Evaluating Metagenomic Assemblies Using Gene Metrics Derived from Protein Coding Sequences

Felix Andrade May[1]

[1]**Department of Computer Science, Aberystwyth University**

Corresponding author:
Felix Andrade May[1]

Email address: f.andrademay@gmail.com

ABSTRACT

The High-C binning metagenomic assembly technique sometimes produces collapsed repeats and the loss of complex regions of DNA that do not assemble well. This creates less gene dense Metagenome-Assembled Genomes (MAGs), which are harder to accurately classify (Stewart et al., 2018). A classification accuracy of 100% makes the difficult task of identifying genetic samples from within the rumen easier and circumvents the need to culture bacteria before sequencing. By developing new methods to analyse the ability of metagenomic assembly techniques it becomes easier to remove poor MAGs from classification training data and improve the classification accuracy. This paper develops and tests a pipeline for calculating gene metrics, primarily gene density, on consumer grade hardware that could be deployed in crowdsourced citizen science projects. Using four MAGs produced by Stewart et al. (2018), the pipeline can be used to evaluate the High-C binning method of metagenomic assembly used to assemble the four MAGs. The proposed method was able to calculate gene counts and densities in line with expected values for bacterial genomes. When tested on the RefSeq Saccharomyces Cerevisiae genome Engel et al. (2022) it was able to accurately calculate the gene density to within $\pm 40.5\ gMBP^{-1}$ (genes per Mega Base Pairs). This pipeline confirms that the High-C binning assembly technique is capable of producing accurate MAGs. The ability of the pipeline can be easily improved by selecting a harsher threshold for the e-value of alignments produced by BLAST+ search alignments.

INTRODUCTION

Biological environments such as the rumen of cattle contain large numbers of organisms that are difficult to culture in a laboratory environment. As a result, samples must be collected directly from the rumen and then assembled using metagenomic techniques like High-C binning. If these samples are able to be easily and accurately metagenomically assembled then it would improve our ability to sequence, analyse, and classify more genomes and employ the knowledge gained from those processes in developing novel medical techniques.

It is hard to assess the quality of metagenomic assembly because the generated Metagenome-Assembled Genomes (MAGs) cannot be easily aligned against a reference genome using a BLAST+ search algorithm (Camacho et al., 2009) as the MAG may contain the genome for a prokaryote, the host cattle, or a combination of the two. A solution would be to identify Open Reading Frames (ORFs) and compare those to ORFs that are known to produce proteins. These verified ORFs can then be used to approximate a gene density for the genome. This density can be used to evaluate the MAG. This paper lays out a pipeline to solve this problem. The pipeline works to: identify possible ORFs; align the ORFs using BLAST+ against a protein database (The UniProt Consortium, 2020); filter the results of the alignment to exclude any alignments with an e-value beneath a given threshold; calculate an approximation for the number of genes in, and therefore the gene density of, the genome. This pipeline provides a novel technique that, with further development, could be used on a wider scale in crowdsourced citizen science projects as it is capable of running smoothly on consumer grade hardware running Unix style operating systems.

This paper will use the gene density and related metrics calculated from four of the genomes (RUG005, RUG154, RUG431, RUG466) from the Stewart et al. (2018) to analyse the accuracy of the High-C binning method used to produce 913 Metagenome-Assembled Genomes. I will compare the four MAGs to the S. Cerevisiae genome released by Engel et al. (2022), which will be processed using the same pipeline described in Section . It will then be possible to compare the result of this pipeline with known metrics of other prokaryotes and validate the ability of both the proposed pipeline and the High-C binning technique used by Stewart et al. (2018).

MATERIALS AND METHODS

To calculate the gene density of a FastA file with nucleotide sequences from an unknown genome, the FastA must first be processed to identify potential ORFs. These ORFs are then aligned with a BLAST+ search against the Uni-Prot Swiss-Prot protein database (The UniProt Consortium, 2020). The Pairwise format output from the BLAST+ search is processed with several proprietary Python scripts that refine and filter the results. A Python file is then used to calculate the approximate number of genes for each given genome, allowing for the calculation of gene density.

The Prodigal (Hyatt et al., 2010) v2.6.3 Command Line program was used to identify unique ORFs from each genome, creating a corresponding `.gff3` file. Used in combination with Bedtools (Quinlan and Hall, 2010) v2.31.0 these ORFs are extracted from the original genome FastA file into a new FastA file containing only the identified ORFs.

