

What's in my BLAST results? A bioinformatics analysis tool.

Final Report for CS39440 Major Project

Author: Jakub Kawka (jak30@aber.ac.uk)

Supervisor: Dr. Amanda Clare (afc@aber.ac.uk)

3rd May 2018

Version 1.0 (Final)

This report is submitted as partial fulfilment of a BEng degree in
G600 Software Engineering (Inc. Integrated Industrial and Professional Training)

Department of Computer Science
Aberystwyth University
Aberystwyth
Ceredigion
SY23 3DB
Wales, UK

Declaration of originality

I confirm that:

- This submission is my own work, except where clearly indicated.
- I understand that there are severe penalties for Unacceptable Academic Practice, which can lead to loss of marks or even the withholding of a degree.
- I have read the regulations on Unacceptable Academic Practice from the University's Academic Quality and Records Office (AQRO) and the relevant sections of the current Student Handbook of the Department of Computer Science.
- In submitting this work, I understand and agree to abide by the University's regulations governing these issues.

Name Jakub Kawka

Date 20/04/2018

Consent to share this work

By including my name below, I hereby agree to this dissertation being made available to other students and academic staff of the Aberystwyth Computer Science Department.

Name Jakub Kawka

Date 20/04/2018

Acknowledgements

I am grateful to my supervisor, Amanda Clare, for the feedback and guidance throughout the project.

I would like to thank Sam Nicholls for walking me through the initial stages of understanding BLAST & Bioinformatics.

I would also like to thank my old computer science Facebook group chat "Comp Sci Miami" - particularly Jon Tetrino, Gareth James Jones, and Oliver Earl. Useless lot when it comes Bioinformatics, but they definitely made up for it in moral support.

Abstract

Bioinformatics, alongside big data and machine learning – all of which feature a heavy overlap, is currently one of the fastest growing fields in computer science. [1] Nowadays there exist a multitude of various bioinformatics tools, yet due to their relative lack of maturity and the fact they are primarily written “by scientists, for scientists”, many suffer from the common functionality and usability pitfalls that often besets research software. [2]

The Basic Localised Alignment Search Tool [3], more commonly known as BLAST, refers to both the algorithm as well as the tool itself, and is used to compare given nucleotide (DNA) and amino acid (protein) sequences with those indexed in an existing database. BLAST is one of the most commonly used tools in the field and forms a crucial cornerstone of modern bioinformatics.

Nevertheless, BLAST is an aged and monolithic piece of software and the results provided are fairly basic. A list of sequences that matched and their respective matches, as well as a handful of scores identifying match accuracy.

The aim of this project is to take existing BLAST data and augment it with additional useful information, starting off with some of the simpler enhancements before moving onto more advanced features such as overall statistics, phylogenetic breakdowns, isolating gene functions, or identifying protein domains. The results will be provided in the form of a text output, or ideally a web front-end visualisation.

In the end, not all of these features have been accomplished and the program still needs some quality-of-life polish, but the progress made so far has been promising and serves as a solid foundation to further expand upon.

Terminology

Alignment – A found match between the query and the subject. Contains one or more hits/HSPs.

Bioinformatics – “Research, development, or application of computational tools and approaches for expanding the use of biological, medical, behavioral or health data, including those to acquire, store, organize, archive, analyze, or visualize such data.” as defined by the National Center for Biotechnology Information [4]

Hit – Also known as High Scoring Pair(HSP). A particular high-scoring local match between the target and query sequences.

HSP – High Scoring Pair. Also known as a hit. A particular high-scoring local match between the target and query sequences.

NCBI – National Center for Biotechnology Information

Phylogenetics – A study of evolutionary history and relationships between living organisms.

Query – The biological sequence used in the BLAST comparison.

Subject – Also known as target. The biological sequence that a query has been compared and matched against.

Target – Also known as subject. The biological sequence that a query has been compared and matched against.

Table of Contents

1. BACKGROUND, ANALYSIS, AND PROCESS	8
1.1. BLAST	8
1.1.1. FASTA & Biology Review	8
1.1.2. BLAST Algorithm	10
1.1.3. BLAST Tools	10
1.1.4. BLAST Outputs	11
1.2. ANALYSIS	14
1.2.1. Databases and Results	14
1.2.2. Problem Domain	14
1.2.3. Similar Existing Software	15
1.2.4. Initial Proposed Features	16
1.3. PROCESS	18
2. DESIGN	18
2.1. OVERALL ARCHITECTURE	18
2.2. LANGUAGE CHOICE	19
2.3. BLAST OUTPUT FORMATS	20
2.4. USER INTERFACE	24
2.5. NCBI TAXONOMY DATABASE	25
2.6. INDEXING MOST/LEAST COMMON RESULTS	25
2.6.1. Edit Distance Based Methods	26
2.6.2. N-Gram Based Approaches	27
2.6.3. Bag-Of-Words Machine Learning	27
2.7. FINAL DESIGN	28
2.8. OTHER	29
2.8.1. Developer Hardware	29
2.8.2. Developer Environment	29
2.8.3. Backup And Version Control	29
2.8.4. Compatibility	29
3. IMPLEMENTATION	30
3.1. BIOPYTHON	30
3.2. RECREATING THE TABULAR OUTPUT	32
3.3. TOTAL QUERY COVERAGE	33
3.4. COUNTING INDIVIDUAL GAPS	33
3.5. BAG-OF-WORDS MATCHING	34
3.5.1. The Taxonomy Database	34
3.5.2. Cleaning The Results	37
3.5.3. Matching The Results	37
3.5.4. Performance	38
3.5.5. Accuracy	39
3.6. GENERATING LINEAGES	40
3.7. RANKED LINEAGES	40
3.8. TAXONOMY TREE	41
3.9. MOST/LEAST COMMON RESULTS	41
3.9.1. The Algorithm	42
3.10. VISUALISATIONS	43
3.10.1. ASCII	44
3.10.2. ETE Tree	45
3.11. INTERFACE	48
3.12. FINAL THOUGHTS	48

4. TESTING	49
4.1. AUTOMATED TESTS.....	49
4.2. MANUAL TEST CASES	50
5. CRITICAL EVALUATION	52
5.1. OBJECTIVES	52
5.2. PROCESS	53
5.3. TOOLS	53
5.4. FUTURE WORK	54
6. APPENDICES.....	55
6.1. THIRD-PARTY CODE, LIBRARIES AND TECHNOLOGIES USED.....	55
6.2. CODE SAMPLES	57
6.2.1. <i>Iterating Over The BLAST Results</i>	57
6.2.2. <i>Providing The Tabular Output</i>	58
6.2.3. <i>Cleaning The Results</i>	58
6.2.4. <i>Sample Of The FullNameLineage.dmp File</i>	59
6.2.5. <i>Sample Of The RankedLineage.dpm File</i>	60
6.2.6. <i>Processing The Ranked Lineage Data</i>	61
6.3. ETHICS FORM.....	62
7. BIBLIOGRAPHY.....	64

1. Background, Analysis, And Process

1.1. BLAST

A comprehensive knowledge of bioinformatics and BLAST is not required, but nevertheless some background information needs to be established before one can delve into the main project. This shall be explained in relative layman terms, as explained by a layman, and does not constitute a fully accurate overview, though it will strive for a degree of relative accuracy.

For clarity, the algorithm itself, as well as, 'BLAST+' - the official National Center for Biotechnology Information (NCBI) software release of the algorithm, will both be referred to as BLAST.

1.1.1. FASTA & Biology Review

BLAST sequences are stored in FASTA file format. This stands for 'Fast-A', which further stands for 'Fast-All', in reference to the depreciated individual formats 'Fast-N' (Nucleotide) and 'Fast-P' (Protein) and the associated FASTA algorithm/tool that served as BLAST's predecessor. [5] FASTA has since gone on to become one of the few established bioinformatics standards.

Though a BLAST database first needs to be specifically pre-processed and indexed by the BLAST tool, both the query and target database are derived from a FASTA file. Given a pair of FASTA files, it is possible to run one against the other and vice versa.

FASTA and BLAST support both nucleotide and protein sequences.

Here is an example of a nucleotide sequence. A sequence identifier, beginning with '>', followed by a sequence of letters representing **A**denine, **C**ytosine, **G**uanine and **T**hymine which make up the four bases of DNA. When dealing with mRNA instead, Thymine is replaced with **U**racil. The sequence identifier can be anything, though most common databases tend to employ a consistent format – e.g. GenBank uses '*gb|accession|locus*'

>This is a random sequence taken from HIV genome

GTAATTAGATCCGCCAATTTACAGACAATACTAAAATCATAATAGTACAGCT

There are also some special characters such, as **R** which stands for '**A** or **G**' or **B** which stands for "Not **A**", though they are relatively rare.

Here is a common example of a protein sequence. The letters of the alphabet correspond to 25 different amino acids with three additional special characters such as '*' which indicates the end of a sequence.

>MCHU - Calmodulin - Human, rabbit, bovine, rat, and chicken

**ADQLTEEQIAEFKEAFSLFDKDGDTITTKELGTVMRSLGQNPTEAELQDMINEVD
ADGNGTIDFPEFLTMMARKMKDSTDSEEEIREAFRVFDKDGNGYISAAELRHVMTN
LGEKLTDEEVDEMIREADIDGDGQVNYEEFVQMMTAK***

Of note, is the concept of 'codons' which means that a single amino acid will correspond to three consecutive nucleotide bases. A single amino acid can represent more than one set of bases. In the case of Glutamine, these are **CAA** and **CAG**. Thus, converting mRNA back to its base amino acids is a relatively trivial undertaking, but attempting to extrapolate a nucleotide sequence from a set of amino acids is considerably more difficult.

```
>NODE_1_length_180_cov_7.511111
CTCCCTCTTTTTTCGGATATGCTGGTGATTTGGCGACGAACAGTATCTGGGCAGCTCTAT
TTATCTCGAAGGCAGTTTATTTAATAAAATATTGTGGCGAAAATATTCACCTACTCCGTC
GGTTTGCCAGTTTCTATCTGGTCGGATAAACTAGTGATTTGGCGACAGGCGGTATCGGG
TAAGCTTGTTTTATCCCGACGATCAGTGCCTGAAATACTGACAGATACTG
>NODE_3_length_74_cov_9.391891
ATTAAAAAGGGTGTAATTGAATATTATAATATGTATAGGAGTCGGATAGTCGTGACCTA
GTTTGTCGCTCTGGTCAACTGTTGACGAGAAGCGTGTAAGAGATCGCGTGCTACACAA
TCGA
>NODE_9_length_70_cov_41.885715
ACGTGACGTCATGGAAGAATAAACACACAATAATATAATAATATTGAATTAGATATAGTA
TAATATTGAATTATTTTGAAATGTTAATTTGGGTTAAATTTGTCAATAAAATATTGT
>NODE_16_length_154_cov_11.662337
TGCCTATTACAGGTCTTTCTTTAGCCTGAAATTTTATCTAAATTATTAAGATACATTA
ATTAACATATCCAGACCATTAATGAACCTCTCGATGTCTACACTAGCCTTTGATATTGACT
GATAATTAAATCAAAAAATGGATAATCGAACTATGCTTGTTATTATACTGACTTTATTA
ATTTCTATTCTTAATTCTTGGTt
>NODE_23_length_138_cov_4.391304
GTACCATCCAAAGTACTAGACATTGTTTCTTGTCAGACCCAATTTTACATCACTGATATG
GGGAAAGCAGATTAGAAAATTGGTGAGGACCTTTTCTCTGCCCATAGAGGAAAGGCAACC
GGGACCTTCTGGTAACTGTGCACCCACCCTCCTCTTACTTATTAaacaggcatctac
taagttga
>NODE_32_length_78_cov_4.871795
ACAATTTATCATAGGTCGAGTAGTCATATTAACAAACACAAGTTTTGGTCCAATCTT
TCCGAAATTCGAACTCATCCTTCCTTAGGGTGTAACGGTGATATAATCAAACGATTTTT
AAATCGGa
>NODE_44_length_58_cov_20.120689
TCGAGAGATGGTTTCTTAAGTAAGGCTTGATCAAAGCATGCTTGAATGGCTGTGGCACA
AAACCAGAGTTTAAGCTGTATTAATGATGGCTGTCAGGATAGGTAAT
>NODE_45_length_126_cov_8.746032
TTCTGTTAATCACGTTACGTAGTGAGGTATATGATCCTATTTTCCAAAAGAACTCAAT
TATAAGCTGTCTATAATCTACCTACCCACGCTTGCAATTAGACCGAAATAACTTAATC
ATTTTGAATTAATATCGCACGTAACAAGTTCGGCCAGCTAATCTGAAAAACATT
>NODE_46_length_450_cov_12.680000
GTACCACGATCGAACGAAATAACAATATTATAGTGATTAATGCAGAACTGACGTGATGGA
AACTGACTGTATAGATTTTTTTTCCAGTCTTATATCCGTGAGTAATCAAGCAACACTGC
GATAATGATTACCCGAGTCTGAAAGTTATTAGCAAAAAAGTGAAAAATAAATTTAAAAA
AATAAAATAAAACAAAAACAACCCCATACAATCTAAGGTTAATCACTGATGTGGGGTT
CGTAACAATATGCTGTAGCCTATTGCTAGAAGAAAAAAGTTAGATATACGGGCATTA
TCTATTACACAGAGTTAGTATAATGGGCTATTGATGATACGGTAGATAATGGAAAACGAG
AACGAAATTGTGCTTGTCTTATACATCCTAAAGTTCAAACAATATGTCTAAAAGATTGC
AACCATTGTACTTGGGGAATGTCGAAAGAAATGTTTTTCAGATTAGCTGGCCGAACCTG
TTACGTGCGATATTAATTC
```

Figure 1. Sample contents of the contigs.faa file.

1.1.2. BLAST Algorithm

In very basic terms, the BLAST algorithm breaks down a query sequence into a set of 'words', which by default consists of three letters at the time. The sequences have no defined starting or finishing points and can be read both forwards and backwards.

Thus, a sequence like **ATTACC..** can be broken down into six possible combinations:

- **[ATT][ACC]**
- **A[TTA]CC**
- **AT[TAC]C**
- **[CCA][TTA]**
- **C[CAT]TA**
- **CC[ATT]A**

BLAST then compares those words against the target sequence. If it finds a matching hit, it will search in both direction for further possible matching neighbours (which, also by default, is three letters at a time). BLAST will score the results in a way where consecutive strongly matching sequences gain more points and carry on expanding, while sequences that begin developing too many gaps and mismatches are heavily penalized. Therefore, a strong match can maintain some level of instability that a weak match could not.

Due to both its localised and highly heuristic nature, BLAST is slightly less accurate than other, more optimal methods such as the Smith–Waterman algorithm but makes up for it through sheer performance when handling large amounts of data. [6]

1.1.3. BLAST Tools

BLAST software comes with a variety of tools, the most notable of which include:

Makeblastdb, as the name implies, is responsible for creating an indexed BLAST database based on a FASTA file provided.

\$ makeblastdb -in database_source.fsa -parse_seqids -dbtype nucl

In this example, the arguments passed are the location of the FASTA file, a flag specifying that the sequence identifiers should be treated as unique keys, and the type of the database to be created.

The core BLAST tools are broken down into **blastn** for nucleotide sequences, **blastp** for protein sequences, and **tblastn/blastx/tblastx** for running searches involving nucleotide-protein translations (see: Codons in **section 1.1.1**).

\$ blastn -query query.faa -db database.faa -out blast_results.txt -outfmt 0

While BLAST offers a wide plethora of command-line options [7], a basic search as the one above is often all that is necessary for reliable results. Of note, is the **-outfmt** parameter which specifies a specific format the results should be displayed in.

