172
TS--GCF 900090275--Micromonospora zamorensis DSM 45600
TS--GCF 900091465--Micromonospora rhizosphaerae DSM 45431
TS--GCF 000308675--Nocardia paucivorans NBRC 100373
TS--GCF 001921205--Actinophytocola xanthii 11-183
TS--GCF 900111005--Lentzea albida DSM 44437
TS--GCF 000023245--Actinosynnema mirum DSM 43827
TS--GCF 900101785--Auraticoccus monumenti MON 22
TS--GCF 000718325--Oerskovia turbata NRRL B-8019
TS--GCF 001570365--Microtetraspora fusca NBRC 13915
TS--GCF 000341025--Nocardiopsis salina YIM 90010
TS--GCF 000340945--Nocardiopsis synnemataformans DSM 44143
TS--GCF 000696185--Kitasatospora cheerisanensis KCTC 2395
TS--GCF 000718025--Kitasatospora purpeofusca NRRL B-1817
TS--GCF 000262345--Streptomyces xiaopingdaonensis DUT180
TS--GCF 900105695--Streptomyces melanosporofaciens DSM 40318
TS--GCF 001187435--Streptomyces caatingaensis CMAA 1322
TS--GCF 000717745--Streptomyces violens NRRL ISP-5597
TS--GCF 000739995--Comamonas granuli NBRC 101663
TS--GCF 000422925--Paludibacterium yongneupense DSM 18731
TS--GCF 000428465--Chitinimonas koreensis DSM 17726
TS--GCF 000310185--Thauera aminoaromatica S2
TS--GCF 000429665--Azovibrio restrictus DSM 23866
TS--GCF 000382285--Solimonas variicoloris DSM 15731
TS--GCF 900101825--Aquimonas voraii DSM 16957
TS--GCF 000224065--Thiorhodococcus drewsii AZ1
TS--GCF 001277195--Cronobacter muytjensii ATCC 51329
TS--GCF 000819805--Aeromonas diversa CDC 2478-85 CECT 4254
TS--GCF 000367845--Aeromonas diversa CDC 2478-85
TS--GCF 000423545--Pseudomonas resinovorans DSM 21078
TS--GCF 900099785--Pseudomonas panipatensis CCM 7469
TS--GCF 002091755--Pseudomonas nitroreducens NBRC 12694
TS--GCF 000513295--Magnetospirillum gryphiswaldense MSR-1 v2
TS--GCF 000515255--Rhodovibrio salinarum DSM 9154
TS--GCF 000423185--Inquilinus limosus DSM 16000
TS--GCF 900101615--Belnapia rosea CPCC 100156
TS--GCF 900104205--Belnapia rosea CGMCC 110758
TS--GCF 001463905--Aureimonas frigidaquae JCM 14755
TS--GCF 000722615--Pseudorhizobium pelagicum R1-200B4
TS--GCF 000705355--Rhizobium marinum MGL06
TS--GCF 000019945--Methylobacterium populi BJ001
TS--GCF 900168195--Bosea thiooxidans DSM 9653
TS--GCF 000379145--Bradyrhizobium elkanii USDA 76
TS--GCF 001641695--Bradyrhizobium neotropicale BR 10247
TS--GCF 001641335--Bradyrhizobium stylosanthis BR 446
TS--GCF 000011365--Bradyrhizobium diazoefficiens USDA 110

Ap. Figure 8: Phylogenetic tree generated by `autoMLST` and visualised in `iTOL`, placing cluster c_00172 (highlighted in blue) into phylogenetic context.

Tables

Ap. Table 1: Overview of the investigation into the 20 high-quality MAGs.

			Size (Mbp)			GC (%)			16S Match
	Top Kraken_DB species identity	**Top Kraken_GTDB species identity**	**Cluster**	**Kraken_DB**	**Kraken_GTDB**	**Cluster**	**Kraken_DB Ref**	**GTDB ref**	**Cluster silva match**
c_00172	9.09% Desulfococcus oleovorans	90.91% UBA2224 sp002348185	5.96	3.94	5.07	60.6 ±0.9	56.20	63.85	90.83% Hydrogenedentes
c_03119	95.83% Nitrospira defluvii	95.83% Nitrospira_A sp900170025	4.35	4.32	4.45	58.9 ±1.5	59.00	58.87	99.66% Nitrospira
c_68001	15.38% Flavisolibacter tropicus	30.77% Ferruginibacter sp002352045	4.07	5.94	3.46	37.8 ±0.9	41.50	37.67	95.34% Ferruginibacter
c_01295	94.44% Simplicispira suum	94.44% Simplicispira_A suum	3.80	4.15	4.15	64 ±0.8	63.29	63.29	All 3 >99.50% burkholderiaceae
c_63114	26.67% Draconibacterium orientale	100% UBA1413 sp002304925	3.65	5.13	3.37	49.2 ±2.1	41.31	49.83	(96.95% unclassified)
c_00486	6.25% Rhodopseudomonas palustris	100% UBA1062 sp002316295	3.61	5.51	3.13	57.5 ±1.2	64-66%	58.13	91.36% Deltaproteobacteria (SAR324 Clade)
c_00480	55.56% Phycicoccus dokdonensis	44.44% Phycicoccus jejuensis	3.45	3.94 (incomp)	3.99	70.7 ±0.8	71.20	73.54	97.47/97.64% Intrasporangiaceae
c_01929	3.7% Chlorobium phaeobacteroides	3.7% Aeromonas caviae	3.17	3.13/2.74 (strains)	2.76	37.9 ±1.2	48.4	61	82.91/84.02% Spirochaetes (V2072-189E03)
c_00599	16.67% Sphaerochaeta globosa	16.67% Flavobacterium alvei	3.17	3.32	3.82	37.1 ±0.6	48.9	34.39	88.4% Ignavibacteria (OPB56)
c_77547	50% Flavobacterium psychrophilum	86.36% Flavobacterium sp002296885	2.95	2.86	2.40	34.6 ±1.1	32.5	36.51	95.79% Flavobacterium
c_63587	13.33% Desulfovibrio vulgaris	26.67% UBA1062 sp001896555	2.86	3.77	3.31	59.3 ±1.5	63.24	57.7	(86.71% unclassified)
c_63297	25% Desulfococcus multivorans	12.5% Methanofastidiosum sp001587595	2.72	4.46	1.50	41.6 ±0.8	56.8	33.52	(94.2% unclassified)
c_63204	22.22% Defluviitoga tunisiensis	11.11% Agathobaculum sp900291975	2.66	2.05	3.21	52.3 ±0.6	31.4	56.40	79.15/.29/.37% Petrotogaceae (SC103)
c_000167	100% Methanothrix soehngenii	100% Methanothrix soehngenii	2.63	3.03	3.03	51.9 ±1.9	50.96	50.99	98.44/.54% Methanosaeta
c_63420	10% Syntrophobotulus glycolicus	100% UBA5389 sp002409965	2.50	3.41	3.23	59.4 ±1.3	46.40	59.90	87.14/86.76/86.65% Clostridia (DTU014)
c_63008	25% Bacteroides coprosuis	100% UBA5429 sp002427605	2.46	2.99	2.11	38.3 ±2.2	35.00	37.70	90.81% Bacteroidales (M2PB4-65 termite group)
c_76920	77.78% Sphaerochaeta globosa	100% Sphaerochaeta sp001604325	2.46	3.32	1.75	58.2 ±1.6	48.9	59.30	93.46/92.92/93.53% Sphaerochaeta
c_63228	10% Flavobacterium gilvum	30% Bact-19 sp002412425	2.28	4.40	2.80	33.6 ±0.8	35.2	30.73	(84.97 unclassified)
c_63947	100% Candidatus Cloacimonas acidaminovorans	100% Cloacimonas acidaminovorans	2.22	2.25	2.25	37.6 ±0.5	37.9	37.87	93.3/94.32/95.08% Candidatus Cloacimonas
c_64222	11.11% Heliobacterium modesticaldum	100% UBA1424 sp002329705	1.19	3.08	0.98	43.9 ±2.2	57.00	46.33	(98.96% unclassified)