A BLAST+ search was performed on the UniProt Swiss-Prot Database (The UniProt Consortium, 2020) to compare these identified ORFs to known protein Coding Sequences (CDSs). The program found in Appendix C, Listing 6, is used to extract just the query names and the lists of significant alignments and write them to a new file. The code found in Appendix B filters out any matches that have e-values greater than a given value ($1e$-5). The filtering of e-values is performed after the BLAST+ search rather than using a threshold during the BLAST+ search. This is because during a BLAST+ search filtering does not occur during the final stage of the search but rather during the earlier scanning stage of the search algorithm (Camacho et al., 2009).

A method described by Santos-Magalhaes and de Oliveira (2015) provides a way to approximate the number of genes in a given genome based on the average protein length expected for a genome. Equation 2 represents a modification to an equation proposed by Santos-Magalhaes and de Oliveira (2015), substituting the average length of amino acid residues in bacteria, $\bar{L}$. Santos-Magalhaes and de Oliveira (2015) proposed 300 as an average value for bacterial genomes. This pipeline instead calculates an average length of amino acid CDS directly from the genome. This provides a more accurate approximation for the number of genes in a genome.

$$R = \frac{Length\ of\ Identified\ ORFs\ (BP)}{Length\ of\ Genome\ (BP)} \quad (1) \qquad g = \frac{Length\ of\ Genome\ (BP)}{(\frac{Average\ Length\ Of\ Protein\ CDS}{R})} \quad (2)$$

Equation 1 describes the ratio between the number of Base Pairs (BPs) in an identified ORF and the number of Base Pairs in the genome. This ratio is used in Equation 2 to calculate an approximate number of genes (g) in the genome, given an average length of protein CDS measured in BP.

The code found in D describes a program which takes as input the genome FastA file, the FastA file of ORFs generated by Prodigal and Bedtools, and the Blast+ output processed by the programs in Appendix B and Appendix C. It uses these inputs to identify ORFs which have been significantly aligned to at least one protein CDS. The program then calculates an average length for all identified protein CDS and applies this information to Equations 1 and 2 to calculate an approximate number of genes for the genome FastA file.

The approximate number of genes can then be divided by the total length of all sequences in the genome FastA file measured in Mega Base Pairs to calculate an approximate gene density, δ (Equation 3).

$$\delta = \frac{g}{Length\ of\ genome\ (MBP)} \quad (3)$$

Appendix E describes examples of commands needed in order to run the pipeline. These commands are presented in the appropriate order.

RESULTS

Table 1 shows the total number of significant alignments that obtained an *e*-value lower than a threshold value of $1e$-5 and the number of nonsignificant alignments which scored higher than than the threshold value. The genomes with shorter overall lengths of ORFs (RUG005 and RUG466) see a higher percent of significant alignments compared to the others, although the there is no significant difference between the genomes. As shown in Table 2, RUG005 and RUG466 also had shorter ORFs compared to RUG154 and RUG431, with the ORFs identified in RUG005 constituting the smallest percent of the genome (80.75%).

Genome	RUG005	RUG154	RUG431	RUG466
Significant Alignments	251,004	309,417	230,145	229,716
Nonsignificant Alignments	85,190	116,488	97,314	84,401
Percent Significant / %	74.66	72.65	70.28	73.13

Table 1. *The number and percentage of alignments with an e-value of <1e-5.*

Genome	RUG005	RUG154	RUG431	RUG466
Total Genome Length / BP	2,345,410	2,774,557	2,770,634	1,925,880
Total ORF Length / BP	1,893,858	2,504,499	2,562,720	1,711,224
Percent ORFs / %	80.75	90.27	92.50	88.85

Table 2. *The length of ORFs and what percentage of the original genome file they make up.*

Table 3 lists the average protein CDS length and the approximate gene count for each genome, calculated by the program found in Appendix D. Combining this information with the Total Genome Length in MBP (Table 2), an approximate gene density is calculated for each genome.

Genome	RUG005	RUG154	RUG431	RUG466
Average Protein CDS Length	1045.54	1060.75	1261.08	1029.03
Approximate Gene Count	1811.36	2361.06	2032.16	1662.95
Approximate Gene density / $gMBP^{-1}$	503.69	850.97	733.46	863.48

Table 3. *Approximate Gene Count and Gene Density (in Genes per Mega BPs) given the Average length of Protein CDS for each genome.*

This pipeline can be verified by processing reference genomes with known metrics. The genome of S. Cerevisiae (Engel et al., 2022) as well as a curated file of Coding Sequences are available through the National Center for Biotechnology Information (NCBI) database (Sayers et al., 2022). The results of processing the full genome with Prodigal and Bedtools to extract ORFs can be compared against the FastA file containing CDS provided by NCBI. Both the ORF and CDS files can be aligned using a BLAST+ search against the Swiss-Prot database. The output of that alignment search can be processed through the Python pipeline described above to gather an approximate gene count and gene density (Table 4).