1.1.4. BLAST Outputs

The official BLAST documentation currently states there are 11 different formats for outputting BLAST data. [7] This information appears to be slightly outdated as the new revised XML and JSON formats in the later version of BLAST bring that count up to 15. [8] The two most commonly used ones are 0, standing for pairwise, and 6 standing for tabular.

```
>Butyrivibrio_proteoclasticus_P6B7__gnl|X|DKKJMKFL_47
Length=1698

Score = 248 bits (134), Expect = 4e-64
Identities = 211/248 (85%), Gaps = 5/248 (2%)
Strand=Plus/Plus

Query 1      GGCCCCCGTCAATTCCTTTGAGTTTCAACCTTGCGGTCGTACTCCCCAGGCGGAGTGCTT 60
      || |||||
Sbjct 639     GGTCCCCCGTCAATTCCTTTGAGTTTCAATTCCTTGCGAACGTACTCCCCAGGTGGAATACTT 698

Query 61     AATGCGTTAGCTGCGGCACTGAAGGGC--GGAAACCTCCAACACCTAGCACTCATCGTT 118
      | |||||
Sbjct 699     ATTGCGTTAGCGACGGCACTGAGCCCTATGGAA--C-CCAACACCTAGTATTTCATCGTT 755

Query 119    TACGGCATGGACTACCAGGGTATCTAATCCTGTTCGCTACCCATGCTTTCGAGCCTCAGC 178
      ||| || |||||
Sbjct 756     TACCGCGTGGACTACCAGGGTATCTAATCCTGTTCGCTCCCCACGCCTTCGAGCCTCAAC 815

Query 179    GTCAGTTGCAGACCAGAGAGCCGCCCTCGCCACTGGTGTTCCTCCTAATATCTACGCATT 238
      |||||
Sbjct 816     GTCAGTTGCTGTCCAGTAAGCCGCCCTCGCCACCGATGTTCTCCTAATATCTACGCATT 875

Query 239    TCACCGCT 246
      |||||
Sbjct 876     TCACCGCT 883

>Butyrivibrio_sp._YAB3001__gnl|X|OEKOEGL_56
Length=1269

Score = 248 bits (134), Expect = 4e-64
Identities = 207/243 (85%), Gaps = 1/243 (0%)
Strand=Plus/Minus

Query 4      CCCCCTCAATTCCTTTGAGTTTCAACCTTGCGGTCGTACTCCCCAGGCGGAGTGCTTAAT 63
      |||||
Sbjct 925     CCCCCTCAATTCCTTTGAGTTTCAATTCCTTGCGAACGTACTCCCCAGGTGGAATACTTATT 866

Query 64     GCGTTAGCTGCGGCACTGAAGGGCGGAAACCTCCAACACCTAGCACTCATCGTTTACGG 123
      |||||
Sbjct 865     GCGTTTGCCTCGGCACTGAAAACCTTTGGTCC-CCAACACCTAGTATTTCATCGTTTACCG 807

Query 124    CATGGACTACCAGGGTATCTAATCCTGTTCGCTACCCATGCTTTCGAGCCTCAGCGTCAG 183
      | |||||
Sbjct 806     CGTGGACTACCAGGGTATCTAATCCTGTTCGCTCCCCACGCCTTCGAGCCTCAACGTCAG 747

Query 184    TTGCAGACCAGAGAGCCGCCCTCGCCACTGGTGTTCCTCCTAATATCTACGCATTTACCC 243
      |||| |
Sbjct 746     TTGCTGTCCAGTAAGCCGCCCTCGCCACCGATGTTCTCCTAATATCTACGCATTTACCC 687

Query 244    GCT 246
      |||
Sbjct 686     GCT 684
```

Figure 2. A typical BLAST pairwise output.

The pairwise output provides some basic statistics and a quick visual representation of the match in question. Key features include:

- Left out from the top and bottom of the figure are the name and the length of the query, an upfront list of significant alignments with their bit score and expect values, and further H/K/Lambda score statistics. These only appear once per query.
- The name of the target sequence and its total length.
- Expect value – a statistical indicator of how likely it is for the match to have happened by chance. Lower value indicates a better match.
- Score/bit score – Similar to the expect value, the bit score is another statistical indicator of the quality of the match. Higher bit scores indicate a better match. While the expect value is database dependant, bit scores are normalised and can be compared with one another regardless of the source of the match.
- Identities – the number of positive matches across a sequence
- Gaps – the number of gaps introduced between the two matching sequences.
- Strand – the direction of the sequence. Plus indicates that the sequence has been matched as is, minus indicates that the sequence has been matched backwards.
- The graphical representation of the match. The top and bottom lines represent the query sequence and subject sequence, with the middle line representing a quick overview of where the gaps and mismatches are. The sequence is broken down into sixty-character chunks for readability, and the numbers on the left and right represent where the particular chunk begins and ends with respect to the query and subject sequences. Gaps are represented by a hyphen in the query/subject string and can be prominently seen in the first example.

Due to the relative size and verbosity of the pairwise output, the compact tabular format is more commonly used.

NODE_78_length_5335_cov_124.731583	Clostridiales_bacterium_R-7_gnl X HMLLHLOM_15	100.000	5376	0	0	1	5376	5376	1	0.0	9928	
NODE_78_length_5335_cov_124.731583	Clostridiales_bacterium_R-7_gnl X HMLLHLOM_15	100.000	104	0	0	5282	5385	5481	5378	5.03e-46	193	
NODE_11442_length_51_cov_119.196075	Clostridiales_bacterium_R-7_gnl X HMLLHLOM_15	100.000	101	0	0	1	101	5427	5327	3.28e-46	187	
NODE_11442_length_51_cov_119.196075	Clostridiales_bacterium_R-7_gnl X HMLLHLOM_15	100.000	41	0	0	1	41	41	1	7.41e-13	76.8	
NODE_42813_length_2486_cov_9.936042	Clostridium_algidicarnis_B3_gnl X PFMGJLBR_1	94.286	35	2	0	1807	1841	306948	306914	1.16e-04	54.7	
NODE_50641_length_51_cov_5.058824	Clostridiales_bacterium_R-7_gnl X HMLLHLOM_15	99.010	101	1	0	1	101	5327	5427	1.53e-44	182	
NODE_50641_length_51_cov_5.058824	Clostridiales_bacterium_R-7_gnl X HMLLHLOM_15	100.000	41	0	0	61	101	1	41	7.41e-13	76.8	
NODE_53627_length_4080_cov_11.740932	Basfia_succiniciproducens_RPR-2_gnl X IEPCCBHJ_2	100.000	29	0	0	4038	4066	15414	15442	1.90e-04	54.7	
NODE_53627_length_4080_cov_11.740932	Basfia_succiniciproducens_DSM_scaffold00003	100.000	29	0	0	4038	4066	15930	15958	1.90e-04	54.7	
NODE_62099_length_659_cov_7.933232	Selenomonas_ruminantium_AC2024_gnl X IEPCCBHJ_1	95.346	709	33	0	1	709	1486864	1486156	0.0	1127	
NODE_62099_length_659_cov_7.933232	Selenomonas_ruminantium_AC2024_gnl X IEPCCBHJ_1	95.063	709	35	0	1	709	723910	723202	0.0	1116	
NODE_62099_length_659_cov_7.933232	Selenomonas_ruminantium_AC2024_gnl X IEPCCBHJ_1	94.640	709	38	0	1	709	40838	41546	0.0	1099	
NODE_62099_length_659_cov_7.933232	Selenomonas_ruminantium_AC2024_gnl X IEPCCBHJ_1	94.640	709	38	0	1	709	453878	454586	0.0	1099	
NODE_62099_length_659_cov_7.933232	Selenomonas_ruminantium_AC2024_gnl X IEPCCBHJ_1	94.217	709	41	0	1	709	1370014	1369306	0.0	1083	
NODE_62099_length_659_cov_7.933232	Selenomonas_ruminantium_AC2024_gnl X IEPCCBHJ_1	94.076	709	42	0	1	709	919283	918575	0.0	1077	

Figure 3. A typical BLAST tabular output.

The information listed is as follows:

- The name of the query.
- The name of the subject sequence matched.
- The percentage of identical matches.
- The length of the alignment.
- The number of mismatches.
- The number of gap openings
- Start of the query alignment.
- End of the query alignment.
- Start of the subject alignment.
- End of the subject alignment.
- Expect value.
- Bit score.

1.2. Analysis

1.2.1. Databases and Results

The particular databases used here were 'contigs.fa', which contained 'contiguous' DNA fragments attempting to replicate the contents of the gut of a blue-rayed limpet, matched against 'hungate.fa' – a 408 genome high quality 'gold-standard' database of organisms known to be found in rumen (cow's stomach). Due to the sheer amount of the data involved, small extracts from the 'contigs.fa' database were later used with the NCBI online BLAST implementation [9] to provide a faster, more practical dataset.

The two FASTA files were approximately 460 megabytes and 1.3 gigabyte respectively, with 730,724 sequences in 'contigs.fa' and 18,323 sequences in 'hungate.fa'. When matched, they provided 12,172 local hits/HSPs across 7582 subject sequences. This information will later come useful when assessing and extrapolating performance.

As seen, the BLAST results provided are fairly rudimentary. The pairwise output is far more readable but takes up vastly more space due to both the overall layout and graphical representation of the sequence, as well as the obligatory inclusion of every empty non-matching query searched.

The 'contigs_0.txt' results file takes up over 185 megabytes of space, with the bulk of it containing only references to queries that received no matches, as compared to the 'contigs_6.txt' results file which takes up 1.5 megabytes of space and contains only the relevant matches. The tabular output is much more efficient to an experienced bioinformatician, but the information is even more abstract than the already sparse pairwise output.

1.2.2. Problem Domain

This project was somewhat unique in that it takes place within a particular, relatively fragmented and niche area of software engineering. In a way, the established technical aspects of the project are far more straightforward relative to the research required to get to that point. In fact, given the process at hand – to a certain extent, it could be argued that the research and analysis aspects of this project never stopped.

For someone with no prior experience with the field, a non-insignificant amount of time had to be devoted to just developing the familiarity with the BLAST software and the bioinformatic concepts involved, before even attempting to branch out into further research.

Further research was then hampered by the fact that, for better or worse, BLAST has become accepted as a core foundation of modern bioinformatics and as such, it is difficult to find many references or discussions regarding BLAST tools in terms of software engineering and usability, as opposed to 'generic' academic papers/tutorials/etc. Worse yet, 'blast' is a ubiquitous term with multiple meanings, which does it no favours when dealing with search engines or open-source repositories.

While it is not uncommon to hear some informal grumbling when discussing the topic with biology students or postgraduates, the vast majority of them lack the appropriate technical knowledge, language, and experience to properly express their gripes into constructive criticism. Likewise, it is also not uncommon to find online criticisms of science/bioinformatics software as a whole [2] [10] [11], yet those complaints prove far too broad to be of much use.

To date, Peter Cock's online blog "Blasted Bioinformatics!?" [12] is just about the only easily found source of detailed, software-oriented criticisms of BLAST. Peter Cock is one of the notable developers of the Python Biopython library [13] and, while an interesting read, it is largely too thorough and detailed for the purposes of this project.

In the end, in spite of the research done, the bulk of requirements and other suggestion have been provided primarily by local experienced bioinformaticians - namely my supervisor, with some help from one of the bioinformatics-focused research students.

1.2.3. Similar Existing Software

There exists a variety of BLAST-related tools, both of various complexity and in various states of progress. Galaxy [14] is perhaps the most sprawling and comprehensive of bioinformatics software suites, though the BLAST features themselves do not seem to come with much expanded functionality beyond the ability to merely support BLAST as part of its general workflow web platform.

The next three seemingly most notable BLAST tools appear to be BlastViewer [15], JAMBLAST [16], and BLASTGrabber [17]. Out of the three, the first two appear to be largely amateur hobby projects, with BLASTGrabber holding the most potential as a published university-developed tool most closely resembling what this project is aiming for.

Nevertheless, all of the above tools are heavily GUI-based and primarily focused on improving the overall user experience. While usability is also of concern in this project, the bulk of bioinformatic work out there is still primarily carried out remotely on HPC (High-Performance Computing) clusters through the use of command line input, and this project intends to primarily focus on terminal-based interaction with improved graphical visualisation serving as a lesser/secondary priority task.

In terms of the more obscure open-source GitHub repositories, the overwhelming majority of BLAST-related software does not seem to extend much past basic BLAST interface wrappers for other languages and frameworks.

1.2.4. Initial Proposed Features

As previously mentioned in **section 1.2.2**, attempting to analyse and come up with features for a bioinformatics tool without previous extensive experience with bioinformatics is a difficult task, and as such most of the suggestions have been provided by my supervisor.

Thus, an initial feature set has been established, with further scope and functionality being introduced as the original requirements are fulfilled:

Counting individual gaps.

Referring back to the BLAST tabular output in **section 1.1.4**, the sixth of the columns provided is 'gaps open'. An example of gaps in a sequence can be seen in *Figure 4*. below.

Sequence gaps allow for short one-sided mismatches if an otherwise strongly matching pair of sequences has a few too many additional nucleotides/amino acids in one of the sequences. As minor gene mutations are relatively common, it is typically more convenient to consider a whole longer match with some gaps or mismatches than to break said match into smaller 'perfect' matches.

```
Query 61 AATGCGTTAGCTGCGGCACTGAAGGGC--GGAAACCCTCCAACACCTAGCACTCATCGTT 118
      | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
Sbjct 699 ATTGCGTTAGCGACGGCACTGAGCCCTATGGAA--C-CCAACACCTAGTATTTCATCGTT 755
```

Figure 4. An example of a matching BLAST sequence with gaps and mismatches.

A large issue with the BLAST tabular output is that 'gaps open' only refers to the presence of gaps, regardless of their size or location. *Figure 4*. shows two gaps in the query sequence, followed by two gaps in the subject sequence, followed by another gap in the subject sequence, for a total of five individual gaps. If counting by open gaps alone, this same match would only have three gaps.

The pairwise output is slightly better in this regard as it will show the number of individual gaps alongside an actual output of the sequence in question (as seen above), but nevertheless there does not exist a compact way to quickly ascertain the number of gaps open, their sizes, and location, in a single output.

Determining the total query coverage.

There is a famous adage stating that humans and chimpanzees share 99% of their respective genome. This same adage also serves to indicate a prevalent issue when comparing sequences using BLAST – organisms, on the whole, share a lot of common DNA with one another. Due to the algorithm's heuristic nature and focus on finding similar localised sequences, highly accurate matches may often be misleading as only a very small subset of the query is being matched. The diagram below better illustrates the problem.

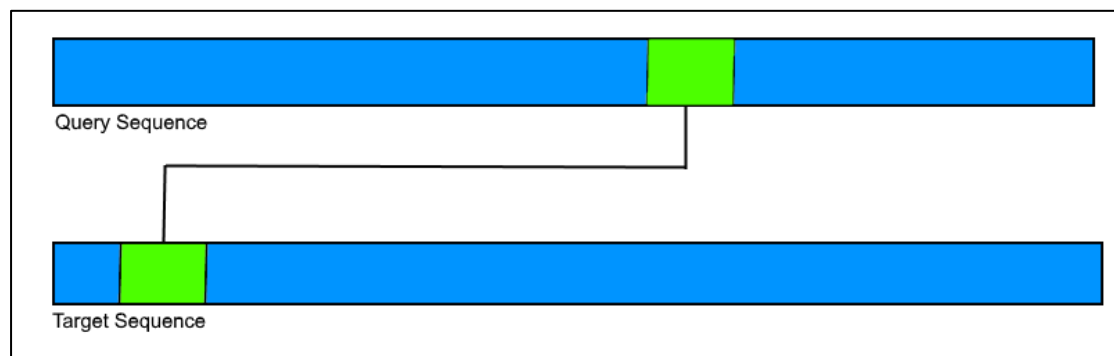


Figure 5. An illustration of what a 'misleading' match may look like. The green represents a 100% match.

In this hypothetical instance, a significant perfect match between the subject and the query could have potentially made for a very interesting result. A perfect match between a tenth of two sequences is significantly less interesting as it may just be an incredibly common and mundane set of proteins. By providing a quick way to establish of much of a particular query has actually been matched, one can make a better-informed judgement regarding the significance of a match.

Indexing the BLAST results and providing statistics on the most and least common matches.

As the amount of bioinformatics data and processing power have both increased exponentially, so has the ability to carry out larger BLAST searches. As previously mentioned in **section 1.2.1**, the two sequence sets provided were used to search over 700,000 queries against 18,000 subjects for a total of 12,000 results. This may sound like a lot, but it is not an uncommon amount - if anything, those two sets are relatively small compared to most of the common BLAST databases.

Default BLAST outputs are only capable of listing results by the order in which the queries have been searched, followed by their respective matches listed in the order of how accurate they were.

1.3. Process

Given the problem domain and the design/requirements flexibility involved, a modified variant of Extreme Programming has been chosen as the most flexible solution. Based primarily on the WikiWikiWeb's "Extreme Programming For One" article and related sub-links [18], this adapted process primarily emphasizes uncertainty and the ability to rapidly respond to change.

The lack of strongly defined concrete requirements or long-term familiarity with the problem domain naturally emphasises a significant amount of iterative design with a heavy focus on prototype spike work, modularised components that can be largely integrated or decoupled at will, heavy amount of refactoring to arrive at a 'simple' design, and finally a consistent use of testing and continuous integration.

2. Design

2.1. Overall Architecture

The overall high-level architecture is a relatively straightforward data pipeline design.

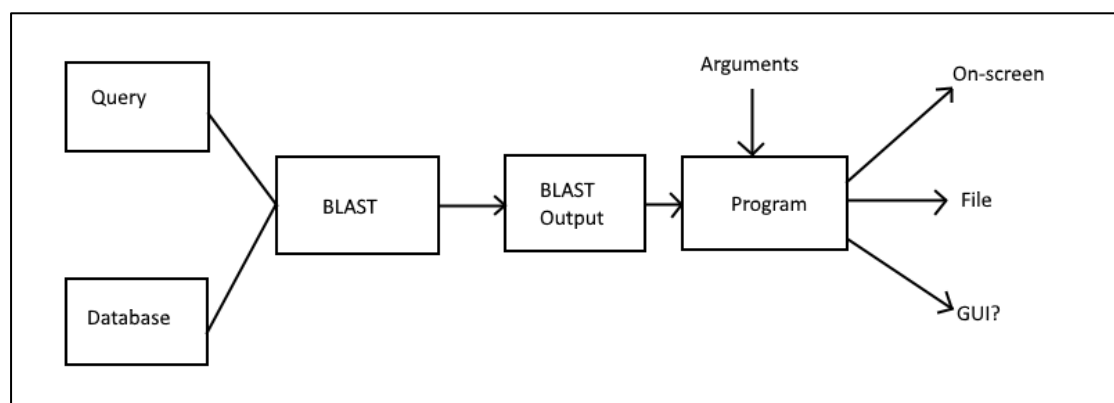


Figure 6. A high-level overview of the architecture.