Code

Snakefile

```
configfile: "config.yaml"

import pandas as pd
df_samples = pd.read_csv(config["samples"], sep ='\t', index_col = 0)
samples = df_samples["sample"].to_list()

JOBID = config["jobid"]
RAW_SR = config["RAW_SR"]
REFIN = config["REFIN"]
CONTIG_T = config["CONTIG_T"]
P_THRESH = config["P_THRESH"]
krakendb = config["krakendb"]
kraken_level = config["kraken_level"]
#for plotting
date_scale = config["date_scale"]
rel_or_abs = "a"
top20 = "n"

if 'y' in top20:
    out_abun = rel_or_abs + '_top20'
else:
    out_abun = rel_or_abs

subworkflow bwa_split:
    snakefile:
        "scripts/bwa_Snakefile"

subworkflow para:
    snakefile:
        "scripts/para_Snakefile"

subworkflow kraken2:
    snakefile:
        "scripts/kraken2_Snakefile"

rule all:
    input:
        expand("logs/{JOBID}_all_bwa_output.txt", JOBID=JOBID),
        expand("logs/{JOBID}_para_out.txt", JOBID = JOBID),
        expand("output/plots/1_{JOBID}_{kraken_level}_plot.png", JOBID = JOBID,
    kraken_level = kraken_level),
        expand("output/plots/{JOBID}_bin_contigs.png", JOBID = JOBID),
        expand("output/clustering/{JOBID}_read_counts_absolute.csv", JOBID = JOBID),
        expand("output/plots/{JOBID}_{out_abun}_abun_plot.png", JOBID = JOBID, out_abun
= out_abun),
        expand("output/{JOBID}_cluster_summary_stats.tsv", JOBID=JOBID)

localrules: test, para_out, plot, bin_plot, abs_derive, abun_plot

rule test:
    input:
        bwa_split(expand("output/clustering/{JOBID}_bwa_output.txt", JOBID = JOBID))
    output:
        "logs/{JOBID}_all_bwa_output.txt"
    shell:
        """
        more *.out > {output} 2> /dev/null
        rm *.out
        """
```

```
rule para_out:
    input:
        clusters = para(expand("logs/{JOBID}_para_done.txt", JOBID = JOBID))
    output:
        "logs/{JOBID}_para_out.txt"
    shell:
        """
        echo "Done" >> {output}
        """

rule plot:
    input:
        file_out = expand("logs/{JOBID}_para_out.txt", JOBID = JOBID),
        checkm = kraken2(expand("output/checkm/{JOBID}_checkm.log", JOBID=JOBID))
    output:
        cluster_plot = "output/plots/1_{JOBID}_{kraken_level}_plot.png"
    params:
        files = "plot_in_files.txt",
        sample_file = config["samples"],
        kraken = expand("output/kraken/{JOBID}_{kraken_level}_top_kraken.out", JOBID =
JOBID, kraken_level = kraken_level),
        date = date_scale,
        seqkit = expand("output/results/{JOBID}_seqkit_stats.tsv", JOBID = JOBID)
    conda:
        "envs/py3.yaml"
    shell:
        """
        ls -S output/results/Cluster*.fasta > {params.files}
        sed -i "s/.fasta/.csv/g" {params.files}
        python scripts/plot.py {params.files} {JOBID} {params.sample_file} {params.date}
-k {params.kraken} -k_l {kraken_level} -cm {input.checkm} -sk {params.seqkit}
        rm {params.files}
        """

rule bin_plot:
    input:
        file_out = expand("output/plots/1_{JOBID}_{kraken_level}_plot.png", JOBID =
JOBID, kraken_level = kraken_level)
    output:
        contig_plot = "output/plots/{JOBID}_bin_contigs.png"
    conda:
        "envs/py3.yaml"
    shell:
        """
        cd output/results/
        cat Cluster*.fasta | awk '$0 ~ ">" {{print c; c=0;printf substr($0,2,100)
"\\t"; }} $0 !~ ">" {{c+=length($0);}} END {{ print c; }}' | sort | uniq >
{JOBID}_sorted_lengths.tsv
        cd ../../
        python scripts/bin_plot.py output/results/{JOBID}_unbinned_contigs_stats.tsv
output/results/{JOBID}_sorted_lengths.tsv {JOBID}
        """

rule abs_derive:
    input:
        expand("output/plots/{JOBID}_bin_contigs.png", JOBID=JOBID)
    output:
        csv = "output/clustering/{JOBID}_read_counts_absolute.csv"
    params:
        thresh = CONTIG_T
    conda:
        "envs/py3.yaml"
    shell:
        """
        python scripts/absolute_derive.py clustering {JOBID} {params.thresh}
        """

rule abun_plot:
    input:
        count_in = expand("output/clustering/{JOBID}_read_counts_absolute.csv", JOBID =
JOBID)
    output:
        plot_out = "output/plots/{JOBID}_{out_abun}_abun_plot.png"
    conda:
        "envs/py3.yaml"
    params:
        roa = rel_or_abs,
        top_20 = top20,
        kraken_in = expand("output/kraken/{JOBID}_{kraken_level}_top_kraken.out", JOBID
= JOBID, kraken_level = kraken_level)
    shell:
        """
        cd output/results/
        for f in C*.fasta; do filename="${{f%%.*}}"; echo ">$f"; seqkit fx2tab -n $f;
done > {JOBID}_binned_cluster_contig.txt
        cd ../../
        python scripts/abun_plot.py {JOBID} {input.count_in}
output/results/{JOBID}_binned_cluster_contig.txt {params.roa} {params.top_20} -s
{samples} Coverage -k {params.kraken_in}
        """

rule clus_stats:
    input:
        expand("output/plots/{JOBID}_{out_abun}_abun_plot.png", JOBID=JOBID, out_abun =
out_abun)
    output:
        csv = "output/{JOBID}_cluster_summary_stats.tsv"
    conda:
        "envs/py3.yaml" #change clustering (below) when add counts folder..
    params:
        checkm = expand("output/checkm/{JOBID}_checkm.log", JOBID=JOBID),
        seqk = expand("output/results/{JOBID}_seqkit_stats.tsv", JOBID=JOBID)
    shell:
        """
        ls output/results/C*.csv > stat_input.txt
        python scripts/clus_stats.py stat_input.txt {JOBID} -cm {params.checkm} -sk
{params.seqk}
        rm stat_input.txt
        """
```

bwa_Snakefile

```
configfile: "config.yaml"

import pandas as pd
df_samples = pd.read_csv(config["samples"], sep ='\t', index_col = 0)
samples = df_samples["sample"].to_list()

JOBID = config["jobid"]
RAW_SR = config["RAW_SR"]
REFIN = config["REFIN"]
CONTIG_T = config["CONTIG_T"]
P_THRESH = config["P_THRESH"]
krakendb = config["krakendb"]
kraken_level = config["kraken_level"]

rule all:
    input:
        expand("{REFIN}.sa", REFIN=REFIN),
        expand('output/clustering/counts_{samples}.txt', samples=samples),
        expand('output/clustering/{jobid}_read_counts.out', jobid= JOBID),
        expand('output/clustering/{jobid}_read_counts_derived.csv', jobid= JOBID),
        expand('output/clustering/{jobid}_values.csv', jobid = JOBID),
        expand('output/clustering/{jobid}_diffs.csv', jobid = JOBID),
        expand("output/clustering/{JOBID}_bwa_output.txt", JOBID = JOBID),

localrules: merge_filecounts, derive, start_feeder, split_file

rule bwa_index:
    input:
        ref = REFIN
    output:
        '{REFIN}.sa'
    threads: 20
    shell:
        """
        module load bio/BWA
        bwa index {input.ref}
        """

rule bwa_mem:
    input:
        fq1 = 'data/{samples}_R1.fastq.gz',
        fq2 = 'data/{samples}_R2.fastq.gz',
        ref = REFIN,
        ref_ind = expand("{reference}.sa", reference=REFIN) #waits for indexed reference
    output:
        counts = 'output/clustering/counts_{samples}.txt'
    params:
        bam = 'output/alignment/{samples}.bam'
    threads: 20
    shell:
        """
        module load bio/BWA
        module load bio/SAMtools
        mkdir -p output/alignment
        bwa mem -M -t {threads} {input.ref} {input.fq1} {input.fq2} | samtools view -buS
- | samtools sort -o {params.bam}
        samtools index {params.bam}
        samtools idxstats {params.bam} > {output.counts}
        """

rule merge_filecounts:
    input:
        test = expand('output/clustering/counts_{SAMPLES}.txt', SAMPLES = samples)
    output:
        txt = 'output/clustering/{JOBID}_read_counts.out'
    conda:
        "../envs/py3.yaml"
    shell:
        """
        python scripts/merge_filecounts.py clustering {JOBID} -l {samples}
        """

rule derive:
    input:
        expand('output/clustering/{JOBID}_read_counts.out', JOBID=JOBID)
    output:
        csv = "output/clustering/{JOBID}_read_counts_derived.csv"
    params:
        thresh = CONTIG_T
    conda:
        "../envs/py3.yaml" #change clustering (below) when add counts folder..
    shell:
        """
        python scripts/derive.py clustering {JOBID} {params.thresh}
        """

rule start_feeder:
    input:
        expand('output/clustering/{JOBID}_read_counts_derived.csv', JOBID=JOBID)
    output:
        values = "output/clustering/{JOBID}_values.csv",
        diffs = "output/clustering/{JOBID}_diffs.csv"
    conda:
        "../envs/py3.yaml"
    shell:
        """
        python scripts/start_feeder.py clustering {JOBID}
        """

rule split_file:
    input:
        diffs = expand("output/clustering/{JOBID}_diffs.csv", JOBID=JOBID)
    output:
        touch("output/clustering/{JOBID}_bwa_output.txt")
    params:
        diffs = expand("output/clustering/{JOBID}_diffs", JOBID = JOBID)
    shell:
        """
        split -d -l 10000 --additional-suffix=.csv {input.diffs} {params.diffs}
        """
```

para_Snakefile

```
configfile: "config.yaml"

import pandas as pd
df_samples = pd.read_csv(config["samples"], sep ='\t', index_col = 0)
samples = df_samples["sample"].to_list()

JOBID = config["jobid"]
RAW_SR = config["RAW_SR"]
REFIN = config["REFIN"]
CONTIG_T = config["CONTIG_T"]
P_THRESH = config["P_THRESH"]
krakendb = config["krakendb"]
kraken_level = config["kraken_level"]

(job, part) = glob_wildcards('output/clustering/{JOBID}_diffs{PART}.csv')

rule all:
    input:
      expand("output/clustering/{JOBID}_output_{PART}.csv", JOBID = JOBID, PART = part),
      expand("output/clustering/{JOBID}_parallel_sets_{PART}.csv", JOBID = JOBID, PART =
part),
      expand("output/clustering/{JOBID}_parallel_merged.out", JOBID = JOBID),
      expand("output/clustering/{JOBID}_non_red_list.out", JOBID = JOBID),
      expand("logs/{JOBID}_para_done.txt", JOBID=JOBID)

localrules: para_sets, non_red_step, file_parser

rule bin_feeder:
    input:
        diffs = 'output/clustering/' + JOBID + '_diffs{PART}.csv'
    output:
        all = "output/clustering/" + JOBID + "_output_{PART}.csv",
    params:
        thresh = P_THRESH,
        all_diffs = expand("output/clustering/{JOBID}_diffs.csv", JOBID = JOBID)
    conda:
        "../envs/py3.yaml"
    shell:
        """
        python scripts/bin_feeder.py {input.diffs} {params.all_diffs} {params.thresh}
{output.all}
        """

rule para_sets:
    input:
        bins = "output/clustering/" + JOBID + "_output_{PART}.csv"
    output:
        "output/clustering/" + JOBID + "_parallel_sets_{PART}.csv"
    params:
        thresh = P_THRESH
    conda:
        "../envs/py3.yaml"
    shell:
        """
        python scripts/para_sets.py {input.bins} {output} {params.thresh}
        """

rule para_merge:
    input:
        expand("output/clustering/{JOBID}_parallel_sets_{PART}.csv", JOBID=JOBID, PART =
part)
    output:
        "output/clustering/{JOBID}_parallel_merged.out"
    resources:
        mem_mb = 64000
    conda:
        "../envs/py3.yaml"
    shell:
        """
        python scripts/parallel_merge_step2.py -i {input} -o {output}
        """

rule non_red_step:
    input:
      expand("output/clustering/{JOBID}_parallel_merged.out", JOBID = JOBID)
    output:
      expand("output/clustering/{JOBID}_non_red_list.out", JOBID = JOBID)
    conda:
      "../envs/py3.yaml"
    shell:
      """
      python scripts/step3.py {input} {output}
      """

rule file_parser:
    input:
        expand("output/clustering/{JOBID}_non_red_list.out", JOBID = JOBID)
    output:
        touch("logs/{JOBID}_para_done.txt")
    params:
        contigs = REFIN,
        csv = expand("output/clustering/{JOBID}_read_counts_derived.csv", JOBID =
JOBID),
        wd = "results/",
        header = samples
    conda:
        "../envs/py3.yaml"
    shell:
        """
        mkdir -p output/{params.wd}
        python scripts/file_parser.py {params.contigs} {params.csv} {input} {params.wd}
-l {params.header}
        """
```