Due to a limitation of available hardware, the S. Cerevisiae genome FastA file was separated into five sequentially numbered files that were then processed through the `gene_calculator.py` script in Appendix D. The results of these five operations were then combined and are shown in Table 4. The same process was applied to the S. Cerevisiae CDS FastA file.

DISCUSSION

Using metrics calculated from the four MAGs and comparing those to metrics of a known genome (S. Cerevisiae), the pipeline calculated the gene density to within $\approx 8\%$ of the expected value. These metrics allow for poorly assembled MAGs to be removed from classification training data, improving the ability of classification. This in turn makes the process of sampling, assembling, and identifying the MAGs easier and more accurate.

One major limitation of this paper is the imprecision of the method used to calculate the number of genes for a genome. The equations put forth by Santos-Magalhaes and de Oliveira (2015) originally use

Genome	S. Cerevisiae	S. Cerevisiae CDS
Significant Alignments	531,043	535,281
Nonsignificant Alignments	275,416	260,454
Percent Significant / %	65.85	67.27
Total Genome Length / BP	12,157,105	8,826,477
Total ORF Length / BP	9,154,275	8,770,452
Percent ORFs / %	75.30	99.37
Average Protein CDS Length	1397.78	1459.62
Approximate Gene Count	6549.14	6008.73
Approximate Gene Density / $gMBP^{-1}$	538.71	494.26 †

Table 4. *Metrics produced by the developed pipeline for the S. Cerevisiae genome.*
† Value calculated using the Total Length of the S. Cerevisiae genome, not the Total Length of the S. Cerevisiae CDS genome, which is already a subset of the full S. Cerevisiae genome.

average values for the length of protein CDS of a taxonomic domain. This paper alters these equations to instead utilise the average length of protein CDS of a genome. While this alteration does allow for a more appropriate value it is still hampered by using the average length of protein CDS across the genome. By using the average lengths it is likely that the true number of genes in a genome is not calculated, which makes it harder to accurately calculate the gene density of a genome and harder again to accurately evaluate the High-C binning used by Stewart et al. (2018) to create the MAGs in the first place. This can be seen with the results of processing the S. Cerevisiae genome as seen in Table 4

Table 4 shows the results of processing the S. Cerevisiae genome in the same manner as the genomes from Stewart et al. (2018). The calculated approximate gene count of 6,549.14 is greater than the actual protein producing gene count of 6,014 recorded by Engel et al. (2022). This difference is likely due to the threshold value set during the filtering stage of the process. Decreasing the threshold from an e-value of $1e$-5 to a value of $1e$-10 may reduce the number of ORFs that found significant alignments during the BLAST+ search, thus decreasing the number of identified protein producing Coding Sequences. This in turn would reduce the number of approximate genes in a genome.

This larger number of genes will also create a larger than expected gene density. Using the gene count of 6,014 combined with total genome length of 12,071,326 BP we can calculate a more accurate gene density of 498.21 $gMBP^{-1}$, a difference of 40.5 $gMBP^{-1}$ from the approximation produced by the pipeline. This difference of $\approx 8\%$ is large, although it does not invalidate the results generated for the four RUG genomes as it would not significantly alter the genomic density for the RUG genomes.

The NCBI RefSeq database also provides the ability to download a curated FastA file that contains only protein CDS. Table 4 shows the metrics produced by the pipeline when processing this CDS FastA file. The pipeline is able to align 99.37% of the curated protein CDS to the Swiss-Prot protein database (The UniProt Consortium, 2020). The approximate gene density calculated for the CDS FastA of 494.26 falls within the expected margin of error ($\approx 8\%$) for the pipeline with these parameters.

Given an error margin of 40.5 $gMBP^{-1}$ for the gene densities of the RUG genomes, the gene density of all four RUG genomes falls within the expected range of 500-1,000 $gMBP^{-1}$ for prokaryotic genomes (O'Leary et al., 2016). These gene densities show that the High-C binning method employed by Stewart et al. (2018) has been able to produce genomes similar in gene composition to validated prokaryotic genomes in the RefSeq database O'Leary et al. (2016).