The two FASTA files are used to perform a BLAST search, the results of which are fed into the program, augmented as specified by the command line arguments, and returned in the form of either terminal output, a text file, or a graphical display.

Considering the general fragmentation of the bioinformatics software ecosystem, some of the biggest design decisions revolved around handling of the BLAST input in the first place. While the overall design philosophy for the program itself was to keep the design simple and easily adaptable, having to first decide on a programming language as well as the appropriate BLAST output was a difficult choice as each of the options had its own advantages and downsides. The program was designed in a way that would potentially allow expanded input functionality with only slight modifications to the existing code, but in practice, picking one over the other would mean staying committed to that particular approach for the duration of the project.

2.2. Language Choice

As bioinformatics is a scientific field which involves a significant number of scientists with no previous programming backgrounds, there is a great deal of quality discussion regarding the most appropriate choice of a programming language for the tasks involved. [19] [20] [21]

Some of the common choices include:

- R – R is a common favourite of scientists, statisticians and mathematicians worldwide due to its strengths with regards to developing statistical software and data analysis. It currently ranks 12th on the TIOBE popularity index [22] and offers some basic Bioinformatics resources at hand. [23]
- Python – first developed in 1991, Python's design philosophy which emphasizes easy of use and readability, has made it yet another common favourite [22], particularly across various scientific communities. Python is arguably the most common language used in Bioinformatics and one with largest third-party support.
- C/C++ - While hugely popular back in the old days of bioinformatics and still the best choice when one wants to create a robust platform with high performance, C's popularity has somewhat waned as processing power become more abundant and simpler languages have become much more common. Relatively poor in terms of third-party support.
- Java – An overall middle ground between various common bioinformatics programming languages in just about every way, including performance, readability and third-party support.
- Perl – Another old favourite. Perl carries a lot of clout due to its former legacy and ubiquity as the 'go to' scripting language back in an era where C and Perl were just about the only sensible choices at the time. Perl was notably used as part of the Human Genome Project. [24] Yet, as far as modern bioinformatics is concerned, Perl is rapidly losing ground to Python as the leading scripting language.

In the end, the choice largely came down to Python and Java. Python boasted ease-of-use and a significant amount of third-party bioinformatics support, while Java came vastly ahead in terms of performance. It should also be noted that the developer in question was significantly more experienced with Java, compared to his little relative experience with Python.

Eventually, Python was chosen for its wide acceptance and support in the bioinformatics and general science communities, as well as the developer's own desire to branch out and expand their horizons. Python is also capable of integrating C and R modules, should performance ever become a bottleneck.

2.3. BLAST Output Formats

There are a lot of jokes regarding the file format ecosystem in bioinformatics. One such example is this seven-year-old 'Biostars' (A bioinformatics-focused StackOverflow-style forum) thread titled "*Question: What Are The Most Common Stupid Mistakes In Bioinformatics?*" [25] where the most upvoted answer is "Invent a new weakly defined, internally redundant, ambiguous, bulky fruit salad of a data format. Again."

While humorous, this observation is also somewhat accurate. BLAST alone boasts a total of 15 different outputs [8], including three different variations of the XML and JSON outputs – original JSON/XML, and the single/split file variations of the newly improved 'version 2.' formats. That is on top the standard pairwise output, tabular, CSV, SAM, and BLAST's own internal ASN.1 variant.

The initial idea was to parse in tabular output, therefore immediately making the program useful to the Aberystwyth IBERS department due to the sheer amount of existing tabular data on the servers. While promising, this design concept was dismissed rather soon on account of practical considerations. As the tabular output lacks a lot of the crucial information needed, further work would have had to be done accessing the original FASTA files in order to extract the exact sequences based on their lengths and start/end points. The locations of gaps and mismatches would then have to be inferred from the original sequences, which requires the exact copies of the FASTA files used. Not only is that approach cumbersome and very computationally intensive in terms of I/O, the algorithm itself would be very challenging and error-prone as at that point one is, de facto, recreating a less robust implementation of the BLAST algorithm itself.

The JSON format did not fare much better. While it was capable of providing the necessary information about the gaps and mismatches, as calculated and inferred through the “starts” and “lens” sections, the overall layout was very obtuse and hard to work with.

The JSON2 format is a vast improvement, and in hindsight would have been heavily considered as choice of input for the project. Regrettably, due to the sheer number of formats available and the confusion between what JSON & JSON2 referred to, it simply was not noticed until the project was already well under way – which serves all the more to highlight the issue of BLAST having too many formats.

```

{
  "num": 1,
  "description": [
    {
      "id": "gi|1353793167|gb|CP027077.1|",
      "accession": "CP027077",
      "title": "Bos mutus isolate yakQH1 chromo",
      "taxid": 72004,
      "sciname": "Bos mutus"
    }
  ],
  "len": 103782134,
  "hsp": [
    {
      "num": 1,
      "bit_score": 51.7806,
      "score": 56,
      "evalue": 0.0088711,
      "identity": 35,
      "query_from": 61,
      "query_to": 97,
      "query_strand": "Plus",
      "hit_from": 93728900,
      "hit_to": 93728937,
      "hit_strand": "Plus",
      "align_len": 38,
      "gaps": 1,
      "qseq": "TTATCTCGAAGGCAGTTTATTTAATA-AAATA",
      "hseq": "TTATCTCGAAGTCAGTTTATTTAATAATA",
      "midline": "||||||| ||||| ||||| ||"
    }
  ]
}

```

Figure 7. JSON2 output example.

JSON output example:

```

"segs": {
  "denseg": {
    "dim": 2,
    "numseg": 7,
    "ids": [
      {
        "local": {
          "str":
"Query_121825"
        }
      },
      {
        "gi": 688521244
      }
    ],
    "starts": [
      101,
      8440,
      -1,
      8462,
      89,
      8466,
      87,
      -1,
      71,
      8478,
      69,
      -1,
      60,
      8494
    ],
    "lens": [
      22,
      4,
      12,
      2,
      16,
      2,
      9
    ],
    "strands": [
      "minus",
      "plus",
      "minus",
      "plus",
      "minus",
      "plus",
      "minus",
      "plus",
      "minus",
      "plus"
    ]
  }
}

```

```

<BlastOutput_query-ID>Query_121825</BlastOutput_query-ID>
<BlastOutput_query-def>NODE_1_length_180_cov_7.511111</BlastOutput_query-d
<BlastOutput_query-len>230</BlastOutput_query-len>
<BlastOutput_param>
  <Parameters>
    <Parameters_expect>10</Parameters_expect>
    <Parameters_sc-match>2</Parameters_sc-match>
    <Parameters_sc-mismatch>-3</Parameters_sc-mismatch>
    <Parameters_gap-open>5</Parameters_gap-open>
    <Parameters_gap-extend>2</Parameters_gap-extend>
    <Parameters_filter>L;m;</Parameters_filter>
  </Parameters>
</BlastOutput_param>
<BlastOutput_iterations>
<Iteration>
  <Iteration_iter-num>1</Iteration_iter-num>
  <Iteration_query-ID>Query_121825</Iteration_query-ID>
  <Iteration_query-def>NODE_1_length_180_cov_7.511111</Iteration_query-def>
  <Iteration_query-len>230</Iteration_query-len>
<Iteration_hits>
<Hit>
  <Hit_num>1</Hit_num>
  <Hit_id>gi|1353793167|gb|CP027077.1|</Hit_id>
  <Hit_def>Bos mutus isolate yakQH1 chromosome 9</Hit_def>
  <Hit_accession>CP027077</Hit_accession>
  <Hit_len>103782134</Hit_len>
  <Hit_hsps>
    <Hsp>
      <Hsp_num>1</Hsp_num>
      <Hsp_bit-score>51.7806</Hsp_bit-score>
      <Hsp_score>56</Hsp_score>
      <Hsp_evalue>0.0088711</Hsp_evalue>
      <Hsp_query-from>61</Hsp_query-from>
      <Hsp_query-to>97</Hsp_query-to>
      <Hsp_hit-from>93728900</Hsp_hit-from>
      <Hsp_hit-to>93728937</Hsp_hit-to>
      <Hsp_query-frame>1</Hsp_query-frame>
      <Hsp_hit-frame>1</Hsp_hit-frame>
      <Hsp_identity>35</Hsp_identity>
      <Hsp_positive>35</Hsp_positive>
      <Hsp_gaps>1</Hsp_gaps>
      <Hsp_align-len>38</Hsp_align-len>
      <Hsp_qseq>TTATCTCGAAGGCAGTTTATTTAATA-AAATATTGTGG</Hsp_qseq>
      <Hsp_hseq>TTATCTCGAAGTCAGTTTTTTAATATAAATATTGTGG</Hsp_hseq>
      <Hsp_midline>||||||| ||||| ||||| |||||</Hsp_midline>
    </Hsp>
  </Hit_hsps>
</Hit>

```

Figure 8. BLAST XML output example.

```

<query-id>Query_121825</query-id>
<query-title>NODE_1_length_180_cov_7.511111</query-title>
<query-len>230</query-len>
<hits>
  <Hit>
    <num>1</num>
    <description>
      <HitDescr>
        <id>gi|1353793167|gb|CP027077.1|</id>
        <accession>CP027077</accession>
        <title>Bos mutus isolate yakQH1 chromosome 9</title>
        <taxid>72004</taxid>
        <sciname>Bos mutus</sciname>
      </HitDescr>
    </description>
    <len>103782134</len>
    <hsps>
      <Hsp>
        <num>1</num>
        <bit-score>51.7806</bit-score>
        <score>56</score>
        <evaluate>0.0088711</evaluate>
        <identity>35</identity>
        <query-from>61</query-from>
        <query-to>97</query-to>
        <query-strand>Plus</query-strand>
        <hit-from>93728900</hit-from>
        <hit-to>93728937</hit-to>
        <hit-strand>Plus</hit-strand>
        <align-len>38</align-len>
        <gaps>1</gaps>
        <qseq>TTATCTCGAAGGCAGTTTATTTAATA-AAATATTGTGG</qseq>
        <hseq>TTATCTCGAAGTCAGTTTTTTTAAATATAAATATTGTGG</hseq>
        <midline>||||||| ||||| ||||| |||||</midline>
      </Hsp>
    </hsps>
  </Hit>

```

Figure 9. BLAST XML2 output example.

The two XML outputs are superficially very similar with only some minor changes between the two. They are highly readable and provide all the necessary information. Unfortunately, they are also overly verbose and quite sizeable. The XML1 output of the previously mentioned 12,000 result BLAST file comes up to around 550 megabytes and is by far the largest of the outputs. The slightly more streamlined XML2 output has not been tested, due to the same issue of being accidentally glossed over as JSON2, but it is unlikely to be significantly smaller.

In the end, XML1 was chosen due to two important designs and maintainability considerations.

As part of the development, extra consideration was given towards the 'do the simplest thing that could possibly work' aspect of the Extreme Programming process. Reading in BLAST results from an XML output was the only format already supported by the BioPython library which saved the extra effort involved in 're-inventing the wheel' - either by writing a new parser from scratch or working with a generic JSON/XML parser.

Even more importantly, according to the BioPython Cookbook:

"As mentioned above, BLAST can generate output in various formats, such as XML, HTML, and plain text. Originally, Biopython had parsers for BLAST plain text and HTML output, as these were the only output formats offered at the time. Unfortunately, the BLAST output in these formats kept changing, each time breaking the Biopython parsers. Our HTML BLAST parser has been removed, but the plain text BLAST parser is still available (see Section 7.5). Use it at your own risk, it may or may not work, depending on which BLAST version you're using.

As keeping up with changes in BLAST became a hopeless endeavor, especially with users running different BLAST versions, we now recommend to parse the output in XML format, which can be generated by recent versions of BLAST. Not only is the XML output more stable than the plain text and HTML output, it is also much easier to parse automatically, making Biopython a whole lot more stable." [26]

In other words, NCBI appears to be constantly tweaking and adjusting the other outputs to the point where even the largest unofficial bioinformatics Python library is unable to keep up with the parser breaking. Much as the XML file size remains a point of concern, it would appear to be by far the most stable solution that can be relied on still working a few years down the line.

2.4. User Interface

As previously mentioned, the general user interface concerns were relatively minor. The program would be primarily operated through a one-time execution using command line arguments, with the interface design primarily focused on sensible parameter names, input sanitization, and composing useful output. At the very least, the application would be expected to accept an XML input file path, and an optional output file path. Additional flags would exist for tweaking the output itself.

Further interface design would primarily revolve around making use of the Jupyter Notebook framework. [27] Jupyter is a set of tools and libraries that allows a developer to run Python and R code embedded within a HTML file. This code can be further edited by the user currently rendering the page and is commonly used for rendering various data visualisations.

As Jupyter Notebook is already heavily favoured by both the general scientific and bioinformatics communities, it would be relatively simple to write a web-based graphical layer that would interact with the program in a pseudo-MVC-pattern manner. While Jupyter Notebook would hardly be an appropriate graphical interface in the vast majority of circumstances, the nature of this particular project makes it possible to make that assumption about the target audience and take it for granted.

2.5. NCBI Taxonomy Database

Select resources from the NCBI taxonomy database were used as a 'gold standard' reference set for comparing BLAST XML results with existing species. [28] [29] The database consists of 13 distinct '.dmp' files, for a total size of 1.3 gigabytes.

The files appear to have been directly 'dumped' from the NCBI's online database (hence the 'taxdump' designation and the '.dmp' extension) and such act as a collection of indexed text files, forming a crude relational database between them.

The more specific details of the taxonomy database will be further covered in **section 3**, but it would be remiss not to briefly mention it in the context of **section 2.6**.

2.6. Indexing Most/Least Common Results

Alignments		Download	GenBank	Graphics	Distance tree of results
					Desci
<input type="checkbox"/>	Mouse DNA sequence from clone RP24-205E7 on chromosome X, complete sequence				
<input type="checkbox"/>	Mouse DNA sequence from clone RP23-257P3 on chromosome X, complete sequence				
<input type="checkbox"/>	Mouse DNA sequence from clone RP23-45G5 on chromosome X, complete sequence				
<input type="checkbox"/>	Mus musculus chromosome 6, clone RP24-315D19, complete sequence				
<input type="checkbox"/>	Mus musculus 6 BAC RP23-235P6 (Roswell Park Cancer Institute (C57BL/6J Female) Mouse BAC Library) complete sequence				
<input type="checkbox"/>	Mus musculus chromosome 6, clone 454_F_16, complete sequence				
<input type="checkbox"/>	Mus musculus chromosome 8, clone RP23-314N20, complete sequence				
<input type="checkbox"/>	Mus musculus BAC clone RP24-545J19 from chromosome 17, complete sequence				
<input type="checkbox"/>	Mus musculus chromosome 8, clone RP24-273M3, complete sequence				
<input type="checkbox"/>	Mus musculus BAC clone RP24-399I16 from chromosome 17, complete sequence				
<input type="checkbox"/>	Mus musculus chromosome 6, clone RP23-177J17, complete sequence				
<input type="checkbox"/>	Mus musculus BAC clone RP23-356C20 from chromosome 17, complete sequence				
<input type="checkbox"/>	Mus musculus BAC clone RP23-101K11 from chromosome 17, complete sequence				
<input type="checkbox"/>	PREDICTED: Mus musculus predicted gene, 41538 (Gm41538), transcript variant X2, ncRNA				
<input type="checkbox"/>	PREDICTED: Mus musculus predicted gene, 41538 (Gm41538), transcript variant X1, ncRNA				
<input type="checkbox"/>	Mus musculus targeted KO-first, conditional ready, lacZ-tagged mutant allele Tmsb15a.tm1a(EUCOMM)Wtsi; transgenic				
<input type="checkbox"/>	Mus musculus targeted non-conditional, lacZ-tagged mutant allele Tmsb15a.tm1e(EUCOMM)Wtsi; transgenic				
<input type="checkbox"/>	Mus musculus chromosome 17, clone RP24-238N14, complete sequence				
<input type="checkbox"/>	Mouse DNA sequence from clone RP23-250F8 on chromosome X, complete sequence				

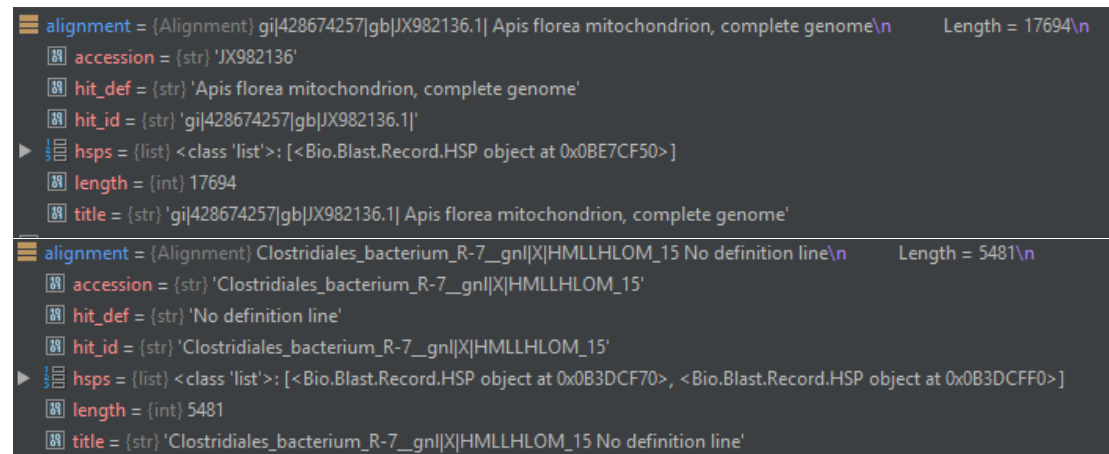
Figure 10. Results taken from analysing a piece of mouse DNA using the online NCBI BLAST tool.