kraken2_Snakefile

```
configfile: "config.yaml"

import pandas as pd
df_samples = pd.read_csv(config["samples"], sep ='\t', index_col = 0)
samples = df_samples["sample"].to_list()

JOBID = config["jobid"]
RAW_SR = config["RAW_SR"]
REFIN = config["REFIN"]
CONTIG_T = config["CONTIG_T"]
P_THRESH = config["P_THRESH"]
krakendb = config["krakendb"]
kraken_level = str(config["kraken_level"])

(CLUSTERS,) = glob_wildcards("output/results/Cluster_{CLUSTER}.csv")

rule all:
    input:
        expand("output/kraken/{JOBID}_Cluster_{CLUSTERS}_kraken.out", JOBID = JOBID,
CLUSTERS = CLUSTERS),
        expand("output/kraken/{JOBID}_{kraken_level}_top_kraken.out", JOBID = JOBID,
kraken_level = kraken_level),
        "tbl2asn_update.out",
        expand("output/prokka/Cluster_{CLUSTERS}/{JOBID}_{CLUSTERS}.err", JOBID = JOBID,
CLUSTERS = CLUSTERS),
        expand("logs/{JOBID}_slurm_prokka.log", JOBID=JOBID),
        "output/results/{JOBID}_seqkit_stats.tsv",
        expand("output/checkm/{JOBID}_checkm.log", JOBID=JOBID)

localrules: kraken_merge, tbl2asn, output, seqkit

rule kraken:
    input:
        "output/results/Cluster_{CLUSTERS}.fasta"
    output:
        report = "output/kraken/{JOBID}_Cluster_{CLUSTERS}_report_kraken.out",
    params:
        db = krakendb,
        output = "output/kraken/{JOBID}_Cluster_{CLUSTERS}_kraken.out"
    conda:
        "../envs/kraken2.yaml"
    threads:
        16
    resources:
        mem_mb = 4000
    shell:
        """
        kraken2 -db {params.db} --threads {threads} --report {output.report} --output
{params.output} --use-names {input}
        """

rule kraken_merge:
    input:
        report = expand("output/kraken/{JOBID}_Cluster_{CLUSTERS}_report_kraken.out",
JOBID = JOBID, CLUSTERS = CLUSTERS)
    output:
        "output/kraken/{JOBID}_{kraken_level}_top_kraken.out"
    params:
        level = {kraken_level}
    shell:
        """
        cd output/kraken
        find -name '{JOBID}*_report_kraken.out' -type f -printf '\\n%p\\t' -exec sh -c
'echo {{}} | sort -k1nr {{}} | grep -P "\\t{params.level}\\t" | head -n1 ' \\; >
{JOBID}_{kraken_level}_top_kraken.out
        """

rule tbl2asn:
    input:
        expand("output/kraken/{JOBID}_{kraken_level}_top_kraken.out", JOBID = JOBID,
kraken_level = kraken_level)
    output:
        touch("tbl2asn_update.out")
    conda:
        "../envs/prokka.yaml"
    shell:
        """
        cd $(dirname $(which tbl2asn))
        rm tbl2asn
        wget https://github.com/tseemann/prokka/raw/master/binaries/linux/tbl2asn
        chmod +x tbl2asn
        """

rule prokka:
    input:
        clusters = "output/results/Cluster_{CLUSTERS}.fasta",
        wait = "tbl2asn_update.out"
    output:
        file = "output/prokka/Cluster_{CLUSTERS}/{JOBID}_{CLUSTERS}.err"
    params:
        dir = "output/prokka/Cluster_{CLUSTERS}/",
        prefix = "{JOBID}_{CLUSTERS}",
        prokka = "output/results/Cluster_{CLUSTERS}_short.fasta"
    conda:
        "../envs/prokka.yaml"
    threads:
        20
    shell:
        """
        awk '/^>/{{print substr($1,1,21); next}}{{print}}' < {input.clusters} >
{params.prokka}
        prokka {params.prokka} --outdir {params.dir} --prefix {params.prefix} --cpus
{threads} --force
        rm {params.prokka}
        """

rule output:
    input: expand("output/prokka/Cluster_{CLUSTERS}/{JOBID}_{CLUSTERS}.err", JOBID =
JOBID, CLUSTERS=CLUSTERS)
    output:
        "logs/{JOBID}_slurm_kraken2SM.log"
    shell:
        """
        cat *.out > {output}
        rm *.out
        """

rule seqkit:
    input: wait = expand("logs/{JOBID}_slurm_kraken2SM.log", JOBID = JOBID)
    output:
          "output/results/{JOBID}_seqkit_stats.tsv"
    conda:
        "../envs/py3.yaml"
    threads:
        10
    shell:
        """
        seqkit stats -a -T -j {threads} output/results/*.fasta > {output}
        """

rule checkm:
    input:
        expand("output/results/Cluster_{CLUSTERS}.fasta", CLUSTERS = CLUSTERS),
        expand("output/results/{JOBID}_seqkit_stats.tsv", JOBID=JOBID)
    output:
        expand("output/checkm/{JOBID}_checkm.log", JOBID=JOBID)
    params:
        out = expand("output/checkm", JOBID=JOBID),
        input = "output/results",
        refin = REFIN
    threads:
        20
    #conda:
    #    "../envs/checkm.yaml"
    shell:
        """
        module load bio/CheckM
        module load math/numpy
        module load lang/Python/2.7.15-foss-2018b
        checkm unbinned -x fasta output/results/ {params.refin}
output/results/{JOBID}_unbinned_contigs.fa
output/results/{JOBID}_unbinned_contigs_stats.tsv
        checkm lineage_wf -f {output} --tab_table -x fasta -t {threads} {params.input}
{params.out}
        """
```

merge_filecounts.py

```
#!/usr/bin/env python3
import os
import glob
import argparse

def firstFile(fileName):
        bits = fileName.readline().rsplit('\t',1)
        return (bits[0])

def otherFile(fileName):
        bits = fileName.readline().split('\t')
        return (bits[2])

parser = argparse.ArgumentParser(description='')
parser.add_argument('loc', help='location count files are in', type=str)
parser.add_argument('jobid', help='location count files are in', type=str)
parser.add_argument('-l', '--sample-list', dest='samples', nargs='+', default=[])
args = parser.parse_args()
loc = str("output/" + args.loc)
jobid = args.jobid
samples = args.samples

fname = []

for i in samples:
    fname.append(str(loc + '/counts_' + i + '.txt')) #change this to read in in the
right order
print ('files with data to be merged: '+str(fname))
# count lines in one file (they should all be the same...)
count = 0
```

```
f = open(fname[0], 'r')
for line in f:
        count +=1
f.close()

# count is now set to the length of the file
print ('number of entries per file: '+str(count))
# now need to read in all files
filedata = [open(file_name, 'r') for file_name in fname]
oo = open(str(loc + '/' + jobid + '_read_counts.out'), 'w')

for runthrough in range (0,count):
        start_marker = filedata[0]
        end_marker = filedata[-1]
        string_text = firstFile(start_marker)+'\t'
        for line_out in filedata[1:-1]:
                string_text = string_text + otherFile(line_out)+'\t'
        string_text = string_text + otherFile(end_marker)+'\n'
        oo.write(string_text)

for closer in filedata:
        closer.close()

print ('merged data saved in: '+jobid+'_read_counts.out')
```

derive.py

```
#!/usr/bin/env python3
# function that adds all the count values together to get a total
def summer(x):
    total = 0
    for loop in x:
        total = total + int(loop)
    return(total)
# function that divides each point by total counts for this contig
def deriver(x,y):
    answers=[]
    for loop in x:
        if y == 0:
            y = 1
        value = int(loop)/y
        answers.append(value)
    return(answers)

import csv as csv
import os
import argparse

parser = argparse.ArgumentParser(description='')
parser.add_argument('loc', help='location count files are in', type=str)
parser.add_argument('jobid', help='jobid - to name output', type=str)
parser.add_argument('thresh', help='minimum size of contig', type=int)
args = parser.parse_args()
loc = str("output/" + args.loc)
jobid = args.jobid
thresh = args.thresh

dir_name = str(loc + '/')
file_name = str(jobid + '_read_counts.out')

new_record=[[]]
nr=False

with open(dir_name+file_name, 'r') as data_store:
    line = csv.reader(data_store, delimiter='\t')
    for i in line:
        if int(i[1]) >= thresh:
            counts = summer(i[2:])
            values = deriver((i[2:]), counts)
            coverage = (counts*150)/int(i[1])
            values.append(coverage)  # add coverage to the end of the entry
            values.insert(0,i[0])    # add contig name to the front of the entry
            if nr:
                new_record.append(values)
            else:                    # identifies first entry in the list
                new_record = [values]
                nr = True

with open(dir_name + '/' +  jobid + '_read_counts_derived.csv', 'w') as f:
    writer = csv.writer(f)
    writer.writerows(new_record)
```

start_feeder.py

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
import pandas as pd
import argparse

parser = argparse.ArgumentParser(description='')
parser.add_argument('loc', help='location count files are in', type=str)
parser.add_argument('jobid', help='location count files are in', type=str)
args = parser.parse_args()
loc = str("output/" + args.loc)
jobid = args.jobid

df = pd.read_csv(loc + '/' + jobid + '_read_counts_derived.csv', header=None, index_col
= 0)

df2 = df #seem to need to have a copy of the df to calc mean

df = df.drop(df.columns[len(df.columns)-1], axis=1) # drop last column so don't include
it in stats - is still in df2..