CONCLUSION

The method proposed in this paper is capable of calculating the gene density of an unknown genome given the alignment of the genome to a protein database. This allows for the analysis of Metagenome-Assembled Genomes by calculating the gene density of reference genomes, either manually using statistics calculated by organisations like NCBI or by processing the reference genomes through the same pipeline as the unknown genome.

The four MAGs produced by Stewart et al. (2018) were found to be within the expected range of gene density for prokaryotic genomes. This similarity shows the High-C binning technique used to assemble the four MAGs is able to accurately assemble reads from the rumen of cattle into distinct genomes.

ADDITIONAL GENE INVESTIGATIONS

Code Overview

The code consists of two Python scripts, `part_one.py` and `similarity.py`. They have been written using Python 3.11 and are designed to be run from the command line. Both scripts should be located in the same directory, FastA files may be located elsewhere, as long as the file path passed to the script is correct. Output files may be sent directly to subdirectories of the `outputs` directory provided the subdirectory already exists.

part_one.py

`part_one.py` should then be run with a single FastA input file, an output filename, and a K value for K-Mer composition operations. An example can be found in Appendix A, Listing 1.

This will run the `part_one.py` script on the `genome_1.fa` file, with the statistics generated by the script written to `./outputs/genome_1_output.txt`. The `outputs` directory will be created in the directory the command was executed from. Running the example command (Appendix A, Listing 1) will also create a file called `genome_1_dict.csv` which is a Comma Separated Values (CSV) file containing all unique K-Mers found in the corresponding FastA file. Each line will begin with the K-Mer in question followed by the number of occurrences of that K-Mer in the provided FastA file.

`part_one.py` generates the GC-Content Mean, GC-Content Standard Deviation (SD), N50, N90, and L50 statistics. These are written to the given output file, with the name of the provided FastA file on line 1, then the GC-Content Mean, GC-Content SD, N50, N90, and L50 statistics, each on a new line. The file ends on a blank line. An example of the output file can be seen in Appendix A, Listing 2.

similarity.py

To compare the Manhattan Distance between any number of genomes, `similarity.py` will be required. The `similarity.py` script takes the `_dict` outputs of `part_one.py` and uses them to calculate the distance between genomes. The `similarity.py` script can be passed a filename for the results to be written too, and two or more `_dict` files and will return a list of each comparison, and the distance between the compared genomes. An example can be found in Appendix A, Listing 3.

The example from Appendix A, Listing 3 uses `similarity.py` to calculate the distance between: `g1_dict` and `g2_dict`; `g1_dict` and `g3_dict`; `g2_dict` and `g3_dict`. These will be returned to the user as a CSV file in the `outputs` directory with the name which was given as the first argument in the example command (`distances.csv`). An example section of this file can be found in Appendix A, Listing 4.

Third Party Libraries

The `part_one.py` script makes use of Pysam (John Marshall, 2023) to read in the sequence names and sequence strings from a given FastA file. Both scripts also utilise NumPy (Harris et al., 2020) for some operations.

Statistics

GC-Content

As shown in Table 5, the mean GC-Contents for each genome vary by 9.014%. Comparing the two GC-Content SDs that are most different - RUG005 (2.4090) and RUG154 (1.1743) - with the Hartley's F-Max test gives an F_{Max} value of 4.10. With over one hundred degrees of freedom, the critical value for this comparison would be 1.00. This means that the GC-Content SD between the two most divergent are statistically heterogenous. This also applies to the RUG466 genome, which is statistically heterogenous from the RUG154 genome but not the RUG005 or RUG431 genomes.

Genome	RUG005	RUG154	RUG431	RUG466
GC-Content Mean / %	48.935	44.288	51.058	57.949
GC-Content SD	2.4090	1.1743	1.9024	2.3720

Table 5. *Table showing the GC-Content means and Standard Deviations per genome.*

Metrics

While it is not possible to compare the N50 and L90 metrics found in Table 6, because of the differing numbers of contigs in each FastA file, it is possible to compare the L50 metrics. The RUG466 genome has the highest L50 value, as well as the lowest N50 and N90 values. This implies that the RUG466 genome has been assembled from shorter and therefore less accurate reads compared to the others. This is consistent with genome having the highest mean average GC-Content (Table 5), which might be expected given the Hi-C-based proximity-guided assembly employed by Stewart et al. (2018).