This task was initially assumed to be relatively trivial. Instead, it soon became apparent it would go on to become the most complex and involved out of the three initial requirements. *Figure 10* shows the search results based on the default 'Nucleotide collection (nr/nt)' database.

As clearly seen, the same sets of results can end up quite varied and not very consistent. The human user can clearly extract the relevant aspects of the information as well as determine that 'Mouse' and 'Mus musculus' are one and the

same but attempting to reliably determine that through an algorithmic approach is a far more challenging task. The vast majority of the information shown, while useful to a bioinformatician, only acts as redundant noise when trying to cluster the results.

Worse yet, while various sequence databases do generally make the effort to keep the information organised in a consistent format (see: **section 1.1.1**) using a combination of ids, definitions and accessions, these are neither consistent with one another nor necessarily always required to be adhered to.



```

alignment = {Alignment} gij428674257|gb|JX982136.1| Apis florea mitochondrion, complete genome\n Length = 17694\n
accession = {str} 'JX982136'
hit_def = {str} 'Apis florea mitochondrion, complete genome'
hit_id = {str} 'gij428674257|gb|JX982136.1'
hsp = {list} <class 'list'>: [<Bio.Blast.Record.HSP object at 0x0BE7CF50>]
length = {int} 17694
title = {str} 'gij428674257|gb|JX982136.1| Apis florea mitochondrion, complete genome'

alignment = {Alignment} Clostridiales_bacterium_R-7_gnl|X|HMLLHLOM_15 No definition line\n Length = 5481\n
accession = {str} 'Clostridiales_bacterium_R-7_gnl|X|HMLLHLOM_15'
hit_def = {str} 'No definition line'
hit_id = {str} 'Clostridiales_bacterium_R-7_gnl|X|HMLLHLOM_15'
hsp = {list} <class 'list'>: [<Bio.Blast.Record.HSP object at 0x0B3DCF70>, <Bio.Blast.Record.HSP object at 0x0B3DCF70>]
length = {int} 5481
title = {str} 'Clostridiales_bacterium_R-7_gnl|X|HMLLHLOM_15 No definition line'

```

Figure 11. A comparison between the NCBI web-based BLAST results set, and results from the local 'contigs-hungate' search.

A more sophisticated method for determining relative string similarity was required.

2.6.1. Edit Distance Based Methods

One of the most common ways of matching approximate strings is through the use of edit distance. Levenshtein distance [30] is a common algorithm that counts the number of insertions, deletions and substitutions required to turn one string into another.

For example, 'cat' and 'hat' have a Levenshtein distance of 1, while 'bioinformatics' and 'biology' have a Levenshtein distance of 10.

Levenshtein and other related algorithms from that family were considered, however while edit distance algorithms excel when it comes to basic tasks such as spell checking individual words, they are largely at a disadvantage and impractical when it comes to dealing with long strings or strings of differing lengths.

Referring back to *Figure 10*, the results provided are far too varied and unreliable to provide a useful metric – as basic prototype demonstrations involving Levenshtein distance quickly proved.

There might be some merit in breaking down the strings into smaller subsets, then iteratively checking the distance against one another, though that might only be useful in the case of clustering similar local results against one another. In any case, this approach would still be unable to determine that 'Mouse' and 'Mus musculus' are of the same species and would be highly inefficient if combined with the taxonomy database approach used in the final product.

2.6.2. N-Gram Based Approaches

As briefly discussed above, n-grams are a method of breaking down a string into individual 'grams' for the purposes of easier semantic analysis. The N in the name stands for the number of consecutive chunks a particular string is broken down into. These can be individual words, letters or syllables.

For a basic example, in the case of "the quick brown fox jumps over a lazy dog", the results would be as such:

N=1: Unigram	N=2: Bigram	N=3: Trigram
"the"	"the quick"	"the quick brown"
"quick"	"quick brown"	"quick brown fox"
"brown"	"brown fox"	"brown fox jumps"
"fox"	"jumps over"	"fox jumps over"
"jumps"	"over lazy"	"jumps over lazy"

N-grams are a popular approach due to their customisability, ability to infer a degree of context with regards to word order, and the well-understood space/performance trade-offs involved.

Breaking down a title such as "Clostridiales_bacterium_R-7__gnl|X|HMLLHLOM_15" into ["Clostridiales bacterium", "bacterium R-7", "R-7 gnl|X|HMLLHLOM", "gnl|X|HMLLHLOM 15"] bigrams offers the obvious benefit of having at least one algorithmically legible word to make a comparison against.

The n-grams could then be compared against each other using a set-based method such as the Jaccard index.

A method for using unigrams and bigrams in order to compare the BLAST results against the taxonomy database was briefly prototyped, however it was soon dismissed due to unacceptable performance. Even ignoring further complexity introduced by individual n-gram comparisons, the program would have to iteratively compare one of 12,000 given results against over 1.8 million results in the database.

2.6.3. Bag-Of-Words Machine Learning

The bag-of-words approach is a popular natural language processing model, also known as the vector space model.

Bag-of-words collects all the individual words in a string (though again, letter and syllable n-grams are also acceptable) to form a matrix of extracted features that can be used for comparison.

For a very basic example, let us compare the previously used "the quick brown fox jumps over a lazy dog" string alongside two more: "I like the brown fox" and "I like a lazy dog".

A bag-of-words approach would first gather all the words across the three strings and count their combined occurrences.

Example: [“the”:2, “quick”:1, “brown”:2, “fox”:2, “jumps”:1, “over”:1, “a”:2, “lazy”:2, “dog”:2, “I”:2 “like”:2]

The above matrix of total words can be more succinctly broken down into [2,1,2,2,1,1,1,2,2,2], which is also referred to as the ‘vector space’.

The respective three matrices for the individual string would thus break down into
 String 1: [1,1,1,1,1,1,1,1,1,0,0],
 String 2: [1,0,1,1,0,0,0,0,0,1,1],
 and String 3: [0,0,0,0,0,0,1,1,1,1,1].

Introducing a fourth string such as “I like **the** lazy dog” would result in a matrix of [1,0,0,0,0,0,0,1,1,1,1] which, when compared against other matrices, would most closely match to “I like a lazy dog”.

It should be noted, that the approach used in the explanation above is one that makes use of “term frequency” which weighs the matrices based on most commonly occurring words. As some words are vastly more prevalent than others, it is reasoned that the more unique words would hold more semantic importance and, therefore, the term frequency–inverse document frequency (usually referred to as tf-idf) statistic is commonly employed to adjust the word weights to favour the less common terms over the more common ones.

Ultimately, it was decided for the program makes use of a bag-of-words model using the cosine similarity metric [31] as a comparison point. It was the only approach with any reasonable degree of performance.

2.7. Final Design

The final design of the program is broken down into three primary classes: **Main**, **TaxMatch** and **TaxTree**.

- The **Main** class is responsible for defining and handling the incoming command line arguments, parsing the BLAST XML results, and outputting them back in a tabular format with additional augmented information regarding gaps and coverage.
- The **TaxMatch** class takes in a list of subject match names and employs machine learning techniques to match the provided results to an NCBI taxonomy database, returning a series of Taxonomy IDs in the process.
- The **TaxTree** class takes in the Taxonomy IDs and cross-references them with the taxonomy database in order to build a taxonomy tree. This class also contains methods that use the aforementioned tree to calculate most/least common result sets, as well as visualise them.

In hindsight, those classes could stand to be further refactored into five or six distinct classes as further future functionality introduces further complexity, but being as they are right now, further breaking up of classes was deemed to introduce more clutter than be of any objective benefit.

Each of the classes currently requires the class preceding it in order to work, but they are otherwise capable of working fully independently. **Main** only requires an XML input, **TaxMatch** only requires a list of names, while **TaxTree** only requires an appropriate list of integer IDs.

2.8. Other

2.8.1. Developer Hardware

The development and testing primarily took place on the following machine:

CPU: Intel i5-3570k, overclocked to 4.0ghz

RAM: 16 Gigabytes, DDR3.

Motherboard: Gigabyte Z77X-3DH

GPU: NVIDIA GeForce GTX 1070

Hard-Drive: 160GB SSD (Windows), 1TB HDD (Linux)

Operating Systems: Windows 10 (latest stable developer preview), Fedora 27 (kept up to date daily)

2.8.2. Developer Environment

The program was primarily developed using IntelliJ PyCharm Community Edition IDE. Python-configured Visual Studio Code and Notepad++ were also heavily used, though primarily while dealing with various text files more so than actual development.

2.8.3. Backup And Version Control

The project was maintained using a Git repository set up on a Dropbox cloud and file syncing platform. While generally ill-advised when dealing with large groups of developers, due to the risk of corrupting the repository, this setup remained reliable when dealing with a single developer using multiple machines and operating systems.

2.8.4. Compatibility

The program is designed to work in Linux distributions running Python 3.6 and above. It is recommended for the program to be run in a Linux environment. Windows compatibility is possible, assuming all the required dependencies have been fulfilled, but not guaranteed - all the usual caveats of working with Python in Windows apply.

3. Implementation

3.1. Biopython

The open-source Biopython [32] library proved a massive benefit when streamlining the application. With nothing more than:

```
from Bio.Blast import NCBIXML  
blastRecords = NCBIXML.parse(filepath)
```

The program was capable of dynamically parsing the XML file using the SAX approach and automatically converting the results into an iterable object. This convenient availability of easily-organised data limited the amount of work required to reformat the same data into the tabular output and the use of a SAX parser alleviated performance concerns when dealing with large XML files.

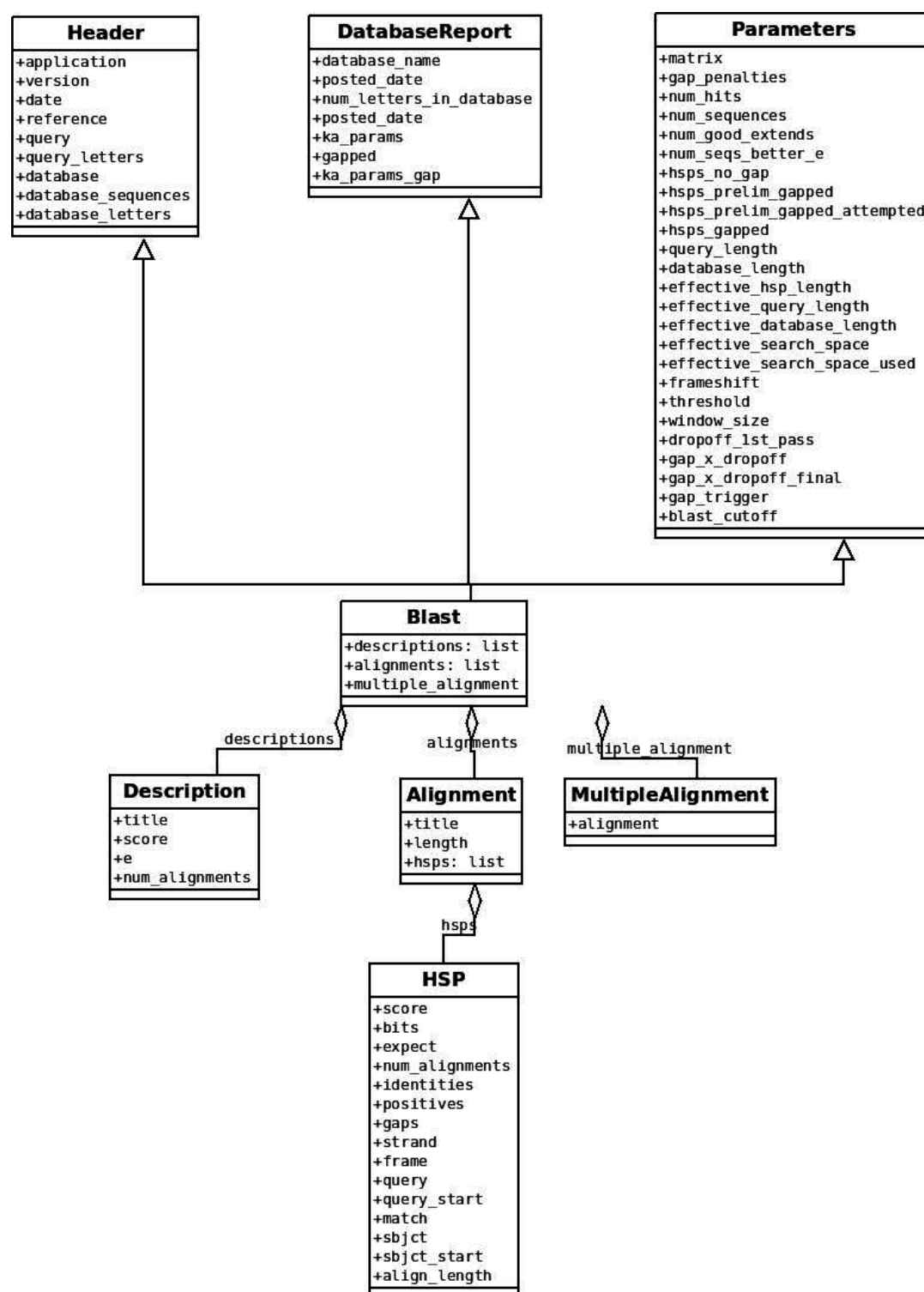


Figure 12. The BLAST Record class diagram, taken from the Biopython Cookbook, section 7.4. [13]

```

blast_record = {Blast} <Bio.Blast.Record.Blast object at 0x0B43CD30>
alignments = {list} <class 'list'>: [<Bio.Blast.Record.Alignment object at 0x0B43CDD0>]
0 = {Alignment} Clostridiales_bacterium_R-7_gnl|X|HMLLHLOM_15 No definition line\n
  accession = {str} 'Clostridiales_bacterium_R-7_gnl|X|HMLLHLOM_15'
  hit_def = {str} 'No definition line'
  hit_id = {str} 'Clostridiales_bacterium_R-7_gnl|X|HMLLHLOM_15'
  hsps = {list} <class 'list'>: [<Bio.Blast.Record.HSP object at 0x0B43CF70>, <Bio.Blast.Record.HSP object at 0x0B43CF70>]
  0 = {HSP} Score 5376 (9928 bits), expectation 0.0e+00, alignment length 5376\nQuery:
    align_length = {int} 5376
    bits = {float} 9928.71
    expect = {float} 0.0
    frame = {tuple} <class 'tuple'>: (1, -1)
    gaps = {int} 0
    identities = {int} 5376
    match = {str} '|||||
    num_alignments = {NoneType} None
    positives = {int} 5376
    query = {str} 'CTCTCACGTTGGCTGACGACCGATTAGAGGCGTTTATGATAATCCCA
    query_end = {int} 5376
    query_start = {int} 1
    sjct = {str} 'CTCTCACGTTGGCTGACGACCGATTAGAGGCGTTTATGATAATCCCA
    sjct_end = {int} 1
    sjct_start = {int} 5376
    score = {float} 5376.0
    strand = {tuple} <class 'tuple'>: (None, None)
    1 = {HSP} Score 104 (193 bits), expectation 5.0e-46, alignment length 104\nQuery: 524
    _len_ = {int} 2

```

Figure 13. A live example of a BLAST record object.

3.2. Recreating The Tabular Output

Although most of the information was readily available within the parsed XML file, a couple of the columns had to be recalculated based on the information available. Notable examples include the number of mismatches, the number of open gaps and the total match percentage. Those figures have some slight differences in the rounding or the number of significant figures and have not been adjusted in a timely enough manner to allow for an automated comparison between the original BLAST results and the parsed XML results.

Still, a brief visual comparison reveals the two outputs to be virtually indistinguishable.

Original:

```

NODE_78_length_5335_cov_124.731583
Clostridiales_bacterium_R-7__gnl|X|HMLLHLOM_15
100.000      5376  0      0      1      5376  5376  1      0.0      9928

```

Output:

```

NODE_78_length_5335_cov_124.731583
Clostridiales_bacterium_R-7__gnl|X|HMLLHLOM_15 No definition line
100.000      5376  0      0      1      5376  5376  1      0.0      9929

```

Original:

```

NODE_50641_length_51_cov_5.058824
Clostridiales_bacterium_R-7__gnl|X|HMLLHLOM_15
99.010 101      1      0      1      101      5327  5427  1.53e-44      182

```

Output:

```

NODE_50641_length_51_cov_5.058824
Clostridiales_bacterium_R-7__gnl|X|HMLLHLOM_15 No definition line
99.010 101      1      0      1      101      5327  5427  1.52589e-44  182

```

3.3. Total Query Coverage

This task proved surprisingly trivial to implement, using a single line of code to return the percentage value of a number of positive identities (individual matched nucleotides) in a high-scoring pair, divided by the total size of the query.