df2["mean"] = df.mean(axis=1)
df2["sd"] = df.std(ddof = 1, axis=1)

diffs = df.sub(df.mean(axis=1), axis=0)

diffs.to_csv(loc + '/' + jobid + '_diffs.csv', header = False) #diffs
df2.to_csv(loc + '/' + jobid + '_values.csv', header = False) #diffs
```

bin_feeder.py

```
#!/usr/bin/env python3

import sys, getopt
import csv
import numpy as np
import argparse
import pandas as pd

def pcc(x, y):
       prod = np.sum(np.multiply(x,y))
       divx = np.sqrt(np.sum(np.square(x)))
       divy = np.sqrt(np.sum(np.square(y)))
       result = prod/(divx*divy)
       return result

parser = argparse.ArgumentParser(description='')
parser.add_argument('cut_diffs', help='split diffs', type=str)
parser.add_argument('all_diffs', help='all diffs', type=str)
parser.add_argument('thresh', help='pr threshold', type=float)
parser.add_argument('output', help='output', type=str)
args = parser.parse_args()
cut_diffs = args.cut_diffs
diffs = args.all_diffs
write_file = args.output
thresh = args.thresh

df_diffs_all = pd.read_csv(diffs, header=None, index_col = 0)
df_diffs_cut = pd.read_csv(cut_diffs, header=None, index_col = 0)

with open(write_file, 'w') as sender:
    for contig_x, row in df_diffs_cut.iterrows():
        row = row.to_numpy()
        line_2 = int(np.where(df_diffs_all.index == contig_x)[0])
        for contig_y, row1 in df_diffs_all.iloc[line_2+1:].iterrows():
            row1 = row1.to_numpy()
            resp_val = pcc(row, row1)
            if resp_val >= thresh:
                line_out = (contig_x, contig_y, resp_val)
                writer = csv.writer(sender)
                writer.writerow(line_out)
            line_2 += 1
```

para_sets.py

```
#!/usr/bin/env python3

import sys, getopt
import csv
import json
import argparse

short_list = []
nr_list = []
final_list = []

parser = argparse.ArgumentParser(description='')
parser.add_argument('input', help='location of input', type=str)
parser.add_argument('output', help='location of output', type=str)
parser.add_argument('thresh', help='location of output', type=float)
args = parser.parse_args()
read_file = args.input
write_file = args.output
thresh = args.thresh

with open (read_file, 'r') as incoming:
       file_reader = csv.reader(incoming, delimiter=',')
       for row in file_reader:
              if float(row[2]) >= thresh:
                     short_list.append(row[:2])

while short_list:
       top = short_list[0]
       first = set(top)
       short_list.remove(top)
       for entry in short_list:
```

```
		if first & set(entry):
			first.update(entry)
			short_list.remove(entry)
	x = [list(set(first))]	# convert set to list to make compatible with json
	final_list.extend(x)

with open(write_file, 'w') as outgoing:
	json.dump(final_list, outgoing)
```

parallel_merge_step2.py

```
#!/usr/bin/env python3

import sys, getopt
import csv
import json
import argparse

def set_default(obj):
    if isinstance(obj, set):
        return list(obj)
    raise TypeError

short_list = []
nr_list = []
final_list = []

parser = argparse.ArgumentParser(description='')
parser.add_argument('-i', '--input-list', dest='input', nargs='+', default=[])
parser.add_argument('-o', dest='output', help='location of output', type=str)
args = parser.parse_args()
list_files = args.input
write_file = args.output

# open first file
first_file = list_files[0]

with open (first_file, 'r') as master:
#	print('opening '+str(master))
	master_list = json.load(master)
	l = master_list
	out = []
	while len(l)>0:
		first, *rest = l
		first = set(first)
		lf = -1
		while len(first)>lf:
			lf = len(first)
			rest2 = []
			for r in rest:
				if len(first.intersection(set(r)))>0:
					first |= set(r)
				else:
					rest2.append(r)
				rest = rest2
			out.append(first)
		l = rest
	master_list = out

# open sequential files and merge into the master list if they match
for current_f in list_files[1:]:
	with open (current_f, 'r') as working_file:
		working_list = json.load(working_file)
```

```
              for row_1 in master_list:
                     x = set(row_1)
                     for row_2 in working_list:
                            if x & set(row_2):
                                   x.update(row_2)
                                   working_list.remove(row_2)
       for entries in working_list: # add any sets left to the end of the master list
              y = [list(set(entries))]
              master_list.extend(y)

l = master_list
out = []
while len(l)>0:
          first, *rest = l
          first = set(first)
          lf = -1
          while len(first)>lf:
                        lf = len(first)
                        rest2 = []
                        for r in rest:
                                if len(first.intersection(set(r)))>0:
                                        first |= set(r)
                                else:
                                        rest2.append(r)
                                rest = rest2
                        out.append(first)
          l = rest
master_list = out

with open(write_file, 'w') as outgoing:
       json.dump(master_list, outgoing, default=set_default)
```

step3.py

```
#!/usr/bin/env python3

# STEP 3 STARTS HERE:
# make a non-redundant list from the sets
# generated in step 2

import json
import argparse

parser = argparse.ArgumentParser(description='')
parser.add_argument('input', help='location of input', type=str)
parser.add_argument('output', help='location of output', type=str)
args = parser.parse_args()
input_file = args.input
output_file = args.output

final_list = []
with open(input_file ,'r') as in_file:
       master_list = json.load(in_file)

while True:
       test = master_list[0]
       working_list = test
       for test_list in master_list[1:]:
              if not (set(test_list).intersection(test)):
                     a = False
              else:
                     a = True
              if a == False:
```

```
                        next
                else:
                        working_list.extend(test_list)
                        master_list.remove(test_list)
        master_list.remove(test)
        x = set(working_list)
        x = list(x)
        final_list.append(x)
        if master_list == []:
                break

with open(output_file, 'w') as out_file:
        json.dump(final_list, out_file)
```

file_parser.py

```
#!/usr/bin/env python3
# code requirements
import json
import re
import csv as csv
import argparse
from Bio import SeqIO
from Bio import SeqUtils as su
from Bio.Seq import Seq
from Bio.SeqRecord import SeqRecord

# file names - change these as required
parser = argparse.ArgumentParser(description='')
parser.add_argument('contigs', help='the files', type=str)
parser.add_argument('csv', help='read_counts_derived', type=str)
parser.add_argument('clusters', help='output from step3', type=str)
parser.add_argument('output', help='output directory', type=str)
parser.add_argument('-l', '--header-list', dest='header', nargs='+', default=[])
args = parser.parse_args()
contig_file = args.contigs
csv_file = args.csv
cluster_file = args.clusters
wd = str("output/" + args.output)
header = args.header

#add context to header columns
header = ['contig'] + header + ['cover', 'length', 'GC']
print(header) #testing...

# dictionaries and lists

cluster_stats = []     # list of stats on cluster data for export to .csv
bun_dict = {}

# open .fasta file containing contigs and store as dict
print('Opening contig sequence file')
contig_dict = SeqIO.to_dict(SeqIO.parse(contig_file, "fasta"))
print(str(len(contig_dict))+' sequences loaded')

# open .csv file and store as list(?)
print('Loading abundance data from .csv file')

with open(csv_file, 'r') as abundance:
    bun_entry = csv.reader(abundance)
    bun_list = list(bun_entry)
    for bun_record in bun_list:
        bun_dict[bun_record[0]] = bun_record[1:]
```

```
print('Loading cluster details')