Genome	RUG005	RUG154	RUG431	RUG466
N50	46061	43578	52738	16904
N90	12229	15257	22961	5244
L50	12	24	15	31

Table 6. *Table showing the N50, N90, and L50 metrics for each genome.*

Genome Similarity

Table 7 shows the Manhattan Distance between each of the four genomes. The least similar genomes are the RUG466 and RUG154 genomes, with a distance of 1,607,019. This mirrors the findings of other metrics which, using GC-Content, showed the RUG466 and RUG154 genomes to be statistically heterogenous.

Genome	RUG005	RUG154	RUG431	RUG466
RUG005	0	541788	547265	628785
RUG154	541788	0	456163	1067019
RUG431	547265	456163	0	908798
RUG466	628785	1067019	908798	0

Table 7. *The Manhattan distance between each of the four genomes with regards to 2-Mer frequencies.*

A ADDITIONAL GENETIC INVESTIGATIONS EXAMPLES

```
$ python3 part_one.py data/gene_1.fa gene_1_output.txt 2
```

Listing 1. Example command line input to run `part_one.py`

```
genome_1.fa
12.34565432102345
1.234567654321012
98765
12345
13579

```

Listing 2. Example contents of an output file generated by `part_one.py`

```
$ python3 similarity.py distances.csv gene_1_dict.csv gene_2_dict.csv gene_3_dict.csv
```

Listing 3. Example command line input to run `similarity.py`

```
AAAAAAA,494
AAAAAAC,329
AAAAAAG,791
AAAAAAT,777
AAAAACA,347
AAAAACC,275
AAAAACG,248
AAAAACT,434
AAAAAGA,852
AAAAAGC,654
AAAAAGG,670
AAAAAGT,424
AAAAATA,625
AAAAATC,663
AAAAATG,544
AAAAATT,540
AAAACAA,211
AAAACAC,94

```

Listing 4. Example contents of an output file generated by `similarity.py`

B `FILTERER.PY` CODE

```python
import sys
import re


def filter_bad_matches(filename):
    """Removes any alignments from a blastx alignment that are above a given threshold
    ,
    writes a file with the name of the input file with the '_filtered.txt' suffix
    instead
    :param filename: filename of blastx alignments to be filtered by e-value
    :return: good_matches: number of alignments below the threshold value
    :return: bad_matches: number of alignments above the threshold"""

    # alter this float to desired value
    threshold = float(1e-5)
    # pattern matches the pairwise alignments produced from a blastx search
    pattern = re.compile(r"^([A-Z0-9]+)\s+([^\n]+)\s+(\d+\.?\d+)\s+([^\n]+)$", re.
    MULTILINE)
    good_matches = 0
    bad_matches = 0
    with open(filename, 'r') as infile:
        outfile_name = filename.split('.')[0] + '_filtered.txt'
        with open(outfile_name, 'w') as outfile:
            # could be converted to tqdm  (or other) progress bar
            print("Filtering file ...")
            first_line = True
            for i, line in enumerate(infile):
                if line.startswith('Query='):
                    # we don't want to start the file with a blank line
                    if first_line:
                        outfile.write(line)
                        first_line = False
                    else:
                        # but we do want subsequent queries to be separated by a blank
     line
                        outfile.write("\n" + line)
                for match in re.finditer(pattern, line):
                    # once we have matched a query line, then write all its good
    alignments
                    if float(match[4]) < threshold:
                        outfile.write(match.group() + "\n")
                        good_matches += 1
                    else:
                        bad_matches += 1

    return good_matches, bad_matches


if __name__ == '__main__':
    good, bad = filter_bad_matches(sys.argv[1])
    print("good: "+str(good)+
            "\nbad:  "+str(bad))
    print("Percent Good = "+str('{0:.2f}'.format(good/(good+bad)*100)))

```

Listing 5. filterer.py code

C `TRIMMER.PY` CODE

```python
import sys
import re


def trim_file(filename):
    """Trims the Pairwise output file of a blastx search and extracts only the query
    names
    and the list of significant alignments and writes them to a new file
    :param filename: filename of the alignment file to be trimmed"""

    pattern = re.compile(r"^([A-Z0-9]{6,})\s+([^\n]+)\s+(\d+\.?\d+)\s+([^/])([^\n]+)$"
    , re.MULTILINE)
    with open(filename, 'r') as infile:
        outfile_name = filename.split('.')[0] + '_trimmed.txt'
        with open(outfile_name, 'w') as outfile:
            print("Trimming file ...")
            first_line = True
            for i, line in enumerate(infile):
                if line.startswith('Query='):
                    if first_line:
                        outfile.write(line.strip() + "\n")
                        first_line = False
                    else:
                        outfile.write("\n" + line)
                for match in re.finditer(pattern, line):
                    outfile.write(match.group().strip() + "\n")

    print("File trimmed!")


if __name__ == '__main__':
    trim_file(sys.argv[1])