3.4. Counting Individual Gaps

A somewhat more involved algorithm, the individual gaps are counted in each sequence.

The algorithm iterates through a sequence, keeping track of the previous letter in the sequence.

If it finds a new gap, it notes its location relative to the sequence as well as notes 1 as its size. If the gap has already existed based on the value of the previous letter in the sequence, the algorithm instead increases the current gap by count by 1.

After the entire sequence has been iterate through, the algorithm returns a list of relative gap locations and their individual sizes.

```

def count_individual_gaps(sequence):
    last = ""
    index = 0
    gaps = []
    start = []
    for letter in sequence:
        index += 1
        if letter == "-":
            if last != "-":
                gaps.append(1)
                start.append(index)
            else:
                gaps[-1] += 1
        last = letter
    return gaps, start

```

Figure 14. The gap counting algorithm.

The then program uses the sequences' starting and end positions to determine if the match is a normal (left-to-right) or mirrored sequence. It then uses that information to calculate the actual gap locations in the sequence.

The final output is shown in *Figure 15*.

```

NODE_961993_length_139_cov_6.201439 Selenomonas_ruminantium_AB3002__gnl|X|LAMBFHGB_7
Individual gaps: Q: 1:55/1:89/1:108 S: 2:59/1:84 Query-Subject Coverage: 86.772
NODE_961993_length_139_cov_6.201439 Selenomonas_ruminantium_AB3002__gnl|X|LAMBFHGB_2
Individual gaps: Q: 1:55/1:89/1:108 S: 2:59/1:84 Query-Subject Coverage: 86.772
NODE_961993_length_139_cov_6.201439 Selenomonas_ruminantium_AB3002__gnl|X|LAMBFHGB_2
Individual gaps: Q: 1:88/1:106 S: 1:55/1:83 Query-Subject Coverage: 85.714
NODE_961993_length_139_cov_6.201439 Selenomonas_ruminantium_AB3002__gnl|X|LAMBFHGB_2
Individual gaps: Q: 1:88/1:106 S: 1:55/1:83 Query-Subject Coverage: 85.714
NODE_961993_length_139_cov_6.201439 Allisonella_histaminiformans_DSM__gnl|X|DCAGDDNC_

```

Figure 15. A sample program output. The rest of the tabular output has been cropped out of view.

The labelling system is fairly straightforward. It denotes whether the sequence is a query or a subject, followed by the gap sizes and their locations.

In the top-most instance, the output can be read as Query: 1 gap at 55, 1 gap at 89, 1 gap at 108, Subject: 2 gaps at 59, 1 gap at 84. The query-subject coverage then indicates that only 86.772% of the total query was matched.

In the event one of the sequence has no gaps, S/Q: None is displayed. In the event neither of the sequences have any gaps, only the query-subject coverage is shown.

3.5. Bag-Of-Words Matching

This aspect of the program was a major roadblock as various attempts to research how to efficiently handle a bag-of-words implementation in Python were either too focused on the broad strokes approach, or hyper-specialised with regards to a specific task. Likewise, the documentation for various libraries or frameworks was generally rather obtuse and acting more as a quick reference for someone already familiar with the general approach. In the end, credit would like to be extended to one particular blog [31] which really helped.

3.5.1. The Taxonomy Database

As previously mentioned in **section 2.5**, the official NCBI taxonomy database dump was used as a reference point for species name. In its current iteration it holds over 1.8 million species. The full details regarding the database can be found in the `taxdump_readme.txt` file [29], but of note are the 'names.dmp', 'rankedlineage.dmp' and 'fullnamelineage.dmp' files.

The 'names.dmp' file contains all the known English names of all the known species, alongside their associated taxonomy IDs. Note that multiple rows will refer back to the same ID.

9600		orang utan		orang utan <Pongo pygmaeus>		common name	
9600		Pongo pygmaeus		Pongo pygmaeus		scientific name	
9601		orang utan		orang utan <Pongo abelii>		common name	
9601		Orang-utan		Orang-utan <Sumatran orangutan>		common name	
9601		Orangutan		Orangutan <Sumatran orangutan>		common name	
9601		Pongo abelii		Pongo abelii		scientific name	
9601		Pongo pygmaeus abelii		Pongo pygmaeus abelii		synonym	
9601		Pongo pygmaeus abeli		Pongo pygmaeus abeli		synonym	
9601		Sumatran orangutan		Sumatran orangutan		genbank common name	
9602		Pongo pygmaeus pygmaeus		Pongo pygmaeus pygmaeus		scientific name	
9603		Pongo sp.		Pongo sp.		scientific name	
9603		Pongo spp.		Pongo spp.		synonym	
9604		great apes		great apes		genbank common name	
9604		Hominidae Gray, 1825		Hominidae Gray, 1825		authority	
9604		Hominidae		Hominidae		scientific name	
9604		Pongidae		Pongidae		synonym	
9605		Homo		Homo		scientific name	
9605		Homo Linnaeus, 1758		Homo Linnaeus, 1758		authority	
9605		humans		humans		common name	
9606		Homo sapiens		Homo sapiens		scientific name	
9606		Homo sapiens Linnaeus, 1758		Homo sapiens Linnaeus, 1758		authority	
9606		human		human		genbank common name	
9606		man		man		common name	
9608		Canidae		Canidae		scientific name	
9608		dog, coyote, wolf, fox		dog, coyote, wolf, fox		genbank common name	
9611		Canis		Canis		scientific name	
9612		Canis lupus		Canis lupus		scientific name	
9612		Canis lupus Linnaeus, 1758		Canis lupus Linnaeus, 1758		authority	
9612		gray wolf		gray wolf		genbank common name	
9612		grey wolf		grey wolf		common name	
9614		Canis latrans		Canis latrans		scientific name	
9614		coyote		coyote		genbank common name	
9615		beagle dog		beagle dog		includes	
9615		beagle dogs		beagle dogs		includes	
9615		Canis canis		Canis canis		synonym	
9615		Canis domesticus		Canis domesticus		synonym	
9615		Canis familiaris		Canis familiaris		synonym	
9615		Canis lupus familiaris		Canis lupus familiaris		scientific name	
9615		dog		dog		genbank common name	
9615		dogs		dogs		common name	

Figure 16. A sample of 'names.dmp' file.

In order for the results to be more usable and to cut down on the file size, this file has been pre-processed and all of the redundant data has been removed until just the names and their respective taxonomy IDs remained.

```

9600 | orang utan
9600 | Pongo pygmaeus
9601 | orang utan
9601 | Orang-utan
9601 | Orangutan
9601 | Pongo abelii
9601 | Pongo pygmaeus abelii
9601 | Pongo pygmaeus abeli
9601 | Sumatran orangutan
9602 | Pongo pygmaeus pygmaeus
9603 | Pongo sp.
9603 | Pongo spp.
9604 | great apes
9604 | Hominidae Gray, 1825
9604 | Hominidae
9604 | Pongidae
9605 | Homo
9605 | Homo Linnaeus, 1758
9605 | humans
9606 | Homo sapiens
9606 | Homo sapiens Linnaeus, 1758
9606 | human
9606 | man
9608 | Canidae
9608 | dog, coyote, wolf, fox
9611 | Canis
9612 | Canis lupus
9612 | Canis lupus Linnaeus, 1758
9612 | gray wolf
9612 | grey wolf
9614 | Canis latrans
9614 | coyote
9615 | beagle dog
9615 | beagle dogs
9615 | Canis canis
9615 | Canis domesticus
9615 | Canis familiaris
9615 | Canis lupus familiaris
9615 | dog
9615 | dogs

```

Figure 17. A sample of the `names-trimmed.txt` file, found in `bin/databases`.

3.5.2. Cleaning The Results

In order to improve the match accuracy, the method 'format_results' attempts to remove most of the 'noise' by getting rid of common terms such as "complete", "sequence", "scaffold", "chromosome", "DNA", "group", "linkage", "assembly", "strain", etc., as well as most of the common special characters.

Thus, a string such as **"gi|551368999|emb|BX284682.8| Zebrafish DNA sequence from clone DKEYP-74C5 in linkage group 22, complete sequence"** can be reduced to **"gi551368999embBX284682 8 Zebrafish DKEYP-74C5 22"** which should aid both the performance and the accuracy of bag-of-words matching.

While the hit/definition/accession issue mentioned in **section 2.6** cannot be resolved, reducing **"gi|551368999|emb|BX284682.8|"** to a single word before feeding it into vector space should vastly reduce any mismatches that it might cause.

3.5.3. Matching The Results

The algorithm for matching the results, first reads in the contents of names-trimmed.txt and splits them up into a list of indexes and a list of species names.

The species names are then fed into the scikit-learn [33] TF-IDF vectorizer which generates a Numpy [34] csr_matrix representing the bag-of-words matrix for the overall NCBI taxonomy, using the fit and transform approach which makes the data act as a training set.

The same vectorizer is then used on the cleaned-up BLAST subject names, but this time it is only fitted to the existing model. The vectorizer then returns another matrix representing the subject names.

As a side note, while loading in the list of species only takes about 2 seconds, the process of creating the two matrices will last about 20-30 seconds on the hardware used – particularly when it comes to the taxonomy matrix. Attempts have been made to keep the taxonomy matrix and/or the trained vectorizer stored as a serialised file and loaded into the program, but the two objects seem particularly picky about being freshly generated and there does not exist a simple non-convoluted way of loading them in.

The subject matrix is then iterated through, running cosine similarity comparisons between the particular entry in the subject matrix (representing one BLAST subject match) and the taxonomy matrix.

The returned matrix contains the cosine similarity between the subject in question and all the entries in the taxonomy matrix. The results are sorted by the highest cosine similarity, with the top five returned as a list – for debugging purposes.

```

2 out of: 1005 Here's a list of likely matches for gi|551368999
7955      zebrafish
0.87246530267703
1331678    zebrafish metagenome
0.7160340051988083
1286775    zebrafish nervous necrosis virus
0.5896698423757607
1764017    bacterium s-22
0.43254416088064224
1379890    bacterium r-22
0.43254416088064224
3 out of: 1005 Here's a list of likely matches for gi|690048999
387005     onchocerca flexuosa
0.8425444361959824
864677     ulva flexuosa subsp. flexuosa
0.6834418635487401
241016     calceolaria flexuosa subsp. flexuosa
0.6713409555698597
592601     chamaecrista flexuosa var. flexuosa
0.671192176118465
167778     tylophora flexuosa var. flexuosa
0.6595286230219238

```

Figure 18. A debug output indicating the top five matches for each subject.

The cropped strings are “gi|551368999|emb|BX284682.8| Zebrafish DNA sequence from clone DKEYP-74C5 in linkage group 22, complete sequence” and “gi|690048995|emb|LM563677.1| Onchocerca flexuosa genome assembly O_flexuosa_Cordoba ,scaffold OFLC_scaffold0004583”, respectively.

Since the matrices are only labelled in the form of indexes, the algorithm uses the `tax_indexes` list, originally created when loading in the taxonomy file, in order to convert the matrix index values into ‘true’ NCBI taxonomy database indexes.

The NCBI taxonomy indexes are then returned for further processing.

3.5.4. Performance

The matching algorithm is a relatively slow process. Discounting the negligible BLAST parsing times and subtracting the time it took to compute the matrices, the program took 385 seconds to process 1005 hits, for an average of 0.383 seconds per match. The 12,000 result XML set takes around 1.5 hours in order to be fully matched. It is unclear if and how the performance could be improved.

3.5.5. Accuracy

Using a very crude testing method, which merely checks whether the suggested string is part of the original string, the accuracy of the two smaller provided result sets is around 90%. The actual value is likely to be slightly higher as the method used is not very subtle and prone to false negatives.

```
def check_accuracy(original,
matched):
    a = original.lower()
    b = matched.lower()
    b = " ".join(b.split())

    if b not in a:
        return True
    return False
```

Likewise, a recurring failing match will vastly drive the overall accuracy down. One such example is the model's consistent mislabelling of "Hepatitis B" as "Hepatitis C", which while understandable, makes up a vast majority of the *sample.xml* mismatches.

The 12,000 result *contigs.xml* set could not be used as the naming convention e.g "Clostridium_mangenotii_LM2__gnl|X|KEDLAHPD_1" automatically failed 99% of the results.

Common examples of the types of false negatives include:

- **Arabis alpina genome assembly vs. arabis alpina l.**
- **Myroides profundus strain D25 vs. myroides sp. d25**
- **Gallus gallus BAC clone CH261-166B3 from chromosome z, complete sequence vs. gallus gallus gallus**

Some examples the genuine failed matches include:

- **Bos taurus BAC CH240-105A12 (Children's Hospital Oakland Research Institute Bovine BAC Library (male)) complete sequence vs. bacillus sp. bac**
- **Mouse DNA sequence from clone RP23-118K22 on chromosome 2, complete sequence vs glomus sp. rp23**
- **Mus musculus 7 days neonate cerebellum cDNA, RIKEN full-length enriched library, clone:A730010G01 product:unclassifiable, full insert sequence vs.**
- **expression vector full-length t7 tnap**
- **Mouse DNA sequence from clone DN-29B18 on chromosome 1, complete sequence vs pseudomonas sp. dn-2**
- **Hepatitis B virus isolate s04007149 S protein gene, complete cds vs. hepatitis c virus isolate h**

Some of the mismatches such as Hepatitis B vs C are reasonable, but it seems as though the model particularly struggles with single word species names. It is unclear what could be done to improve the accuracy, though attempting to use the same bag-of-words model without employing tf-idf might be worth exploring.

In the second example, it is highly likely that “Mouse” was a much more common term than “RP23” and therefore matching “glomus sp. **rp23**” to “**RP23-118K22**” was more favoured by the model. In any case, the objective correct match is almost always within the top five matches found.

Nevertheless, the fact that the model still achieved a 90% accuracy rate based on what is possibly the least robust accuracy testing methods, indicates good results with regards to further analytics.

3.6. Generating Lineages

The program takes in the IDs, previously acquired in TaxMatch, and uses them to generate full taxonomy trees.

One of the benefits of using the NCBI taxonomy database IDs as the identifiers was the ability to access additional information from the database to greatly enhance the output. It also quickly became apparent that given a number of tax IDs, it only took a little more effort to generate a full ranked taxonomy, and the act doing so would vastly streamline the rest of the design.

This was a relatively challenging, but enjoyable part of the development.

The program was originally based around the full lineages contained in the “fullnamelineage.dmp” file (See **appendix 4.1.4**), with parsing and tree generation fully implemented. However, it soon became apparent that this approach lacked an easy way to scrape any useful information about the complete results set.

Consequently, the program and existing code was repurposed to instead make use of the “rankedlineage.dmp” file which contained a slightly abridged version of the data, but made up for it by being ranked according to the Superkingdom-Kingdom-Phylum-Class-Order-Family-Genus-Species. (See **appendix 4.1.5**)

3.7. Ranked Lineages

Handling the rankedlineage.dmp input was fairly easy to understand, but fairly awkward to implement. The file is split into ten separated values indicating in order:

Tax Name | Name | Species | Genus | Family | Order | Class | Phylum | Kingdom | Superkingdom |

An example of that is the “Homo sapiens neanderthalensis” row which is as follows:

63221 | Homo sapiens neanderthalensis | Homo sapiens | Homo | Hominidae | Primates | Mammalia | Chordata | Metazoa | Eukaryota |

Issues arose from the fact that the name column carried multiple meanings depending on the overall context of row. For a clearer, related example:

9606 | Homo sapiens | | Homo | Hominidae | Primates | Mammalia | Chordata | Metazoa | Eukaryota |

In the previous instance, Homo sapiens neanderthalensis indicated a sub-species of Homo Sapiens. In this instance, the species column is empty, thereby indicating that Homo Sapiens is the species.

More so, in some cases it is possible for a species to skip some of the immediate ranks as is the case with the example below.

**889948 | Candidatus Nanosalina sp. J07AB43 | | Candidatus Nanosalina | | |
Nanohaloarchaea | Candidatus Nanohaloarchaeota | | Archaea |**

Candidatus Nanosalina sp. J07AB43 is a species with no family, order or kingdom.

Thereby, as the rows are retrieved and parsed, they need to be processed as follows:

1. The tax ID needs to be stripped.
2. Each of the remaining columns needs to be assigned its respective rank.
3. The name column needs to overwrite the right-most consecutive empty column.
4. The remaining empty columns must be removed.

A relatively simple process once explained, but in practice it made for some obtuse looking code (see **appendix 4.1.6**)

The corrected lineage is then added to the dynamically-built taxonomy tree.

3.8. Taxonomy Tree

The Taxonomy Tree is a relatively standard custom tree data structure. It holds the information about the name of the node, the rank of the node, the parent node, and the children nodes.

The two notable features it has, is its ability to recursively count the total number of children under each node and further from there, use that information to report the most and least common sets of results in a given rank.