with open(cluster_file, 'r') as clusters:
    working_cluster = json.load(clusters)
    for current_cluster in working_cluster:
        cluster_filename = (wd+'Cluster_'+str(current_cluster[0])+'.csv')
        fasta_cluster_filename = (wd+'Cluster_'+str(current_cluster[0])+'.fasta')
        fasta_entry = []
        with open(cluster_filename, 'w', newline='') as csvfile:
            csv_writer = csv.writer(csvfile, delimiter=',', quoting=csv.QUOTE_NONE,
escapechar=' ')
            csv_writer.writerow(header)
            for cluster_name in current_cluster:
                csv_string = cluster_name, ', '.join(map(str, bun_dict[cluster_name])),
len(contig_dict[cluster_name]),su.GC(contig_dict[cluster_name].seq)
                csv_writer.writerow(csv_string)
                fasta_entry.append(contig_dict[cluster_name])
            SeqIO.write(fasta_entry, fasta_cluster_filename, 'fasta')
```

plot.py

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
import statistics
import matplotlib
#matplotlib.use('pdf')
#%matplotlib inline
import matplotlib.gridspec as gsp
import matplotlib.pyplot as plt
import pandas as pd
import argparse
import re
import datetime
from pandas.plotting import register_matplotlib_converters
register_matplotlib_converters()


# ==============================================================================
# Command line parsing
# ==============================================================================
parser = argparse.ArgumentParser(description='usage = python in_file prefix -k
kraken_file -k_l kraken_level -cm checkm -sk seqkit_file samples_file date(y/n)')
parser.add_argument('in_file', help='the name of the file containing a list of csv
files', type=str)
parser.add_argument('prefix', help='prefix of the jobs', type=str)
parser.add_argument('-k', '--kraken', help = 'merged kraken input file', type=str)
parser.add_argument('-k_l', '--kraken_level', help = 'merged kraken input file',
type=str)
parser.add_argument('-cm', '--checkm_file', help = 'checkm output file - in tab format',
type = str)
parser.add_argument('-sk', '--seqkit', help = 'seqkit output file - in tab format', type
= str)
parser.add_argument('samples', help='samples.tsv file', type=str)
parser.add_argument('dates', help='plot date scale y/n', type=str)

args = parser.parse_args()
in_file = args.in_file
prefix = args.prefix
samples = args.samples

if args.kraken_level:
    prefix = str(prefix + "_" + args.kraken_level)
else:
    prefix = prefix
```

```
# ==============================================================================
# Plot global settings
# ==============================================================================
matplotlib.rcParams['lines.linewidth'] = 0.5
matplotlib.rcParams['ytick.left'] = True
matplotlib.rcParams['ytick.minor.size'] = 1
matplotlib.rcParams['ytick.minor.width'] = 0.25
matplotlib.rcParams['axes.linewidth'] = 0.5
colours = ["crimson", "purple", "tab:cyan", "seagreen", "darkorange", "tab:pink",
"darkslateblue", "darkgoldenrod", "teal", "darkolivegreen"]

# ==============================================================================
# Read in input file(s)
# ==============================================================================
set_groups = set()
if "y" in args.dates.lower():
    df_samples = pd.read_csv(samples, sep ='\t', parse_dates = ["date"])
else:
    df_samples = pd.read_csv(samples, sep ='\t')

groups = df_samples["group"].tolist() #get rid of header

for item in groups:
    set_groups.add(item)
dc = {}
for item in list(set_groups):
    dc[item] = -1
for i in groups:
    dc[i] +=1

with open(in_file, 'r') as text_file:
    files = text_file.read().strip().split()

# ==============================================================================
# Plot
# ==============================================================================
counter = 0
for i in range(0, len(files), 30):
    gs = gsp.GridSpec(5,6)
    gsplace = 0
    sub_files = files[i:i+30]
    counter += 1
    mean_df = pd.DataFrame(columns=df_samples["sample"].to_list())
    for file in sub_files:
        with open(file, 'r') as f:
            file = str(file)
            df = pd.read_csv(f, index_col='contig')
            av_cov = str('{0:.1f}'.format(statistics.mean(df['cover'].tolist())))
            sd_cov = str('{0:.1f}'.format(statistics.stdev(df['cover'].tolist())))
            tot_len = str('{0:.1f}'.format(sum(df['length'].tolist())/1000))
            av_gc =  str('{0:.1f}'.format(statistics.mean(df['GC'].tolist())))
            sd_gc = str('{0:.1f}'.format(statistics.stdev(df['GC'].tolist())))
            na = str(file.split('/')[-1:][0].split('.')[0][8:])
            file_na = str(file.split('/')[-1:][0].split('.')[0][8:])
#df['length'].idxmax()
            nu = str(len(df))
            axes1 = plt.subplot(gs[gsplace])

            x_start = 0
            x_prev_start = 0
            x_prev_end = 0

            mean_list =[]

            for item in list(set_groups):
                x_end = x_start + dc[item] + 1
                y_mean = df.mean()[x_start:x_end]
                mean_list.extend(df.mean()[x_start:x_end].to_list())
                #print(mean_list)
                top = df.max()[x_start:x_end]
                bottom = df.min()[x_start:x_end]
                #print(df.mean()[x_start:x_end])
                if "y" in args.dates.lower():
                    df_samples["date"] = pd.to_datetime(df_samples["date"])
                    if x_start == 0:
                        x_data = df_samples["date"][x_start:x_end]
                    else:
                        x_data = df_samples["date"][x_start:x_end] + (x_data[-
1:][x_start-1] - df_samples["date"][0] + datetime.timedelta(days=5))
                else:
                    x_data = range(x_start, x_end)
                plt.plot(x_data, y_mean, color=colours[item])
                plt.fill_between(x_data, top, bottom, facecolor='gray', alpha=0.5)
                x_prev_start = x_start
                x_prev_end = x_end
                x_start = x_end

            plt.tick_params(labelbottom=False)

            plt.semilogy()

            x1,x2,y1,y2 = plt.axis()
            plt.axis((x1,x2,0.0001,100))
            plt.axhline(y=0.01, ls='--', lw = 0.25, c = 'black')

            if args.seqkit:
                seqkit_df = pd.read_csv(args.seqkit, sep = '\t', index_col =0)
                file_fa = file.replace(".csv",".fasta")
                n_50 = seqkit_df["N50"][file_fa]
                if "y" in args.dates.lower():
                    plt.text(df_samples["date"][1], 1.7, "N50: " + str(n_50),
fontsize=2)
                else:
                    plt.text(0.5, 1.7, "N50: " + str(n_50), fontsize=2)

            if args.checkm_file:
                checkm_df = pd.read_csv(args.checkm_file, sep = '\t', index_col = 0)
                clus = file.split('/')[-1:][0][:-4]
                comp = checkm_df["Completeness"][clus]
                conta = checkm_df["Contamination"][clus]
                if "y" in args.dates.lower():
                    plt.text(df_samples["date"][1], 0.7, str(comp)+'%: Complete ' +
str(conta)+'%: Contamination', fontsize=2)
                else:
                    plt.text(0.5, 0.7, str(comp) + '%:  Complete ' + str(conta) + '%:
Contamination', fontsize=2)

            if args.kraken:
                for line in open(args.kraken, 'r'):
                    if re.search(file_na, line):
                        cont = line.split('\t')[-1].strip()
                        if './' in cont:
                            cont = ' '
                        else:
                            per = line.split('\t')[1]
                            per = per.strip()
                        if "y" in args.dates.lower():
                            plt.text(df_samples["date"][1], 0.3, per+'%:  ' + cont,
fontsize=2)
                        else:
                            plt.text(0.5, 0.3, per+'%:  ' + cont, fontsize=2)
            if "y" in args.dates.lower():
                plt.text(df_samples["date"][1], 40, na, fontsize = 2, fontweight='bold')
                plt.text(df_samples["date"][1], 9, nu+' cov:'+av_cov+'+/-'+sd_cov + ', '
+ tot_len +'kb', fontsize=2)
                plt.text(df_samples["date"][1], 4, 'GC% '+ av_gc +'+/-'+ sd_gc,
fontsize=2)
            else:
                plt.text(0.5, 40, na, fontsize = 2, fontweight='bold')
                plt.text(0.5, 9, nu +', cov:'+av_cov+'+/-'+sd_cov + ', ' + tot_len
+'kb', fontsize=2)
                plt.text(0.5, 4, 'GC% '+ av_gc +'+/-'+ sd_gc, fontsize=2)
            plt.tick_params(axis='x', labelsize=2, pad=0, direction='out', length=1,
width=0.25)
            plt.tick_params(axis = 'y', labelsize=2, pad=0, direction='out', length=1)
            plt.tick_params(right=False, top=False)
            gsplace += 1
        mean_df = mean_df.append(pd.Series(mean_list, name = file, index =
df_samples["sample"].to_list()))

    plt.savefig('output/plots/' + str(counter) + '_' + prefix + '_plot.png', type='png',
dpi=600)
    print('Generated plot number ' + str(counter) + ' -> ' + str(counter) + '_' + prefix
+ '_plot.png')
    plt.close('all')

mean_df.to_csv(prefix + "_clus_means.csv")
```

abun_plot.py

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

import matplotlib.pyplot as plt
import matplotlib as mpl
from matplotlib import rc
import argparse
import pandas as pd
import math
from scipy.cluster import hierarchy as hc
import re
import matplotlib.cm as cm
import numpy as np

# =============================================================================
# Command line parsing
# =============================================================================

parser = argparse.ArgumentParser(description='usage = python prefix csv_in binned_in
plot(r/a) top20(y/n) -s sample_list -k kraken_file ')
parser.add_argument('prefix', help='prefix of the jobs', type=str)
parser.add_argument('csv_in', help='file containing absolute abundance counts for every
contig', type=str)
parser.add_argument('binned_in', help='file listing contigs in each cluster', type=str)
parser.add_argument('plot', help='relative or absolute abundance output (r/a)',
type=str)
parser.add_argument('top20', help = 'output top19 and other or output all y/n', type =
str)
parser.add_argument('-s', '--sample', dest='sample', nargs='+', default=[])
parser.add_argument('-k', '--kraken', dest='kraken', help = 'kraken top output file', 
type = str)

args = parser.parse_args()

prefix = args.prefix
csv_in = args.csv_in
plot = args.plot
binned_in = args.binned_in
sample = args.sample
top20 = args.top20

if args.kraken:
    kraken = args.kraken
    taxo = "y"
else:
    taxo = "n"

#Dendogram options
LinkMethod = "weighted"
metric = 'correlation'

# ==============================================================================
#
# ==============================================================================
df_abun = pd.read_csv(csv_in, index_col = 0, names = sample)
df_abun = df_abun.drop(columns='Coverage')
tot = df_abun.sum(axis = 0)
cluster_abun = pd.DataFrame(columns=df_abun.columns)
new_df_abun = pd.DataFrame(columns = sample).drop(columns='Coverage')
abun = pd.DataFrame(columns = sample).drop(columns='Coverage')

with open(binned_in, 'r') as binned_list:
    for line in binned_list:
        if taxo == "y":
            if line.startswith(">"): #get cluster info
                cluster = line.strip('>').strip()[:-6] #strip things
                with open(kraken, 'r') as kraken_f:
                    for line2 in kraken_f:
                        if re.search(cluster, line2):
                            name = line2.split('\t')[-1].strip()
                            if name =="":
                                name = 'Unclassified'
            else:
                line = line.strip().split(' ')[0] #split may not work with NAB_997 - 
check
                if line in df_abun.index:
                    if name in new_df_abun.index:
                        new_df_abun.loc[name] = 
new_df_abun.loc[name].add(df_abun.loc[line])
                    else:
                        new_df_abun.loc[name] = df_abun.loc[line]
        else:
            if line.startswith(">"): #get cluster info
                cluster = line.strip('>').strip()[:-6] #strip things
                name = cluster
            else:
                line = line.strip().split(' ')[0]
                if line in df_abun.index:
                    if name in new_df_abun.index:
                        new_df_abun.loc[name] = 
new_df_abun.loc[name].add(df_abun.loc[line])
                    else:
                        new_df_abun.loc[name] = df_abun.loc[line]
                else:
                    print("something wrong here")
```