```

Listing 6. trimmer.py code

D GENOMIC_CALCULATOR.PY CODE

```
import sys
import pysam
import part_one as po
import re


def process_gene_orf_inputs(genome_path, orf_path):
    """Calls part_one.py to process the FastA file and the corresponding FastA file of
     ORFs
    :param genome_path: path to the FastA genome file, passed as the first cli
    argument
    :param orf_path: path to the FastA genome ORF file, passed as the second cli
    argument
    :return: genome_sequence_names: list of sequence names from the genome FastA file
    :return: orf_sequence_names: list of ORF sequence names from the ORF FastA file
    :return: genome_file: Pysam object representing the genome FastA file
    :return: orf_file: Pysam object representing the ORF FastA file"""

    genome_sequence_names = po.get_sequence_names(genome_path)
    orf_sequence_names = po.get_sequence_names(orf_path)
    genome_file = pysam.FastaFile(genome_path)
    orf_file = pysam.FastaFile(orf_path)

    return genome_sequence_names, orf_sequence_names, genome_file, orf_file


def process_filtered_orf_inputs(filtered_orf_path):
    """Processes the filtered file of alignments and returns a list of ORFs that had
    at least one
    filtered significant alignment
    :param filtered_orf_path: path to the filtered file of alignments produced by
    blastx,
    filterer.py and trimmer.py
    :return: protein_producers: list of identified ORFs that had at least one filtered
    significant alignment
    :return: chance_orfs: list of identified ORFs that had no filtered significant
    alignments"""

    pattern = re.compile(r"^([A-Z0-9]+)\s+([^\n]+)\s+(\d+\.?\d+)\s+([^\n]+)$", re.
    MULTILINE)
    protein_producers = []
    chance_orfs = []
    with open(filtered_orf_path, 'r') as infile:
        lines = infile.readlines()
        for index, line in enumerate(lines):
            if line.startswith('Query=') and lines[index] != lines[len(lines)-1]:
                next_line = lines[index + 1]
                if re.match(pattern, str(next_line)):
                    protein_producers.append(line[7:].strip())
                else:
                    chance_orfs.append(line[7:].strip())

    return protein_producers, chance_orfs


def get_average_length_of_protein_cds(protein_producers, orf_file, orf_sequence_names)
    :
    """Calculates the average length of protein producing ORFs across a genome
    :param protein_producers: list of protein producing ORFs, from
    process_filtered_orf_inputs()
    :param orf_file: file of all identified ORFs, from process_gene_orf_inputs()
    :param orf_sequence_names: list of ORF sequence name strings, from
    process_filtered_orf_inputs()
    :return: float value for the average length of protein coding sequences"""

    length_of_protein_cds = 0
    print("orf sequences              = " + str(len(orf_sequence_names)))
```

```
    print("protein sequences          = " + str(len(protein_producers)))
    for sequence in protein_producers:
        for orf in orf_sequence_names:
            if orf.find(sequence) != -1:
                read = orf_file.fetch(orf)
                length_of_protein_cds += len(read)

    return length_of_protein_cds / len(protein_producers)


def calc_approx_gene_count(genome_length, average_cds_length, orf_length):
    """Calculates an approximate gene count given the number of protein producing
    coding sequences
    and the total length of all identified ORFs and the total length of the genome in
    base pairs
    :param genome_length: total length of the genome in base pairs, from
    get_num_useful_sequences()
    :param average_cds_length: average length of protein CDS, from
    get_average_length_of_protein_cds()
    :param orf_length: total length of ORFs in base pairs, from
    get_num_useful_sequences()
    :return g: float value of the approximate gene count"""

    g = (genome_length / (average_cds_length / (orf_length / genome_length)))
    return g


def get_sequence_lengths(genome_sequence_names, orf_sequence_names, genome_file,
    orf_file):
    """Calculates the total length of ORFs and the total length of the whole genome,
    both in base pairs
    :param genome_sequence_names: list of sequence names from the genome FastA, from
    process_gene_orf_inputs()
    :param orf_sequence_names: list of sequence names from the ORF FastA, from
    process_gene_orf_inputs()
    :param genome_file: Pysam object representing the genome FastA file, from
    process_gene_orf_inputs()
    :param orf_file: Pysam object representing the ORF FastA, from
    process_gene_orf_inputs()
    :return genome_length: total length of the genome in base pairs
    :return orf_length: total length of all identified ORFs in base pairs"""

    genome_length = 0
    orf_length = 0

    for sequence_name in genome_sequence_names:
        read = genome_file.fetch(sequence_name)
        genome_length += len(read)

    for sequence_name in orf_sequence_names:
        read = orf_file.fetch(sequence_name)
        orf_length += len(read)

    return genome_length, orf_length


if __name__ == '__main__':
    (genome_sequence_names, orf_sequence_names,
        genome_file, orf_file) = process_gene_orf_inputs(sys.argv[1], sys.argv[2])

    genome_length, orf_length = get_sequence_lengths(genome_sequence_names,
    orf_sequence_names,
                                                     genome_file, orf_file)
    # prints the different metrics, aligned to each other by the '=' sign for
    readability
    # could be modified to output these in a csv file instead (like part_one.
    write_statistics() does)
    print("total genome length       = " + str(genome_length))
    print("total orf length          = " + str(orf_length))