3.9. Most/Least Common Results

By entering a command line parameter in the form of “[rank][number of results]”, e.g. -m “Species 20 Phylum 3 Genus 7” for most common, the user can be provided with an output from the taxonomy tree. There is also the “All [Number]” option which will print out the provided number of results for every rank.

The arguments are first passed through a rigorous set of syntax checks in `Main.check_indexing_args`, before then being passed onto `TaxTree`. There, they are separated into pairs and ran iteratively through the `calculateCommonnessByRank` method.

3.9.1. The Algorithm

The algorithm iterates through the entire tree, checking the nodes by each rank. Upon finding a node of the specified rank, it will count the total amount of children it has relative to the total amount of children in the master root node. The name of the node and the percentage of the total tree this node makes up are then added to a list. After the entire tree has been traversed, the list is returned and sorted based on the Boolean value specified – either ascending or descending. In the event that the total number of results in that particular rank is less than the number of results specified, all of the results available will be printed out.

```
The most commonly occurring values for Species are:
5.9405940594059405 % - Cucumis melo
5.9405940594059405 % - Mycoplasma hyorhinitis
4.9504950495049505 % - Cyprinus carpio
4.9504950495049505 % - Oryzias latipes
4.9504950495049505 % - Vigna angularis
3.9603960396039604 % - Danio rerio
2.9702970297029703 % - Gossypium hirsutum
2.9702970297029703 % - Solanum pennellii
1.9801980198019802 % - Apis florea
1.9801980198019802 % - Parasteatoda tepidariorum
1.9801980198019802 % - Cynoglossus semilaevis
1.9801980198019802 % - Dracunculus medinensis
1.9801980198019802 % - Wuchereria bancrofti
1.9801980198019802 % - Strongyloides stercoralis
1.9801980198019802 % - Arabis alpina
1.9801980198019802 % - Theobroma cacao
1.9801980198019802 % - Solanum lycopersicum
1.9801980198019802 % - Populus trichocarpa
1.9801980198019802 % - Plasmodium chabaudi
1.9801980198019802 % - Methanococcus maripaludis
The most commonly occurring values for Phylum are:
27.722772277227726 % - Streptophyta
21.782178217821784 % - Chordata
9.900990099009901 % - Nematoda
The most commonly occurring values for Genus are:
5.9405940594059405 % - Cucumis
5.9405940594059405 % - Mycoplasma
4.9504950495049505 % - Cyprinus
4.9504950495049505 % - Oryzias
4.9504950495049505 % - Vigna
4.9504950495049505 % - Solanum
3.9603960396039604 % - Danio
Finished in 80.35824179649353
```

Figure 19. An example of the output based on -m "Species 20 Phylum 3 Genus 7" and "shortsamples.xml"

3.10. Visualisations

A couple of different methods were tried for handling tree visualisations, including Graphviz [35] and NetworkX [36], as well a few more obscure open-source Newick-format [37] based libraries. Regrettably neither of those could handle significant amounts of data, such as more than 100 nodes.

A significant breakthrough was made towards the end, courtesy of ETE Toolkit [38], but the final result is a little rushed and not as polished as would have been preferable.

In particular it is likely that the existing taxonomy tree would have been refactored into a sub-class of the ETETree rather than the current implementation which involves first populating the taxonomy tree, then populating the ETETree with the data from the taxonomy tree.

3.10.1. ASCII

```

/ Danio - Danio rerio
/ Cypriniformes Cyprinidae Cyprinus - Cyprinus carpio
\ Sinocyclocheilus - Sinocyclocheilus grahami
- Beloniformes Adrianichthyidae Oryzias - Oryzias latipes
/ Actinopteri Sciaenidae Larimichthys - Larimichthys crocea
- Centropomidae Lates - Lates calcarifer
- Pleuronectiformes Cynoglossidae Cynoglossus - Cynoglossus semilaevis
\ Fundulidae Fundulus - Fundulus heteroclitus
\ Cyprinodontiformes
| / Xiphophorus - Xiphophorus maculatus
| \ Poeciliidae
| | /- Poecilia latipinna
| | \ Poecilia
| | \- Poecilia mexicana
- Amphibia Anura Pipidae Xenopus - Xenopus laevis
/ Ovis - Ovis canadensis
/ Bovidae
| /- Bos mutus
| \ Bos
| \- Bos taurus
/ Hylobatidae Nomascus - Nomascus leucogenys
/ Homo - Homo sapiens
- Pan - Pan troglodytes
- Hominidae /- Pongo pygmaeus
| Pongo
| \ Pongo abelii
| Gorilla - Gorilla beringei
| /- Macaca fascicularis
- Primates / Macaca
| \ Macaca mulatta
| Chlorocebus - Chlorocebus sabaeus
- Cercopithecidae Papio - Papio anubis
| Ptilocolobus - Ptilocolobus tephrosceles
| Rhinopithecus - Rhinopithecus bieti
- Cebus - Cebus capucinus
- Mammalia Cebidae
| Callithrix - Callithrix jacchus
| Aotidae Aotus - Aotus nancymae
/ Chordata
| /- Mus musculus
| / Muridae Mus
| \ Mus caroli
| / Heterocephalus - Heterocephalus glaber
- Rodentia Bathyergidae
| Fukomys - Fukomys damarensis
| Spalacidae Nannospalax - Nannospalax galili
| Caviidae Cavia - Cavia porcellus
- Carnivora Canidae Canis - Canis lupus
| /- Pteropus vampyrus
| / Pteropus
| / Pteropodidae \- Pteropus alecto

```

Figure 20. A zoomed-out section of the ASCII output.

The ASCII output is capable of providing a brief overview of the results, in terms of subject matches that have appeared and their respective taxonomy rankings. It can be displayed in the terminal, saved to a separate file or appended to the same file as the BLAST results. It lacks a lot of the crucial details but can be used for quick reference.

3.10.2. ETE Tree

The ETE visualisation is far more interesting as it comes with an interactive representation of the tree. The ranks are labelled and colour-coded and each node denotes the number of nodes under it.

Due to performance issues, the nodes are capped at "Species" as rendering all the leaves will generally either make the visualisation unusable or outright crash the program. The species nodes will, however, still denote the number of individual matches under them.

An attempt was made to have the thickness of the tree border increase in size based on areas where the results are heavily concentrated, though the results can be hit-or-miss and require further tweaking.

The interactive tree can be saved in Newick format, rendered as a whole in PDF, or have a section of it rendered in PDF.

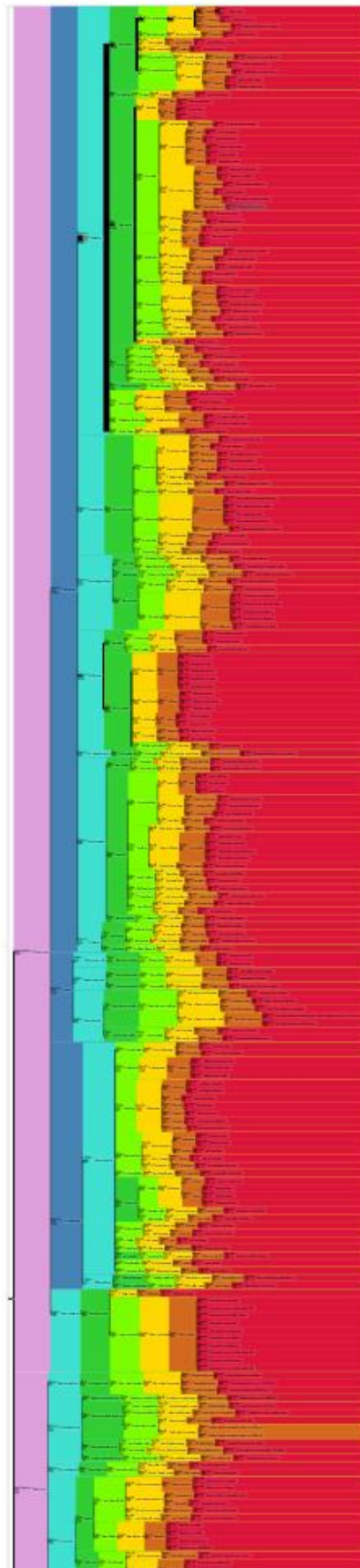


Figure 21. A zoomed-out example of a 1000 node tree.

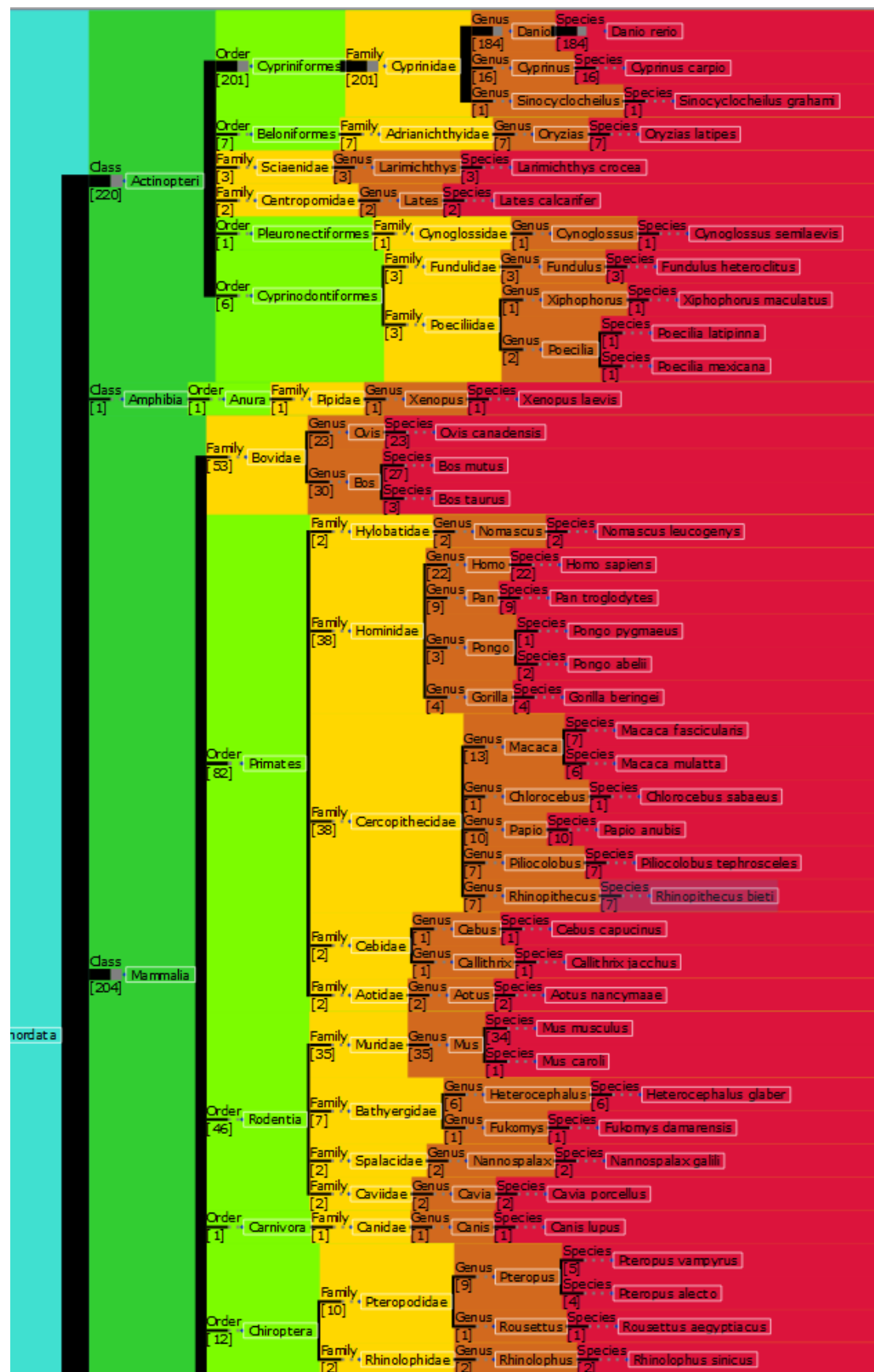


Figure 22. An up-close example of the same tree.

3.11. Interface

The interface relies entirely on command-line input and has been rigorously tested and error-proofed.

The available commands are:

- `--input` or `-i`: the file path of the XML file acting as a source
- `--output` or `-o`: the file path of the output
- `--verbose` or `-v`: currently unimplemented, as for the purposes of demonstration, most of the debug output has been left enabled. Verbose mode would enable or disable active terminal output.
- `--ascii` or `-at`: renders the ASCII tree in terminal.
- `--save_ascii` or `-sa`: Saves the ASCII tree to a specified file or appends it to the default output file if “default” is entered. Interestingly enough, one could not find a way to have this parameter be allowed to be left empty, hence having to type in “default”.
- `--most` or `-m`: Show the most common results per rank, based on the syntax specified in **section 3.9**.
- `--least` or `-l`: Show the least common results per rank, based on the syntax specified in **section 3.9**.
- `--visualise` or `-vt`: Show a GUI-based visualisation of the tree.

```
parser = argparse.ArgumentParser(description='Augmenting BLAST results.')
parser.add_argument("--input", "-i",
                    help=("(The path of the BLAST XML results.")
parser.add_argument("--out", "-o",
                    help=("(The path for the processed results"))
parser.add_argument("--verbose", "-v",
                    help=("(Print complete output in the terminal."),
                    action="store_true")
parser.add_argument("--ascii", "-at",
                    help=("(Print an ASCII representation of the processed results in terminal."
                          "Will require use of taxonomy database matching and may therefore take a long time for large result sets."),
                    action="store_true")
parser.add_argument("--save_ascii", "-sa",
                    help=("(Save the ASCII representation of the taxonomy tree to a specified file or enter 'default' to append to the end of existing output file.")
parser.add_argument("--most", "-m", nargs='?',
                    help=("(List the most commonly occurring matches. Format: [Rank][Number of results] e.g: -m 'Genus 5 Family 5 Class 5'"
                          "Accepts: Species Genus Family Order Class Phylum Kingdom Superkingdom or All"
                          "Will require use of taxonomy database matching and may therefore take a long time for large result sets."),
                    action="store",
                    default=None)
parser.add_argument("--least", "-l", nargs='?',
                    help=("(List the least commonly occurring matches. Format: [Rank][Number of results] e.g: -l 'Genus 5 Family 5 Class 5'"
                          "Accepts: Species Genus Family Order Class Phylum Kingdom Superkingdom or All"
                          "Will require use of taxonomy database matching and may therefore take a long time for large result sets."),
                    action="store",
                    default = None)
parser.add_argument("--visualise", "-vt",
                    help="Show a GUI representation of the taxonomy tree after the results have been processed.",
                    action="store_true")
```

Figure 23. Command-line parameter definitions.

3.12. Final Thoughts

In terms of the initially outlined requirements, all of them have been fulfilled, and the feature scope was further expanded with the inclusion of visualised taxonomy trees.

The Jupyter Notebook based GUI approach was not touched upon, which is regrettable, as it seemed like an interesting technology to try out.

Overall, the final state of the product is functional and satisfactory, but there are a lot of minor quality-of-life improvements that could be introduced in terms of the interface and general usability, if only they were thought of slightly earlier.

Notable ideas include making some of the current debug output an option-toggable feature, sorting the tree children nodes in order of how many total children they have, integrating BLAST API functionality directly into the program, actually implementing a Jupyter Notebook front end, etc.

4. Testing

The testing aspect is, admittedly, lacking some thoroughness. Although Extreme Programming typically encourages Test-Driven Development, a lot of the overall structure and design kept changing throughout the project, which meant prospective unit tests would keep having to be scrapped.

On top of that, due to the limited amount of inputs and outputs, relatively low technical complexity/intricacy, quick running times, and the fact the work was done by a single developer who was inherently familiar with the code, the vast majority of the time it was far easier and more convenient to keep iteratively stepping through new methods using the in-built IDE debugger to make sure everything was in order, than it was to write out and run unit tests.

This approach did cause some issues further down the line as some methods needed to be refactored due to being more coupled than was convenient (and TaxTree still needs some further refactoring), but ultimately the program does have some degree of automated unit/integration testing for the most barebones forms of troubleshooting, even if the bulk of the testing was still done through a manual black-box approach.

Due to the fact a lot of the data structures involved were relatively complex, a lot of the unit tests rely on serialised data originally just “grabbed” through pickle and then unpickled for the purposes of the tests.

Seeing how this approach was done in a really ad-hoc manner at the time, and the data structures will likely have to re-pickled if their functionality changes significantly, a specific script that runs the program and creates new copies of serialized data (after said data has been verified to be correct, of course) would likely be a useful addition.

4.1. Automated Tests

There are currently seven automated tests in the program, six of which are centred around the TaxTree class. Admittedly, the TaxTree class is where most of the future developments would take place and therefore where the tests are most crucial.

`def test_extract_data(self)` – ensures that the information parsed in from `rankedlineage.dmp` is correctly formatted into a list.

`def test_lineage_fix(self)` – ensures that the algorithm responsible for fixing up the taxonomy (as mentioned in **section 3.7**) is working as intended.