```
if 'y' in top20:
    new_df_abun["sum"]=new_df_abun.sum(axis=1)
    top_df_abun = new_df_abun.sort_values('sum', axis=0,
ascending=False).head(19).drop(columns = "sum")
    other_df_abun = new_df_abun.sort_values('sum', axis=0, ascending=False).iloc[19:,]
    top_df_abun.loc["Other"] = other_df_abun.sum(axis=0).drop(columns="sum")
    new_df_abun = top_df_abun

for column in new_df_abun: #iterate over columns
    per = []
    for val in new_df_abun.loc[:, column]:
        if 'a' in plot:
            per.append(val)
        if 'r' in plot:
            per.append(((val/new_df_abun[column].sum())*100))
    abun[column] = pd.Series(per, name = column) #sort name out

abun_sum = abun.cumsum()

abun.index = new_df_abun.index.values.tolist()
if 'r' in plot:
    abun.to_csv(prefix + "_relative_counts.csv")

prev = ""
previous = pd.Series()

fig = plt.figure(1)

for i in range(0, len(abun.index.values.tolist())): #for each cluster i.e list of abun
index
    if not prev:
        plt.bar(abun.keys(), abun.iloc[i, :], label=abun.index.values.tolist()[i],
width=0.9)
    else:
        plt.bar(abun.keys(), abun.iloc[i, :], bottom=previous,
label=abun.index.values.tolist()[i],  width=0.9)
        #print(abun.keys())
    prev = 'y'
    if i != 0:
        previous = abun.iloc[i, :] + abun_sum.iloc[i-1, :]
    else:
        previous = abun.iloc[i, :]

plt.xticks(abun.keys(), abun.keys(), rotation='vertical', fontsize = 4,
verticalalignment='center_baseline')
plt.legend(loc='upper left', bbox_to_anchor=(1,1), ncol=1, frameon=False, fontsize = 5)

if 'a' in plot:
    plt.margins(x = 0.01, y=0.05)
if 'r' in plot:
    plt.margins(0)
if 'y' in top20:
    plot = plot + '_top20'

fig.savefig(str("output/plots/" + prefix + '_' + plot + '_abun_plot.png'),
bbox_inches='tight', dpi = 400)
plt.show()

if 'y' in top20:
    colours = ['#502db3', '#008080', '#c200f2', '#f2c200','#36a3d9',
'#e6beff','#8c0025', '#f58231', '#bf0080', '#cad900','#911eb4','#e5001f','#0066bf',
'#000075','#338000', '#f032e6','#1bca00','#1d4010','#9a6324','#a9a9a9']

    abun_flip = abun.transpose()
    abun_flip.plot.bar(stacked=True, legend = None, figsize=(15,10), color=colours,
width=0.9)
    plt.legend(loc='center left', labelspacing=-2.5, bbox_to_anchor=(1.0, 0.5),
frameon=False)
    plt.ylim(0,100)
    plt.tight_layout()
    plt.savefig('output/plots/' + prefix +'_' + plot +'_'+ 'abun.png',
bbox_inches='tight')

GenusData = abun

z = hc.linkage(GenusData.values.T, method=LinkMethod, metric=metric)

plt.figure(num=None, figsize=(20, 10),  facecolor='w', edgecolor='k')
dendrogram = hc.dendrogram(z, labels=GenusData.columns,  color_threshold=0.04,
leaf_font_size=10, leaf_rotation=90)
for key in dendrogram.keys():
    if key == 'ivl':
        DenOrder = dendrogram[key]
plt.savefig("output/plots/" + prefix +'_' + plot +'_'+ LinkMethod + metric
+'_dendro.png', bbox_inches='tight', dpi = 400)

GenusData = GenusData[DenOrder]
GenusData = GenusData.transpose()
GenusData.plot.bar(stacked=True, legend = None, figsize=(30,20), width=0.9)
plt.legend(loc='center left', labelspacing=-2.5,  bbox_to_anchor=(1.0, 0.5))
plt.savefig("output/plots/" + prefix +'_' + plot +'_'+ 'ord_abun.png',
bbox_inches='tight')
```

clus_stats.py

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

import statistics
import pandas as pd
import argparse

# ==============================================================================
# Command line parsing
# ==============================================================================
parser = argparse.ArgumentParser(description='usage = python in_file prefix -k
kraken_file -k_l kraken_level -cm checkm -sk seqkit_file samples_file date(y/n)')
parser.add_argument('in_file', help='the name of the file containing a list of csv
files', type=str)
parser.add_argument('prefix', help='prefix of the jobs', type=str)
parser.add_argument('-cm', '--checkm_file', help = 'checkm output file - in tab format',
type = str)
parser.add_argument('-sk', '--seqkit', help = 'seqkit output file - in tab format', type
= str)

args = parser.parse_args()
in_file = args.in_file
prefix = args.prefix

# ==============================================================================
# Read in input file(s)
# ==============================================================================
set_groups = set()
with open(in_file, 'r') as text_file:
    files = text_file.read().strip().split()

# ==============================================================================
# Stats
# ================================================================================
stats_df = pd.DataFrame(columns=['no_seq','tot_len','av_cov','sd_cov',
'av_gc','sd_gc','n_50','comp', 'contam'])

for file in files:
    with open(file, 'r') as f:
        file = str(file)
        df = pd.read_csv(f, index_col='contig')
        av_cov = str('{0:.1f}'.format(statistics.mean(df['cover'].tolist())))
        sd_cov = str('{0:.1f}'.format(statistics.stdev(df['cover'].tolist())))
        tot_len = str(sum(df['length'].tolist()))
        av_gc =  str('{0:.1f}'.format(statistics.mean(df['GC'].tolist())))
        sd_gc = str('{0:.1f}'.format(statistics.stdev(df['GC'].tolist())))
        na = str(file.split('/')[-1:][0].split('.')[0])
        nu = str(len(df))
        if args.seqkit:
            seqkit_df = pd.read_csv(args.seqkit, sep = '\t', index_col =0)
            file_fa = file.replace(".csv",".fasta")
            n_50 = seqkit_df["N50"][file_fa]
        if args.checkm_file:
            checkm_df = pd.read_csv(args.checkm_file, sep = '\t', index_col = 0)
            clus = file.split('/')[-1:][0][:-4]
            comp = checkm_df["Completeness"][clus]
            conta = checkm_df["Contamination"][clus]
        stats_df.loc[na] = [nu,tot_len,av_cov,sd_cov,av_gc,sd_gc,n_50,comp,conta]

stats_df.to_csv("output/" + prefix + "_cluster_summary_stats.tsv", sep='\t')
```

bin_plot.py

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

import numpy as np
import matplotlib.pyplot as plt
from matplotlib import rc
import argparse
import pandas as pd
import math

def bins(file, bins):
    data = open(file, 'r')
    counts = []
    for i in range(len(bins)):
        counts.append(0)
    x = data.readline()
    x = data.readline()
    while x:
       a = x.split(sep='\t')
       pos = 0
       for loop in bins:
          if int(a[1]) < loop*100000:
             counts[pos] += 1
             break
          else:
             pos += 1
       x = data.readline()
    data.close()
    gt = 0
    for i in counts:
       gt += i
    return counts

# ================================================================================
# Command line parsing
# ==============================================================================
parser = argparse.ArgumentParser(description='usage = python entrez_down.py
file_list_of_queries')
parser.add_argument('unbinned_file', help='output from checkm unbinned', type=str)
parser.add_argument('binned_file', help='output from bash script', type=str)
parser.add_argument('prefix', help='prefix of the jobs', type=str)

args = parser.parse_args()

unbinned_file = args.unbinned_file
binned_file = args.binned_file
prefix = args.prefix

unbin_df = pd.read_csv(unbinned_file, sep = '\t')
bin_df = pd.read_csv(binned_file, sep = '\t', names = ["Contig", "Length"])
max_len = max(math.ceil(unbin_df["Length"].max()/100000),
math.ceil(bin_df["Length"].max()/100000))

groups = [0.02,0.05,0.1,0.2,0.5]
for i in range(1, max_len+1):
    groups.append(i)

unbinned = bins(unbinned_file, groups)
binned = bins(binned_file, groups)

x = []

for i in range(len(groups)):
    x.append(i)

bin_bars = []
unbin_bars = []

for i in range(len(unbinned)):
    total = unbinned[i] + binned[i]
    if total > 0:
        if binned[i] > 0 :
            bin_bars.append((binned[i]/total)*100)
        else:
            bin_bars.append(0)
        if unbinned[i] >0:
            unbin_bars.append((unbinned[i]/total)*100)
        else:
            unbin_bars.append(0)
    else:
        bin_bars.append(0)
        unbin_bars.append(0)

fig = plt.figure(1)
plt.bar(x, bin_bars, color='#25335d', edgecolor='none', label='binned', width=1)
plt.bar(x, unbin_bars, bottom=bin_bars, color='#abb9e3', edgecolor='none',
label='unbinned', width=1)
plt.legend(loc='upper left', bbox_to_anchor=(1,1), ncol=1, frameon=False)
plt.xlabel('Size (100Kb)', size=8)
plt.ylabel('% Contigs', size=8)
plt.xticks(x, groups)
plt.tick_params(labelsize = 7)
fig.savefig(str('output/plots/' + prefix + '_bin_contigs.png'), bbox_inches='tight', dpi
= 400)
counts = pd.DataFrame(index=groups)
counts["binned"] = binned
counts["unbinned"] = unbinned
counts.to_csv(prefix + "_bin_group_stats.csv")
```

absolute_derive.py