```

```
    filtered_orf_path = sys.argv[3]
    protein_producers, chance_orfs = process_filtered_orf_inputs(filtered_orf_path)

    average_cds_length = get_average_length_of_protein_cds(protein_producers, orf_file
    , orf_sequence_names)
    print("Average protein CDS length = "+str(average_cds_length))

    gene_count = calc_approx_gene_count(genome_length, average_cds_length, orf_length)
    print("Gene count                 = " + str(gene_count))
```

Listing 7. genomic_calculator.py code

E EXAMPLE PIPELINE COMMANDS

This supplemental file shows an example workflow to follow when using the pipeline. These commands are applicable to the MacOS zsh 2.14 (452) Terminal, but are translatable to other Unix derived operating systems.

Third Party Command Line Programs

```
prodigal -i genome_1.fa -o genome_1.gff3 -f gff
```

Listing 8. Example Prodigal Command

```
bedtools getfasta -fi genome_1.fa -bed genome_1.gff3 -s -fo genome_1.orfs.fa
```

Listing 9. Example Bedtools Command

```
time blastx -query genome_1.fa -db uniprot_sprot -out genome_1_matches.txt
```

Listing 10. Example Blast+ Command

Inclusion of the `time` command is optional, but handy when running larger BLAST+ alignments on less powerful hardware. the `uniprot_sprot` database is the Swiss-Prot protein database.

The Pipeline

The proprietary Python scripts developed for the pipeline must be located in the same level directory, in this example they are all located in the `/[my_pipeline]/code/` directory. They are all run from the top level directory (`/[my_pipeline]/`).

```
python3 code/trimmer.py outputs/genome/genome_1_matches.txt
```

Listing 11. Example trimmer.py Command

```
python3 code/filterer.py outputs/genome/genome_1_matches_trimmed.txt
```

Listing 12. Example filterer.py Command

```
python3 code/genomic_calculator.py data/genome/genome_1.fa data/genome/genome_1.orfs.
    fa outputs/genome/genome_1_matches_trimmed_filtered.txt
```

Listing 13. Example genomic_calculator.py Command

The `genomic_calculator.py` script takes three command line arguments. The first is the path from the user's present working directory to the FastA file containing the genome. The second is the path to the FastA file of Open Reading Frames generated by Prodigal and Bedtools. The final argument is the path to the trimmed and filtered file of alignments generated by the BLAST+ alignment search. The

Optional Pipeline Files

```
python3 code/splitter.py data/genome/genome_1.fa
```

Listing 14. Example splitter.py Command

`splitter.py` will take a FastA file and divide it into a given number of sequentially numbered FastA files - i.e. genome_1.fa becomes: genome_1_1.fa, genome_1_2.fa. This may be needed if the original FastA genome file is too large to be processed on the user's hardware. Through testing a FastA filesize of $< 5MB$ was desirable. Above this threshold the Pysam third party library may run out of memory and report an unexpected end of file error. If the original FastA file must be processed with the splitter then each sequential FastA file must be run through the `genomic_calculator.py` individually, although the second and third arguments need not change.

```
python3 code/clean_query.py outputs/genome/genome_1_matches_trimmed_filtered.txt
```

Listing 15. Example clean_query.py Command

The `clean_query.py` and script should be run if there is a mismatch in the sequence names between the three different files that are input to the `genomic_calculator.py` script. `clean_query.py` takes the processed `matches_trimmed_filtered.txt` file and renames the sequences to remove improper characters, such as any "[]" characters and any characters in between these. This is needed as the `genomic_calculator.py` must compare the sequence names for some operations. To change what the offending characters are, open `clean_query.py` with a preferred editor and update the RegEx `pattern` variable on line 5.