`def test_most & test_least` – ensures the correct output for counting most/least commonly occurring ranks

`def test_ETETree(self)` - ensures the ETE Toolkit generated tree remains consistent. This method will eventually need reworking as due to the lack of an easy way to test a tree without a comprehensive method involving full traversal, the expected tree and the actual tree are currently just having `Tree.describe()` methods run on them, to provide some rudimentary statistics.

def tree_integrity_ascii(self) – Ensures that all the aspects of tree generation work as expected by ensuring the actual ASCII representation matches the expected ASCII representation.

def test_name_cleaning(self) – ensures the test dataset is correctly stripped of stop words and special characters as discussed in **section 3.5.2**

4.2. Manual Test Cases

Test ID	Description	Steps	Inputs	Expected Result	Pass/Fail
Test 1	Correct XML file provided.	Launch the program with the parameters provided.	-i "xmlesamples/shortsample.xml"	The BLAST results start processing.	Pass
Test 2	Incorrect filename provided.	Launch the program with the parameters provided.	-i "xmlesamples/shortttsample.xml"	Unhandled Error	Pass
Test 3	Empty file provided	Launch the program with the parameters provided.	-i "xmlesamples/empty.xml"	Unhandled Error	Pass
Test 4	Save to output.	Launch the program with the parameters provided.	-i "xmlesamples/shortsample.xml" -o "output.txt"	Output.txt file created, containing BLAST tabular output and additional information.	Pass
Test 5	Enable verbose mode	Launch the program with the parameters provided.	-i "xmlesamples/shortsample.xml" -v	Should run as normal as the feature is not enabled.	Pass
Test 6	Empty input	Launch the program with the parameters provided.	-i	Unhandled Error	Pass
Test 7	Empty output	Launch the program with the parameters provided.	-o	Unhandled Error	Pass
Test 8	ASCII Tree	Launch the program with the parameters provided.	-i "xmlesamples/shortsample.xml" -at	An ASCII tree should be rendered in terminal at the end of the program, before the program exits.	Pass

Test 9	Save ASCII Tree to file.	Launch the program with the parameters provided.	-i "xmlesamples/shortsample.xml" -sa "tree_output.txt"	A new text file containing the ASCII tree should appear.	Pass
Test 10	Append ASCII to the existing output file	Launch the program with the parameters provided.	-i "xmlesamples/shortsample.xml" -o "output.txt" -sa "default"	The ASCII tree is appended to the end of the existing output file.	Pass
Test 11	Toggle the GUI visualisation.	Launch the program with the parameters provided.	-i "xmlesamples/shortsample.xml" -vt	The GUI interface appears.	Pass
Test 12	No input for most/least	Launch the program with the parameters provided.	-i "xmlesamples/shortsample.xml" -m	Handled Error	Semi-Fail [1]
Test 13	Empty input for most/least	Launch the program with the parameters provided.	-i "xmlesamples/shortsample.xml" -m "	Handled Error	Semi-Fail [2]
Test 14	Incorrect number of arguments for most/least	Launch the program with the parameters provided.	-i "xmlesamples/shortsample.xml" -m "Species 5 Genus 2 Family"	Handled Error	Pass
Test 15	Incorrect spelling of a rank for most/least.	Launch the program with the parameters provided.	-i "xmlesamples/shortsample.xml" -m "Species 5 Gennus 2 Family 5"	Handled Error	Pass
Test 16	Not a digit for most/least	Launch the program with the parameters provided.	-i "xmlesamples/shortsample.xml" -m "Species A Genus 2 Family 5"	Handled Error	Pass
Test 17	Correctly parses "All"	Launch the program with the parameters provided.	-i "xmlesamples/shortsample.xml" -m "All 5"	Most common statistics for 5 of each rank are provided.	Pass

[1] – The argument was set to 'None' which is the same as the default state. It therefore passed through the safety checks as the existing safety checks were coded to just ignore it. As the functionality did not trigger, none of the behaviour changed which is technically fine, but it is an unexpected outcome.

[2] Parsed as an empty list. There was supposed to be a check for

if $\text{len}(\text{list}) \% 2$ is not 0: throw error, but since $0 / 2 = 0$, it passed.

The program will still crash as the code is trying to access an empty list, but this is still an unexpected outcome.

5. Critical Evaluation

Overall, the final outcome of this project has been satisfactory in terms what has been accomplished. A part of me feels as though I could have definitely done more and the initial requirements provided were laid out with the expectations of more to come, even beyond the implemented expanded features.

Yet at the same time, neither the technical submission nor this final report seem particularly lacking relative to the work of my student contemporaries, so I guess perhaps I am just slightly unsure as to what is typically expected from a major project.

5.1. Objectives

In terms the initial outlined functionality and overall usability, all the requirements have been met and even in its current state, the project should be capable of being of an immediate use to any possible bioinformatician, even in spite of some of the minor existing shortcomings. I am particularly pleased with the way the tree visualisations turned out in spite of the fact it took a while to find a framework that could properly handle it and, as such, the final implementation is somewhat rushed.

I will say however, that as fascinating as it was to explore a scientific field I have previously had no experience in and was able to solve real practical problems with regards to, the fact I was and still am fundamentally a little out of my depth with regards to the Bioinformatics occasionally made it a frustrating experience when attempting to simultaneously learn it, design and implement at what often felt like the same time. Which goes back to the problem domain issues, originally noted in **section 1.2.2.**

I still would have likely picked this project if I had the choice to choose again, but at times I did wish I had picked one with a more transparent structure and goals that could be more easily “waterfalled”, so to speak.

5.2. Process

While the various technical practices of Extreme Programming have worked out extremely well in terms of the overall design - particularly going for the simplest implementation possible, iterating from there, and refactoring back to basics, I feel as though the actual planning/managerial aspect of this “one-man XP variant” has failed on account of being just a bit too lax.

It seems like a lot of students, myself included, are lured by XP's hyper-agile approach to software engineering and a relative lack of “boring management paperwork to grind through”, but in practice Extreme Programming has to either be adhered to all the way, or not at all. There are no half-measures here, and I now feel as though one of the primary benefits of pair programming is just having a fellow person on your back to keep you sharp and focused.

While the project started off fairly strong, I ended up falling victim to a really bad flu around March which left me bedridden for little under two weeks, with another two weeks to properly recover - at which point it was already Easter. Coupled with some existing development roadblocks at the time, this really took the wind out of me and killed all of my momentum, which made me partly disappear and detach myself for a while there with regards to the overall project.

I feel as though I definitely brought it back in the end, but my biggest regret is not getting in touch with my supervisor more in order to get back on track. While I cannot say for sure, I think combining some of the XP practices with the largely overlying aspects of Scrum or Kanban would have vastly helped with staying on top of things.

5.3. Tools

One of the biggest upsides to this project is my newly found love of Python. Python has always been one of those languages I would promise myself I would learn, but it always ended up being put on the backburner on account of its simplicity. I reasoned I could always pick it up over a couple of days, so I never got around to it.

While the lack of a proper 2D array structure is rather annoying and an issue that has come up a couple of times during the project, I am glad I decided to challenge myself and chose Python over Java as the language is simply delightful – though the fact PyCharm is an absolutely excellent tool (as is most IntelliJ software, to be fair) hugely contributed to my workflow and overall enjoyment of the language.

As a big fan of machine learning module, I enjoyed being able to utilise it as part of this project – even if that particular feature was one of the more annoying aspects.

5.4. Future Work

Before focusing on any future features, I would first like to carry on polishing up and further refactoring the existing code. While I would argue that the overall quality is good, there are some particular areas that could stand to be improved.

After the initial further refactoring, I would primarily focus on developing some proper form a test harness to really stress the system and greatly expand the test functionality.

Most of the issues have been mentioned throughout this report, but generally speaking, the entire TaxTree class is far more clunky than I would have liked due to way the absolute rootNode of the tree needs to be passed onto multiple methods in order for them to work. This was essentially an ad-hoc fix that was deemed good enough at a time and should absolutely be refactored into something slightly more sensible.

The code relating to the ETE Toolkit library is also rather “messy” as that particular library was a relatively addition, following a bunch of other approaches that did not work. The current tree visualisation features certainly need to be tweaked and I am considering further enhancing the existing interface to perhaps make it possible to interact with the nodes. For example, clicking on an individual node could display all of the BLAST results associated with it.

I would, also, like to explore further methods of machine learning to further improve the existing accuracy.

6. Appendices

6.1. Third-Party Code, Libraries And Technologies Used

[1] C. S. Perone, "Machine Learning :: Cosine Similarity for Vector Space Models (Part III) | Terra Incognita," 12 September 2013. [Online]. Available: <http://blog.christianperone.com/2013/09/machine-learning-cosine-similarity-for-vector-space-models-part-iii/>. [Accessed 3 May 2018].

Code is perhaps a bit of a stretch as the examples cited are only about five lines, but nevertheless for the sake of disclosure, it should be included. This was my primary reference point for how to implement a tf-idf vectorizer in Python.

[2] "Biopython · Biopython," [Online]. Available: <http://biopython.org/>. [Accessed 3 May 2018].

This library has been used for parsing the BLAST XML files.

As per the license agreement located at
<https://github.com/biopython/biopython/blob/master/LICENSE.rst>

"Permission to use, copy, modify, and distribute this software and its documentation with or without modifications and for any purpose and without fee is hereby granted, provided that any copyright notices appear in all copies and that both those copyright notices and this permission notice appear in supporting documentation, and that the names of the contributors or copyright holders not be used in advertising or publicity pertaining to distribution of the software without specific prior permission."

No changes to the BioPython library have been made.

[3] "scikit-learn: machine learning in Python — scikit-learn 0.19.1 documentation," [Online]. Available: <http://scikit-learn.org/stable/>. [Accessed 3 May 2018].

The website states that scikit-learn has a BSD license.

This library has been used for its machine learning tools.

The BSD license states:

"Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
3. Neither the name of the copyright holder nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission."

No changes to the scikit-learn library have been made.

[4] "ETE Toolkit - Analysis and Visualization of (phylogenetic) trees," [Online]. Available: <http://etetoolkit.org/>. [Accessed 3 May 2018].

This library has been used for its tree visualisation capabilities. This library has no been modified.

The bottom of the website states that "ETE is free software (GPL)". The actual full text of the GPL license is a bit of a handful and can be found under <https://www.gnu.org/licenses/gpl-3.0.en.html>.

ETE Toolkit may in some circumstances require an additional installation of PyQt 4/5 in order to work. PyQt 4/5 are also licensed under GPL.

I guess this technically renders my project part of the GPL license. I am not entirely sure what this means in practical terms, but I am willing to co-operate in any way necessary.

6.2. Code Samples

6.2.1. Iterating Over The BLAST Results

```
def blast_parse(blast_records):

    blast_total_processed = 0
    blast_results_matched = 0
    blast_individual_hits = 0

    match_names = []

    if args.out: file = open(args.out, 'w')

    start = time.time()

    for blast_record in blastXMLRecords:
        blast_total_processed += 1
        for alignment in blast_record.alignments:
            blast_results_matched += 1
            for hsp in alignment.hsps:
                blast_individual_hits += 1

                tab = compile_tab_output(blast_record, alignment, hsp)
                print(tab)

                stats = get_gaps_and_coverage(blast_record, hsp)
                print(stats)
                match_names.append(alignment.title)

            if args.out:
                file.write(tab)
                file.write(stats)

    end = time.time()
    print("BLAST parsing finished in ", end - start)
    print("BLAST results processed:", blast_total_processed)
    print("BLAST results with hits:", blast_results_matched)
    print("BLAST total hits:", blast_individual_hits)

    return match_names
```

6.2.2. Providing The Tabular Output

```

"""A wrapper function for compiling the tabular BLAST output. Takes in current blast_record, alignment and HSP """
def compile_tab_output(blast_record, alignment, hsp):
    tab_output = (blast_record.query,
                  alignment.title,
                  calculate_match_percentage(hsp),
                  hsp.align_length,
                  calculate_mismatches(hsp),
                  calculate_gap_opens(hsp),
                  hsp.query_start,
                  hsp.query_end,
                  hsp.sbjct_start,
                  hsp.sbjct_end,
                  hsp.expect,
                  round(hsp.bits))

    tab_output = '\t'.join(map(str, tab_output))
    tab_output += '\n'

    return tab_output

```

6.2.3. Cleaning The Results

```

def format_results(results):

    # Mildly redundant as the vectoriser probably has a way to insert stop words,
    # but frankly I don't want to dive into that documentation anymore. Possibly refactor later.

    stop_words = ("complete sequence", " complete ", " sequence ", " scaffold ", " chromosome ",
                  " DNA ", " group ", " linkage ", " assembly ", " strain ",
                  " PREDICTED ", " hypothetical ", " partial ", " protein ",
                  " from ", " transcript ", " variant ", " clone ", " in ", " genome ", " contig ", " mRNA ")

    cleaned=[]

    for x in results:
        #print(x) #DEBUG
        x = x + " "
        x = x.replace("_", " ")
        x = x.replace(","," ")
        x = x.replace("||", "") # Intentionally replaced with no space to reduce the effect it will have on bag of words.
        x = x.replace(".", " ")
        x = x.replace("=", " ")
        x = x.replace(";", " ")
        x = x.replace(":", " ")

        for word in stop_words:
            x = x.replace(word, " ")
        cleaned.append(x)

    return cleaned

```

6.2.4. Sample Of The FullNameLineage.dmp File

```

2012404      |      Rotylenchus sp. D9-5 |      cellular organisms; Eukaryota;
Opisthokonta; Metazoa; Eumetazoa; Bilateria; Protostomia; Ecdysozoa; Nematoda;
Chromadorea; Tylenchida; Tylenchomorpha; Tylenchoidea; Hoplolaimidae;
Rotylenchulinae; Rotylenchus;
290124      |      Rotylenchus sp. JH-2004 |      cellular organisms;
Eukaryota; Opisthokonta; Metazoa; Eumetazoa; Bilateria; Protostomia; Ecdysozoa;
Nematoda; Chromadorea; Tylenchida; Tylenchomorpha; Tylenchoidea;
Hoplolaimidae; Rotylenchulinae; Rotylenchus;
1497539     |      Rotylenchus sp. KD-sp15 |      cellular organisms;
Eukaryota; Opisthokonta; Metazoa; Eumetazoa; Bilateria; Protostomia; Ecdysozoa;
Nematoda; Chromadorea; Tylenchida; Tylenchomorpha; Tylenchoidea;
Hoplolaimidae; Rotylenchulinae; Rotylenchus;
364624      |      Rotylenchus sp. SAS-2006 |      cellular organisms;
Eukaryota; Opisthokonta; Metazoa; Eumetazoa; Bilateria; Protostomia; Ecdysozoa;
Nematoda; Chromadorea; Tylenchida; Tylenchomorpha; Tylenchoidea;
Hoplolaimidae; Rotylenchulinae; Rotylenchus;
984665      |      environmental samples |      cellular organisms;
Eukaryota; Opisthokonta; Metazoa; Eumetazoa; Bilateria; Protostomia; Ecdysozoa;
Nematoda; Chromadorea; Tylenchida; Tylenchomorpha; Tylenchoidea;
Hoplolaimidae;
984641      |      Hoplolaimidae environmental sample |      cellular
organisms; Eukaryota; Opisthokonta; Metazoa; Eumetazoa; Bilateria; Protostomia;
Ecdysozoa; Nematoda; Chromadorea; Tylenchida; Tylenchomorpha; Tylenchoidea;
Hoplolaimidae; environmental samples;
315429      |      Meloidogynidae |      cellular organisms; Eukaryota;
Opisthokonta; Metazoa; Eumetazoa; Bilateria; Protostomia; Ecdysozoa; Nematoda;
Chromadorea; Tylenchida; Tylenchomorpha; Tylenchoidea;
33285 |      Meloidogyninae |      cellular organisms; Eukaryota;
Opisthokonta; Metazoa; Eumetazoa; Bilateria; Protostomia; Ecdysozoa; Nematoda;
Chromadorea; Tylenchida; Tylenchomorpha; Tylenchoidea; Meloidogynidae;
189290      |      Meloidogyne |      cellular organisms; Eukaryota;
Opisthokonta; Metazoa; Eumetazoa; Bilateria; Protostomia; Ecdysozoa; Nematoda;
Chromadorea; Tylenchida; Tylenchomorpha; Tylenchoidea; Meloidogynidae;
Meloidogyninae;
1929980     |      Meloidogyne aberrans |      cellular organisms;
Eukaryota; Opisthokonta; Metazoa; Eumetazoa; Bilateria; Protostomia; Ecdysozoa;
Nematoda; Chromadorea; Tylenchida; Tylenchomorpha; Tylenchoidea;
Meloidogynidae; Meloidogyninae; Meloidogyne;
1965296     |      Meloidogyne africana |      cellular organisms; Eukaryota;
Opisthokonta; Metazoa; Eumetazoa; Bilateria; Protostomia; Ecdysozoa; Nematoda;
Chromadorea; Tylenchida; Tylenchomorpha; Tylenchoidea; Meloidogynidae;
Meloidogyninae; Meloidogyne;
325750      |      Meloidogyne arabicida |      cellular organisms;
Eukaryota; Opisthokonta; Metazoa; Eumetazoa; Bilateria; Protostomia; Ecdysozoa;
Nematoda; Chromadorea; Tylenchida; Tylenchomorpha; Tylenchoidea;
Meloidogynidae; Meloidogyninae; Meloidogyne;
288492      |      Meloidogyne ardenensis |      cellular organisms;
Eukaryota; Opisthokonta; Metazoa; Eumetazoa; Bilateria; Protostomia; Ecdysozoa;
Nematoda; Chromadorea; Tylenchida; Tylenchomorpha; Tylenchoidea;
Meloidogynidae; Meloidogyninae; Meloidogyne;