```
#!/usr/bin/env python3

# function that adds all the count values together to get a total
def summer(x):
    total = 0
    for loop in x:
        total = total + int(loop)
    return(total)
# function that divides each point by total counts for this contig
def deriver(x,y):
    answers=[]
    for loop in x:
        if y == 0:
            y = 1
        value = int(loop)
        answers.append(value)
    return(answers)

import csv as csv
import os
import argparse

parser = argparse.ArgumentParser(description='')
parser.add_argument('loc', help='location count files are in', type=str)
parser.add_argument('jobid', help='location count files are in', type=str)
parser.add_argument('thresh', help='location count files are in', type=int)
args = parser.parse_args()
loc = str("output/" + args.loc)
jobid = args.jobid
thresh = args.thresh

dir_name = str(loc + '/')
file_name = str(jobid + '_read_counts.out')
new_record=[[]]
nr=False

with open(dir_name+file_name, 'r') as data_store:
    line = csv.reader(data_store, delimiter='\t')
    for i in line:
        if int(i[1]) >= thresh:
            counts = summer(i[2:])
            values = deriver((i[2:]), counts)
            coverage = (counts*150)/int(i[1])
            values.append(coverage) # add coverage to the end of the entry
            values.insert(0,i[0])   # add contig name to the front of the entry
            if nr:
                new_record.append(values)
            else:                   # identifies first entry in the list
                new_record = [values]
                nr = True
with open(dir_name + '/' +  jobid + '_read_counts_absolute.csv', 'w') as f:
    writer = csv.writer(f)
    writer.writerows(new_record)
```

References

Alanjary, M., Steinke, K. and Ziemert, N. (2019) ‘AutoMLST: an automated web server for generating multi-locus species trees highlighting natural product potential’, Nucleic acids research, 47(W1), pp. W276–W282.

Alkan, C., Sajjadian, S. and Eichler, E. E. (2011) ‘Limitations of next-generation genome sequence assembly’, Nature methods, 8(1), pp. 61–65.

Almeida, A. *et al*. (2019) ‘A new genomic blueprint of the human gut microbiota’, Nature, 568(7753), pp. 499–504.

Alneberg, J. *et al*. (2014) ‘Binning metagenomic contigs by coverage and composition’, Nature methods, 11(11), pp. 1144–1146.

Altschul, S. F. *et al*. (1990) ‘Basic local alignment search tool’, Journal of molecular biology, 215(3), pp. 403–410.

Amgarten, D. *et al*. (2018) ‘MARVEL, a Tool for Prediction of Bacteriophage Sequences in Metagenomic Bins’, Frontiers in genetics, 9, p. 304.

Andrews S. (2010). FastQC: a quality control tool for high throughput sequence data. Available online at: http://www.bioinformatics.babraham.ac.uk/projects/fastqc

Ashton, P. M. *et al*. (2015) ‘MinION nanopore sequencing identifies the position and structure of a bacterial antibiotic resistance island’, Nature biotechnology, 33(3), pp. 296–300.

Ayling, M., Clark, M. D. and Leggett, R. M. (2019) ‘New approaches for metagenome assembly with short reads’, Briefings in bioinformatics. doi: 10.1093/bib/bbz020.

Bankevich, A. *et al*. (2012) ‘SPAdes: a new genome assembly algorithm and its applications to single-cell sequencing’, Journal of computational biology: a journal of computational molecular cell biology, 19(5), pp. 455–477.

Bernard, G. *et al*. (2018) ‘Microbial Dark Matter Investigations: How Microbial Studies Transform Biological Knowledge and Empirically Sketch a Logic of Scientific Discovery’, Genome biology and evolution, 10(3), pp. 707–715.

Bohlin, J. *et al*. (2010) ‘Analysis of intra-genomic GC content homogeneity within prokaryotes’, BMC genomics, 11, p. 464.

Bowers, R. M. *et al*. (2017) ‘Minimum information about a single amplified genome (MISAG) and a metagenome-assembled genome (MIMAG) of bacteria and archaea’, Nature biotechnology, 35(8), pp. 725–731.

Campanaro, S. *et al*. (2019) ‘The anaerobic digestion microbiome: a collection of 1600 metagenome-assembled genomes shows high species diversity related to methane production’, bioRxiv. doi: 10.1101/680553.

Campillo-Balderas, J. A., Lazcano, A. and Becerra, A. (2015) ‘Viral Genome Size Distribution Does not Correlate with the Antiquity of the Host Lineages’, Frontiers in Ecology and Evolution, 3, p. 143.

Castro, C. J. and Ng, T. F. F. (2017) ‘U50: A New Metric for Measuring Assembly Output Based on Non-Overlapping, Target-Specific Contigs’, Journal of computational biology: a journal of computational molecular cell biology, 24(11), pp. 1071–1080.

Chan, J. Z.-M. *et al*. (2012) ‘Defining bacterial species in the genomic era: insights from the genus Acinetobacter’, BMC microbiology, 12, p. 302.

Cornwell, M. *et al*. (2018) ‘VIPER: Visualization Pipeline for RNA-seq, a Snakemake workflow for efficient and complete RNA-seq analysis’, BMC bioinformatics, 19(1), p. 135.

Danczak, R. E. *et al*. (2017) ‘Members of the Candidate Phyla Radiation are functionally differentiated by carbon- and nitrogen-cycling capabilities’, Microbiome, 5(1), p. 112.

De Maio, N. *et al*. (2019) ‘Comparison of long-read sequencing technologies in the hybrid assembly of complex bacterial genomes’, Microbial genomics, 5(9). doi: 10.1099/mgen.0.000294.

Denton, J. F. *et al*. (2014) ‘Extensive error in the number of genes inferred from draft genome assemblies’, PLoS computational biology, 10(12), p. e1003998.

Di Tommaso, P. *et al*. (2017) ‘Nextflow enables reproducible computational workflows’, Nature biotechnology, 35(4), pp. 316–319.

Frisli, T. *et al*. (2013) ‘Estimation of metagenome size and structure in an experimental soil microbiota from low coverage next-generation sequence data’, Journal of applied microbiology, 114(1), pp. 141–151.

Fu, S., Wang, A. and Au, K. F. (2019) ‘A comparative evaluation of hybrid error correction methods for error-prone long reads’, Genome biology, 20(1), p. 26.

Giordano, F. *et al*. (2017) ‘De novo yeast genome assemblies from MinION, PacBio and MiSeq platforms’, Scientific reports, 7(1), p. 3935.

Goldstein, S. *et al*. (2019) ‘Evaluation of strategies for the assembly of diverse bacterial genomes using MinION long-read sequencing’, BMC genomics, 20(1), p. 23.

Grüning, B. *et al*. (2018) ‘Practical Computational Reproducibility in the Life Sciences’, Cell systems, 6(6), pp. 631–635.

Hatfull, G. F. and Hendrix, R. W. (2011) ‘Bacteriophages and their genomes’, Current opinion in virology, 1(4), pp. 298–303.

Hillmann, B. *et al*. (2018) ‘Evaluating the Information Content of Shallow Shotgun Metagenomics’, mSystems, 3(6). doi: 10.1128/mSystems.00069-18.

Huang, W. *et al*. (2012) ‘ART: a next-generation sequencing read simulator’, Bioinformatics, 28(4), pp. 593–594.

Imachi, H. *et al*. (2019) ‘Isolation of an archaeon at the prokaryote-eukaryote interface’, bioRxiv. doi: 10.1101/726976.

Jain, M. *et al*. (2018) ‘Nanopore sequencing and assembly of a human genome with ultra-long reads’, Nature biotechnology, 36(4), pp. 338–345.

Janda, J. M. and Abbott, S. L. (2007) ‘16S rRNA gene sequencing for bacterial identification in the diagnostic laboratory: pluses, perils, and pitfalls’, Journal of clinical microbiology, 45(9), pp. 2761–2764.

Kang, D. D. *et al*. (2015) ‘MetaBAT, an efficient tool for accurately reconstructing single genomes from complex microbial communities’, PeerJ, 3, p. e1165.

Kim, D. *et al*. (2016) ‘Centrifuge: rapid and sensitive classification of metagenomic sequences’, Genome research, 26(12), pp. 1721–1729.

Kirkegaard, R. H. *et al*. (2017) ‘The impact of immigration on microbial community composition in full-scale anaerobic digesters’, Scientific reports, 7(1), p. 9343.

Kolmogorov, M. *et al*. (2019) ‘Assembly of long, error-prone reads using repeat graphs’, Nature biotechnology, 37(5), pp. 540–546.

Koren, S. *et al*. (2017) ‘Canu: scalable and accurate long-read assembly via adaptive k-mer weighting and repeat separation’, Genome research, 27(5), pp. 722–736.

Koren, S. and Phillippy, A. M. (2015) ‘One chromosome, one contig: complete microbial genomes from long-read sequencing and assembly’, Current opinion in microbiology, 23, pp. 110–120.

Köster, J. and Rahmann, S. (2012) ‘Snakemake--a scalable bioinformatics workflow engine’, Bioinformatics, 28(19), pp. 2520–2522.

Latorre-Pérez, A. *et al*. (2019) ‘Assembly methods for nanopore-based metagenomic sequencing: a comparative study’, bioRxiv. doi: 10.1101/722405.

Leipzig, J. (2017) ‘A review of bioinformatic pipeline frameworks’, Briefings in bioinformatics, 18(3), pp. 530–536.

Letunic, I. and Bork, P. (2019) ‘Interactive Tree Of Life (iTOL) v4: recent updates and new developments’, Nucleic acids research, 47(W1), pp. W256–W259.

Li, D. *et al*. (2015) ‘MEGAHIT: an ultra-fast single-node solution for large and complex metagenomics assembly via succinct de Bruijn graph’, Bioinformatics, 31(10), pp. 1674–1676.

Li, H. *et al*. (2009) ‘The Sequence Alignment/Map format and SAMtools’, Bioinformatics, 25(16), pp. 2078–2079.

Li, H. (2013) ‘Aligning sequence reads, clone sequences and assembly contigs with BWA-MEM’, arXiv [q-bio.GN]. Available at: http://arxiv.org/abs/1303.3997.