REFERENCES

Camacho, C., Coulouris, G., Avagyan, V., Ma, N., Papadopoulos, J., Bealer, K., and Madden, T. L. (2009). Blast+: architecture and applications. *BMC Bioinformatics*, 10(1):421.

Engel, S. R., Wong, E. D., Nash, R. S., Aleksander, S., Alexander, M., Douglass, E., Karra, K., Miyasato, S. R., Simison, M., Skrzypek, M. S., Weng, S., and Cherry, J. M. (2022). New data and collaborations at the saccharomyces genome database: updated reference genome, alleles, and the alliance of genome resources. *Genetics*, 220(4).

Harris, C. R., Millman, K. J., van der Walt, S. J., Gommers, R., Virtanen, P., Cournapeau, D., Wieser, E., Taylor, J., Berg, S., Smith, N. J., Kern, R., Picus, M., Hoyer, S., van Kerkwijk, M. H., Brett, M., Haldane, A., del Río, J. F., Wiebe, M., Peterson, P., Gérard-Marchant, P., Sheppard, K., Reddy, T., Weckesser, W., Abbasi, H., Gohlke, C., and Oliphant, T. E. (2020). Array programming with NumPy. *Nature*, 585(7825):357–362.

Hyatt, D., Chen, G.-L., LoCascio, P. F., Land, M. L., Larimer, F. W., and Hauser, L. J. (2010). Prodigal: prokaryotic gene recognition and translation initiation site identification. *BMC bioinformatics*, 11:1–11.

John Marshall, A. H. (2023). Pysam. https://github.com/pysam-developers/pysam.

O'Leary, N. A., Wright, M. W., Brister, J. R., Ciufo, S., Haddad, D., McVeigh, R., Rajput, B., Robbertse, B., Smith-White, B., Ako-Adjei, D., Astashyn, A., Badretdin, A., Bao, Y., Blinkova, O., Brover, V., Chetvernin, V., Choi, J., Cox, E., Ermolaeva, O., Farrell, C. M., Goldfarb, T., Gupta, T., Haft, D., Hatcher, E., Hlavina, W., Joardar, V. S., Kodali, V. K., Li, W., Maglott, D., Masterson, P., McGarvey, K. M., Murphy, M. R., O'Neill, K., Pujar, S., Rangwala, S. H., Rausch, D., Riddick, L. D., Schoch, C., Shkeda, A., Storz, S. S., Sun, H., Thibaud-Nissen, F., Tolstoy, I., Tully, R. E., Vatsan, A. R., Wallin, C., Webb, D., Wu, W., Landrum, M. J., Kimchi, A., Tatusova, T., DiCuccio, M., Kitts, P., Murphy, T. D., and Pruitt, K. D. (2016). Reference sequence (refseq) database at ncbi: current status, taxonomic expansion, and functional annotation. *Nucleic Acids Res.*, 44(D1):D733–45.

Quinlan, A. R. and Hall, I. M. (2010). Bedtools: a flexible suite of utilities for comparing genomic features. *Bioinformatics*, 26(6):841–842.

Santos-Magalhaes, N. S. and de Oliveira, H. M. (2015). Of protein size and genomes. *arXiv preprint arXiv:1502.03732*.

Sayers, E. W., Bolton, E. E., Brister, J. R., Canese, K., Chan, J., Comeau, D. C., Connor, R., Funk, K., Kelly, C., Kim, S., Madej, T., Marchler-Bauer, A., Lanczycki, C., Lathrop, S., Lu, Z., Thibaud-Nissen, F., Murphy, T., Phan, L., Skripchenko, Y., Tse, T., Wang, J., Williams, R., Trawick, B. W., Pruitt, K. D., and Sherry, S. T. (2022). Database resources of the national center for biotechnology information. *Nucleic Acids Res.*, 50(D1):D20–D26.

Stewart, R. D., Auffret, M. D., Warr, A., Wiser, A. H., Press, M. O., Langford, K. W., Liachko, I., Snelling, T. J., Dewhurst, R. J., Walker, A. W., Roehe, R., and Watson, M. (2018). Assembly of 913 microbial genomes from metagenomic sequencing of the cow rumen. *Nature Communications*, 9(1):870.

The UniProt Consortium (2020). UniProt: the universal protein knowledgebase in 2021. *Nucleic Acids Research*, 49(D1):D480–D489.