```

6.2.5. Sample Of The RankedLineage.dpm File

```

#9606 |Homo sapiens|      |Homo |Hominidae|Primates| Mammalia|Chordata |
      |Metazoa      |      |Eukaryota |
#63221 |      |      |Homo sapiens neanderthalensis |      |Homo sapiens|
      |Homo |      |Hominidae |      |Primates |      |Mammalia |
      |Chordata |      |Metazoa |      |Eukaryota |
#741158 |      |      |Homo sapiens subsp. 'Denisova' |      |Homo sapiens|
      |Homo |      |Hominidae |      |Primates |      |Mammalia |
      |Chordata |      |Metazoa |      |Eukaryota |
#11676 |      |      |Human immunodeficiency virus 1 |      |
      |Lentivirus |      |Retroviridae |      |
      |      |      |Viruses |
#7157 |      |Culicidae |      |
      |Diptera|      |Insecta|      |Arthropoda |      |Metazoa |
      |Eukaryota |
#842271 |      |      |Lepidoptera sp. BOLD:AAE5441 |      |
      |      |      |Lepidoptera |      |Insecta|      |Arthropoda |
      |Metazoa |      |Eukaryota |
#1350382 |      |      |Scenedesmus sp. LUCC 015 |      |
      |Scenedesmus|      |Scenedesmaceae |      |Sphaeropleales |
      |Chlorophyceae |      |Chlorophyta |      |Viridiplantae |
      |Eukaryota |
#1289305 |      |      |Chlorophyceae sp. WJT24VFNP4 |      |
      |      |      |Chlorophyceae |      |
      |Chlorophyta |      |Viridiplantae |      |Eukaryota |
#1427098 |      |      |Pseudomonas sp. 161(2013) |      |
      |Pseudomonas|      |Pseudomonadaceae |      |Pseudomonadales |
      |Gammaproteobacteria |      |Proteobacteria|
      |Bacteria |
#1042 |      |Erythrobacter sp. |      |
      |Erythrobacteraceae |      |Sphingomonadales |      |
      |Alphaproteobacteria |      |Proteobacteria|      |Bacteria
      |

```

6.2.6. Processing The Ranked Lineage Data

```
def prepare_for_ranked_tree(ranked_lineage, rootNode):
    ranks = ['Name', 'Species', 'Genus', 'Family', 'Order',
            'Class', 'Phylum', 'Kingdom', 'Superkingdom']

    combined = []

    count = 0
    for item in ranked_lineage: # A better approach would have been to use the double-iterate zip function
        list = [] # but I think this is readable enough.
        list.append(ranks[count])
        list.append(item)
        combined.append(list)
        count += 1

    combined = fixLineage(combined) # fixes the gaps in the lineage.
    build_ranked_tree(combined, rootNode)
```

```
def fixLineage(taxonomy_list):
    # So, this is doing a lot of obtuse looking direct array operations, but there's a reason for that.
    #
    # The list holds lists of two so taxonomy_list[0] will contain another list with
    # ["Candidatus Iainarchaeum andersonii", "Name"], which should make some of what is happening more obvious.
    #
    # A ranked lineage may look like this:
    # Candidatus Iainarchaeum andersonii | Candidatus Iainarchaeum | | | Candidatus Diapherotrites | Archaea |
    #
    # This corresponds to Name|Species|Genus|Family|Order|Class|Phylum|Kingdom|Superkingdom
    # Species is empty which means Candidatus needs to be moved up there as it *is* the species.
    # This is so you can still have subspecies like Homo Neanderthalis > Homo Sapien, without a specific sub-species rank.
    #
    # There's data missing in Family, Order, Class, Kingdom which means this lineage just doesn't belong to a specific
    # subset and these need to be removed from the chain.

    name = taxonomy_list[0] # take out the Name
    remaining_lineage = taxonomy_list[1:9] # keep Species to Superkingdom here.

    last_empty = []
    to_remove = []

    # Moving up name.
    for item in remaining_lineage:
        if item[1] == ' ':
            last_empty = item
            to_remove.append(item)
        else:
            break

    # Removing all the gaps
    for item in remaining_lineage:
        if item[1] == ' ':
            to_remove.append(item)
    remaining_lineage = [x for x in remaining_lineage if x not in to_remove]

    # Associate the name with the last empty rank, and put that rank and name back onto the lineage.

    if len(last_empty) == 0:
        remaining_lineage.insert(0, name) # Strangely enough if not last_empty doesn't work, but this just puts
        # the name back at the beginning if there was nothing to move up
    else:
        last_empty[1] = name[1]
        remaining_lineage.insert(0, last_empty)
        remaining_lineage.insert(0, name)

    return remaining_lineage
```

6.3. Ethics Form

AU Status

Undergraduate or PG Taught

Your aber.ac.uk email address

jak30@aber.ac.uk

Full Name

Jakub Kawka

Please enter the name of the person responsible for reviewing your assessment.

Amanda Clare

Please enter the aber.ac.uk email address of the person responsible for reviewing your assessment

afc@aber.ac.uk

Supervisor or Institute Director of Research Department

cs

Module code (Only enter if you have been asked to do so)

CS39440

Proposed Study Title

MMP - What's in my BLAST results? A bioinformatics analysis tool

Proposed Start Date

02/02

Proposed Completion Date

04/05

Are you conducting a quantitative or qualitative research project?

Mixed Methods

Does your research require external ethical approval under the Health Research Authority?

No

Does your research involve animals?

No

Are you completing this form for your own research?

Yes

Does your research involve human participants?

No

Institute

IMPACS

Please provide a brief summary of your project (150 word max)

One of the most common tasks for bioinformaticians is to take some DNA or protein sequence under investigation and find other sequences that are similar to this sequence, but come from known databases. There are several tools for conducting this sequence similarity search (BLAST, Diamond, USearch, etc). The results they provide are fairly basic - a list of identifiers of sequences that matched, and various scores that tell us how well they matched. We would like a tool that will take these results and add a whole load of further information: from which species/genera/families were the matching sequences? Were the matches whole matches or part matches? What functions did the matching sequences have? What protein domains did they

have? I'll explain all the biology terminology. You would be able to automatically process the data and match it to further online or local databases, providing a text based report, and ideally a web front end visualisation.

Where appropriate, do you have consent for the publication, reproduction or use of any unpublished material?

Not applicable

Will appropriate measures be put in place for the secure and confidential storage of data?

Yes

Does the research pose more than minimal and predictable risk to the researcher?

No

Will you be travelling, as a foreign national, in to any areas that the UK Foreign and Commonwealth Office advise against travel to?

No

Please include any further relevant information for this section here:

If you are to be working alone with vulnerable people or children, you may need a DBS (CRB) check. Tick to confirm that you will ensure you comply with this requirement should you identify that you require one.

Yes

Declaration: Please tick to confirm that you have completed this form to the best of your knowledge and that you will inform your department should the proposal significantly change.

Yes

Please include any further relevant information for this section here:

7. Bibliography

[1] A. G. Levine, "An explosion of bioinformatics careers | Science | AAAS," *Science Magazine*, 13 June 2014. [Online]. Available: <http://www.sciencemag.org/features/2014/06/explosion-bioinformatics-careers>. [Accessed 29 April 2018].

An article discussing the current state of bioinformatics as a field.

[2] Z. Merali, "Computational science: ...Error : Nature News," *Nature*, 13 October 2010. [Online]. Available: <https://www.nature.com/news/2010/101013/full/467775a.html?ref=nf>. [Accessed 29 April 2018].

An article discussing the current state of scientific software.

[3] S. Altschul, W. Gish, W. Miller, E. Myers and D. Lipman, "Basic local alignment search tool.," *J Mol Biol*, 1990.

The original paper for the BLAST algorithm.

[4] P. C. Jan Fassler, "BLAST Glossary - BLAST® Help - NCBI Bookshelf," National Center for Biotechnology Information, 14 July 2011. [Online]. Available: <https://www.ncbi.nlm.nih.gov/books/NBK62051/>. [Accessed 29 April 2018].

A glossary of common BLAST terminology.

[5] D. J. Lipman and W. R. Pearson, "Rapid and Sensitive Protein Similarity Searches," *Science*, 1985.

The original paper for the FASTA algorithm.

[6] C. bio, "Bioinformatics explained: BLAST versus Smith-Waterman," 4 July 2007. [Online]. Available: http://www.montefiore.ulg.ac.be/~kvansteen/GBIO0009-1/ac20092010/Class4/BE-smith-waterman_versus_blast.pdf. [Accessed 30 April 2018].

A paper comparing BLAST and Smith-Waterman algorithms.

[7] "BLAST® Command Line Applications User Manual [Internet]. Bethesda (MD): National Center for Biotechnology Information (US); 2008-. Appendices. 2008 Jun 23 [Updated 2016 May 18]. Available from: <https://www.ncbi.nlm.nih.gov/books/NBK279684/>," [Online].

BLAST documentation showing the myriad of optional parameters.

[8] P. Cock, "Blasted Bioinformatics!?: BLAST XML 2 - does the sequel live up to my hopes?," 8 July 2015. [Online]. Available: <https://blastedbio.blogspot.co.uk/2015/07/blast-xml-2-include-trouble.html>. [Accessed 30 April 2018].

Peter Cock's blog discussing the new XML2 format. Peter Cock is a notable BioPython developer.

[9] "BLAST: Basic Local Alignment Search Tool," [Online]. Available: <https://blast.ncbi.nlm.nih.gov/Blast.cgi>. [Accessed 1 May 2018].

The NCBI Blast Online Tool

[10] biomickwatson, "Call the bioinformatics police! |," 14 January 2013. [Online]. Available: <http://www.opiniomics.org/call-the-bioinformatics-police/>. [Accessed 1 May 2018].

A blog post complaining about the current state of Bioinformatics software.

[11] M., "The 'Horror' that Awaits You, if You Quit Bioinformatics (or Academia)," 17 February 2014. [Online]. Available: <http://homolog.us/blogs/blog/2014/02/17/horror-awaits-quit-bioinformatics-academia/>. [Accessed 1 May 2018].

A blog post complaining about the current state of Bioinformatics as a field.

[12] P. Cock, "Blasted Bioinformatics!?", [Online]. Available: <https://blastedbio.blogspot.co.uk/>. [Accessed 1 May 2018].

Peter Cock's personal BLAST blog. Peter Cock is a notable BioPython developer and has many interesting insights on the subject of BLAST.

[13] J. Chang, B. Chapman, I. Friedberg, T. Hamelryck, M. d. Hoon, P. Cock, T. Antao, E. Talevich and B. Wilczyński, "Biopython Tutorial and Cookbook," 3 April 2018. [Online]. Available: <http://biopython.org/DIST/docs/tutorial/Tutorial.html#htoc94>. [Accessed 3 May 2018].

A link to the BioPython documentation, in this instance used to show Peter Cock making named contributions.

[14] "Galaxy," [Online]. Available: <https://usegalaxy.org/>. [Accessed 1 May 2018].

The main website for the Galaxy web platform.

[15] pgdurand, "GitHub - pgdurand/BlastViewer: This package contains the BLAST Viewer Tool.," [Online]. Available: <https://github.com/pgdurand/BlastViewer>. [Accessed 1 May 2018].

The GitHub repository for BlastViewer.

[16] jaclag, "jamblast download | SourceForge.net," [Online]. Available: <https://sourceforge.net/projects/jamblast/>. [Accessed 1 May 2018].

The SourceForge repository for jamblast.

[17] R. Neumann, S. Kumar, T. Haverkamp and K. Shalchian-Tabrizi, "BLASTGrabber: a bioinformatic tool for visualization, analysis and sequence selection of massive BLAST data. - PubMed - NCBI," 5 May 2014. [Online]. Available: <https://www.ncbi.nlm.nih.gov/pubmed/24885091>. [Accessed 1 May 2018].

The official paper introducing BLASTGrabber.

[18] "Extreme Programming For One," [Online]. Available: <http://wiki.c2.com/?ExtremeProgrammingForOne>. [Accessed 2 May 2018].

The WikiWikiWiki entry discussing the concept of Extreme Programming For One.

[19] M. Christie, "A comparison of bioinformatics programming languages |," 20 November 2012. [Online]. Available: <http://www.molecularrecologist.com/2012/11/a-comparison-of-bioinformatics-programming-languages/>. [Accessed 02 May 2018].

An article comparing common bioinformatics programming languages.

[20] Maria, "For bioinformatics, which language should I learn first? – OMGenomics," 3 March 2017. [Online]. Available: <http://omgenomics.com/programming-languages/>. [Accessed 5 May 2018].

An article discussing which programming language one should learn first, with regards to bioinformatics.

[21] M. Fourment and M. R. Gillings, "A comparison of common programming languages used in bioinformatics," 5 February 2008. [Online]. Available: <https://www.ncbi.nlm.nih.gov/pmc/articles/PMC2267699/>. [Accessed 2 May 2018].

An official paper comparing common programming languages used in bioinformatics.

[22] "TIOBE Index | TIOBE - The Software Quality Company," April 2018. [Online]. Available: <https://www.tiobe.com/tiobe-index/>. [Accessed 2 May 2018].

The TIOBE Programming Community index indicating popularity of programming languages.

[23] A. Coghlan, "Welcome to a Little Book of R for Bioinformatics! — Bioinformatics 0.1 documentation," [Online]. Available: <http://a-little-book-of-r-for-bioinformatics.readthedocs.io/en/latest/>. [Accessed 2 May 2018].

An R-based bioinformatics library.

[24] L. Stein, "How Perl saved human genome - BioPerl," 7 February 2007. [Online]. Available: https://web.archive.org/web/20070202101624/http://www.bioperl.org/wiki/How_Perl_saved_human_genome. [Accessed 2 May 2018].

An article discussing Perl's contributions with regards to the Human Genome Project

[25] J. Leipzig, "What Are The Most Common Stupid Mistakes In Bioinformatics?," 2011. [Online]. Available: <https://www.biostars.org/p/7126/>. [Accessed 02 May 2018].

A forum thread discussing common mistakes made in Bioinformatics

[26] "7.3 Parsing BLAST output - Biopython Tutorial and Cookbook," 3 April 2018. [Online]. Available: <http://biopython.org/DIST/docs/tutorial/Tutorial.html#htoc93>. [Accessed 2 May 2018].

Biopython documentation specifically discussing BLAST results parsing.

[27] "Project Jupyter | Home," [Online]. Available: <http://jupyter.org/>. [Accessed 2 May 2018].

The main website of the Jupyter Notebook framework.

[28] NCBI, "Taxonomy - Site Guide - NCBI," [Online]. Available: <https://www.ncbi.nlm.nih.gov/guide/taxonomy/>. [Accessed 3 May 2018].

The main website of NCBI taxonomy services.

[29] "Index of <ftp://ftp.ncbi.nlm.nih.gov/pub/taxonomy/>," [Online]. Available: <ftp://ftp.ncbi.nlm.nih.gov/pub/taxonomy/>. [Accessed 3 May 2018].

The location of the NCBI taxonomy database dump.

[30] "Levenshtein distance - Rosetta Code," [Online]. Available: http://rosettacode.org/wiki/Levenshtein_distance. [Accessed 2 May 2018].

An explanation of the Levenshtein distance algorithm.

[31] C. S. Perone, "Machine Learning :: Cosine Similarity for Vector Space Models (Part III) | Terra Incognita," 12 September 2013. [Online]. Available: <http://blog.christianperone.com/2013/09/machine-learning-cosine-similarity-for-vector-space-models-part-iii/>. [Accessed 3 May 2018].

A blogpost discussing an implementation of bag-of-words machine learning and cosine vector similarity.

[32] "Biopython · Biopython," [Online]. Available: <http://biopython.org/>. [Accessed 3 May 2018].

Biopython main page.

[33] "scikit-learn: machine learning in Python — scikit-learn 0.19.1 documentation," [Online]. Available: <http://scikit-learn.org/stable/>. [Accessed 3 May 2018].

scikit-learn main page.

[34] "NumPy — NumPy," [Online]. Available: <http://www.numpy.org/>. [Accessed 3 May 2018].

NumPy main page.

[35] "Graphviz - Graph Visualization Software," [Online]. Available: <https://www.graphviz.org/>. [Accessed 3 May 2018].

Graphviz main page.

[36] "NetworkX — NetworkX," [Online]. Available: <https://networkx.github.io/>. [Accessed 3 May 2018].

NetworkX main page.

[37] "The Newick tree format," [Online]. Available: <http://evolution.genetics.washington.edu/phylip/newicktree.html>. [Accessed 3 May 2018].

An explanation of what a Newick tree format is.

[38] "ETE Toolkit - Analysis and Visualization of (phylogenetic) trees," [Online]. Available: <http://etetoolkit.org/>. [Accessed 3 May 2018].

ETE Toolkit main page.