Li, H. (2017) ‘Minimap2: versatile pairwise alignment for nucleotide sequences’, arXiv [q-bio.GN]. Available at: http://arxiv.org/abs/1708.01492.

Li, H. and Durbin, R. (2009) ‘Fast and accurate short read alignment with Burrows-Wheeler transform’, Bioinformatics, 25(14), pp. 1754–1760.

Lin, Y. *et al*. (2016) ‘Assembly of long error-prone reads using de Bruijn graphs’, Proceedings of the National Academy of Sciences of the United States of America, 113(52), pp. E8396–E8405.

Li, Y. *et al*. (2018) ‘DeepSimulator: a deep simulator for Nanopore sequencing’, Bioinformatics, 34(17), pp. 2899–2908.

Martin, M. (2011) ‘Cutadapt removes adapter sequences from high-throughput sequencing reads’, EMBnet.journal, 17(1), pp. 10–12.

McIlroy, S. J. *et al*. (2017) ‘MiDAS 2.0: an ecosystem-specific taxonomy and online database for the organisms of wastewater treatment systems expanded for anaerobic digester groups’, Database: the journal of biological databases and curation, 2017(1). doi: 10.1093/database/bax016.

McNair, K. *et al*. (2019) ‘PHANOTATE: a novel approach to gene identification in phage genomes’, Bioinformatics, 35(22), pp. 4537–4542.

Meric, G. *et al*. (2019) ‘Correcting index databases improves metagenomic studies’, bioRxiv. doi: 10.1101/712166.

Meyer, F. *et al*. (2018) ‘AMBER: Assessment of Metagenome BinnERs’, GigaScience, 7(6). doi: 10.1093/gigascience/giy069.

Mikheenko, A., Saveliev, V. and Gurevich, A. (2016) ‘MetaQUAST: evaluation of metagenome assemblies’, Bioinformatics, 32(7), pp. 1088–1090.

Morgan, J. L., Darling, A. E. and Eisen, J. A. (2010) ‘Metagenomic sequencing of an in vitro-simulated microbial community’, PloS one, 5(4), p. e10209.

Murat Eren, A. *et al*. (2015) ‘Anvi’o: an advanced analysis and visualization platform for ‘omics data’, PeerJ. PeerJ Inc., 3, p. e1319.

Nasko, D. J. *et al*. (2018) ‘RefSeq database growth influences the accuracy of k-mer-based lowest common ancestor species identification’, Genome biology, 19(1), p. 165.

Nicholls, S. M. *et al*. (2019) ‘Ultra-deep, long-read nanopore sequencing of mock microbial community standards’, GigaScience, 8(5). doi: 10.1093/gigascience/giz043.

O’Leary, N. A. *et al*. (2016) ‘Reference sequence (RefSeq) database at NCBI: current status, taxonomic expansion, and functional annotation’, Nucleic acids research, 44(D1), pp. D733–45.

Olson, N. D. *et al*. (2017) ‘Metagenomic assembly through the lens of validation: recent advances in assessing and improving the quality of genomes assembled from metagenomes’, Briefings in bioinformatics. doi: 10.1093/bib/bbx098.

Pace, N. R. (2009) ‘Mapping the tree of life: progress and prospects’, Microbiology and molecular biology reviews: MMBR, 73(4), pp. 565–576.

Parks, D. H. *et al*. (2015) ‘CheckM: assessing the quality of microbial genomes recovered from isolates, single cells, and metagenomes’, Genome research, 25(7), pp. 1043–1055.

Parks, D. H. *et al*. (2017) ‘Recovery of nearly 8,000 metagenome-assembled genomes substantially expands the tree of life’, Nature microbiology, 2(11), pp. 1533–1542.

Parks, D. H. *et al*. (2018) ‘A standardized bacterial taxonomy based on genome phylogeny substantially revises the tree of life’, Nature biotechnology, 36(10), pp. 996–1004.

Pasolli, E. *et al*. (2019) ‘Extensive Unexplored Human Microbiome Diversity Revealed by Over 150,000 Genomes from Metagenomes Spanning Age, Geography, and Lifestyle’, Cell, 176(3), pp. 649–662.e20.

Paszkiewicz, K. and Studholme, D. J. (2010) ‘De novo assembly of short sequence reads’, Briefings in bioinformatics, 11(5), pp. 457–472.

Peces, M. *et al*. (2018) ‘Deterministic mechanisms define the long-term anaerobic digestion microbiome and its functionality regardless of the initial microbial community’, Water research, 141, pp. 366–376.

Quast, C. *et al*. (2013) ‘The SILVA ribosomal RNA gene database project: improved data processing and web-based tools’, Nucleic acids research, 41(Database issue), pp. D590–6.

Rang, F. J., Kloosterman, W. P. and de Ridder, J. (2018) ‘From squiggle to basepair: computational approaches for improving nanopore sequencing read accuracy’, Genome biology, 19(1), p. 90.

Rappé, M. S. and Giovannoni, S. J. (2003) ‘The uncultured microbial majority’, Annual review of microbiology, 57, pp. 369–394.

Reichenberger, E. R. *et al*. (2015) ‘Prokaryotic nucleotide composition is shaped by both phylogeny and the environment’, Genome biology and evolution, 7(5), pp. 1380–1389.

Schmid, M. *et al*. (2018) ‘Pushing the limits of de novo genome assembly for complex prokaryotic genomes harboring very long, near identical repeats’, Nucleic acids research, 46(17), pp. 8953–8965.

Sczyrba, A. *et al*. (2017) ‘Critical Assessment of Metagenome Interpretation-a benchmark of metagenomics software’, Nature methods, 14(11), pp. 1063–1071.

Seemann, T. (2014) ‘Prokka: rapid prokaryotic genome annotation’, Bioinformatics , 30(14), pp. 2068–2069.

Shah, N. *et al*. (2011) ‘Comparing bacterial communities inferred from 16S rRNA gene sequencing and shotgun metagenomics’, Pacific Symposium on Biocomputing. Pacific Symposium on Biocomputing, pp. 165–176.

Sharon, I. *et al*. (2013) ‘Time series community genomics analysis reveals rapid shifts in bacterial species, strains, and phage during infant gut colonization’, Genome research, 23(1), pp. 111–120.

Shen, W. *et al*. (2016) ‘SeqKit: A Cross-Platform and Ultrafast Toolkit for FASTA/Q File Manipulation’, PloS one, 11(10), p. e0163962.

Sieber, C. M. K. *et al*. (2018) ‘Recovery of genomes from metagenomes via a dereplication, aggregation and scoring strategy’, Nature microbiology, 3(7), pp. 836–843.

Simpson, J. T. *et al*. (2017) ‘Detecting DNA cytosine methylation using nanopore sequencing’, Nature methods, 14(4), pp. 407–410.

Sims, D. *et al*. (2014) ‘Sequencing depth and coverage: key considerations in genomic analyses’, Nature reviews. Genetics, 15(2), pp. 121–132.

Solli, L. *et al*. (2014) ‘A metagenomic study of the microbial communities in four parallel biogas reactors’, Biotechnology for biofuels, 7(1), p. 146.

Somerville, V. *et al*. (2019) ‘Long-read based de novo assembly of low-complexity metagenome samples results in finished genomes and reveals insights into strain diversity and an active phage system’, BMC microbiology, 19(1), p. 143.

Sović, I. *et al*. (2016) ‘Evaluation of hybrid and non-hybrid methods for de novo assembly of nanopore reads’, Bioinformatics, 32(17), pp. 2582–2589.

Spang, A. *et al*. (2015) ‘Complex archaea that bridge the gap between prokaryotes and eukaryotes’, Nature, 521(7551), pp. 173–179.

Stewart, E. J. (2012) ‘Growing unculturable bacteria’, Journal of bacteriology, 194(16), pp. 4151–4160.

Stewart, R. D. *et al*. (2018) ‘Assembly of 913 microbial genomes from metagenomic sequencing of the cow rumen’, Nature communications, 9(1), p. 870.

Tamames, J. and Puente-Sánchez, F. (2018) ‘SqueezeMeta, A Highly Portable, Fully Automatic Metagenomic Analysis Pipeline’, Frontiers in microbiology, 9, p. 3349.

Treangen, T. J. and Salzberg, S. L. (2011) ‘Repetitive DNA and next-generation sequencing: computational challenges and solutions’, Nature reviews. Genetics, 13(1), pp. 36–46.

Treu, L. *et al*. (2016) ‘Deeper insight into the structure of the anaerobic digestion microbial community; the biogas microbiome database is expanded with 157 new genomes’, Bioresource technology, 216, pp. 260–266.

Ushiki, N. *et al*. (2017) ‘Nitrite oxidation kinetics of two Nitrospira strains: The quest for competition and ecological niche differentiation’, Journal of bioscience and bioengineering, 123(5), pp. 581–589.

Vaser, R. *et al*. (2017) ‘Fast and accurate de novo genome assembly from long uncorrected reads’, Genome research, 27(5), pp. 737–746.

Visconti, A., Martin, T. C. and Falchi, M. (2018) ‘YAMP: a containerized workflow enabling reproducibility in metagenomics research’, GigaScience, 7(7). doi: 10.1093/gigascience/giy072.

Vollmers, J., Wiegand, S. and Kaster, A.-K. (2017) ‘Comparing and Evaluating Metagenome Assembly Tools from a Microbiologist’s Perspective - Not Only Size Matters!’, PloS one, 12(1), p. e0169662.

Walker, B. J. *et al*. (2014) ‘Pilon: an integrated tool for comprehensive microbial variant detection and genome assembly improvement’, PloS one, 9(11), p. e112963.

Watson, M. and Warr, A. (2019) ‘Errors in long-read assemblies can critically affect protein prediction’, Nature biotechnology, pp. 124–126.

Weirather, J. L. *et al*. (2017) ‘Comprehensive comparison of Pacific Biosciences and Oxford Nanopore Technologies and their applications to transcriptome analysis’, F1000Research, 6. doi: 10.12688/f1000research.10571.1.

Wood, D. E., Lu, J. and Langmead, B. (2019) ‘Improved metagenomic analysis with Kraken 2’, Genome biology, 20(1), p. 257.

Wood, D. E. and Salzberg, S. L. (2014) ‘Kraken: ultrafast metagenomic sequence classification using exact alignments’, Genome biology, 15(3), p. R46.

Wu, Y.-W. *et al*. (2014) ‘MaxBin: an automated binning method to recover individual genomes from metagenomes using an expectation-maximization algorithm’, Microbiome, 2, p. 26.

Zaheer, R. *et al*. (2018) ‘Impact of sequencing depth on the characterization of the microbiome and resistome’, Scientific reports, 8(1), p. 5890.