

University of Tübingen

Faculty of Science

Institute for Bioinformatics and Medical Informatics

Master Thesis Bioinformatics

Design and Implementation of a Memory-Efficient, Multithreaded Taxonomic Classifier for Long Reads using Double Indexing

Noel Kubach

May 2025

Reviewers

Prof. Dr. Daniel Huson

(Bioinformatics)

Institute for Bioinformatics and
Medical Informatics

University of Tübingen

Prof. Dr. Nadine Ziemert

(Translational Genome Mining)

Interfaculty Institute of Microbiology and
Infection Medicine

University of Tübingen

Kubach, Noel:

Design and Implementation of a Memory-Efficient, Multi-threaded Taxonomic Classifier for Long Reads Using Double Indexing

Master Thesis Bioinformatics

University of Tübingen

Thesis period: 18.11.2024 – 18.05.2025

Abstract

The more we learn about bacteria, archaea, fungi, and viruses, the more we understand how much we depend on those microbes. Microbes can work for us; they fixate nitrogen for the plants we are eating and help us process the nutrients after we ate. Many drugs that we use to treat diseases are derived from metabolites of bacteria or fungi, or directly produced by them. On the other hand, microbes can just as easily harm or kill us. An unhealthy microbiome has diverse effects on our health, viruses can cause deadly diseases, and the options to fight multidrug-resistant bacteria are rather limited. These examples illustrate how important the study of those invisibly small organisms is for our health and well-being.

A powerful method to study the properties of microbes is to analyze their genomes, the central control unit that regulates their entire metabolism. For a long time, DNA sequencing was only possible for organisms that could be cultivated in the lab. However, with the development of second- and third-generation sequencing techniques, it is now possible to sequence all DNA directly from an environmental sample. The study of this DNA is called metagenomics.

The most fundamental question that can be answered using raw DNA sequencing reads from a metagenomic sample is what organisms it consists of. For this, the Double-Indexed k-mer Taxonomic Classifier DIAMER was developed in this thesis. DIAMER combines the approaches of two famous tools in the field - DIAMOND and Kraken - to taxonomically classify third-generation sequencing reads on the protein level. The double-indexing approach of DIAMOND, together with the lowest common ancestor database of Kraken, enables DIAMER to classify reads on either a laptop or a high-performance computing cluster. DIAMER can use large reference databases for classification, even with little memory available, to keep up with the rapid growth of reference databases.

Zusammenfassung

Je mehr wir über Bakterien, Fungi und Viren lernen, desto mehr verstehen wir das Ausmaß unserer Abhängigkeit von diesen Mikroorganismen. Mikroorganismen können für uns arbeiten; sie fixieren den Stickstoff für die Pflanzen, die wir essen und helfen uns nach dem Essen, die Nährstoffe zu verarbeiten. Ein großer Teil unserer Medikamente basiert auf den Metaboliten von Bakterien oder Fungi bzw. wird direkt von diesen produziert. Andererseits können uns Mikroben genau so einfach schaden oder gar töten. Ein ungesundes Mikrobiom kann unsere Gesundheit beeinflussen, Viren können tödliche Krankheiten auslösen und die Möglichkeiten, multiresistente Bakterien zu bekämpfen, sind sehr eingeschränkt. Diese Beispiele verdeutlichen, wie wichtig die Erforschung dieser winzig kleinen Organismen für unsere Gesundheit und unser Wohlergehen ist.

Eine bewährte Methode, um etwas über die Eigenschaften von Mikroben zu lernen, ist die Analyse ihres Genoms. Lange Zeit mussten Organismen zum Sequenzieren ihrer DNA im Labor gezüchtet werden. Mit der Entwicklung moderner Sequenzierungstechniken, ist es heute allerdings möglich, die DNA einer beliebigen Probe zu sequenzieren. Die Analyse der so gewonnenen DNA wird Metagenomik genannt.

Die erste Frage, die mit der DNA einer metagenomischen Probe beantwortet werden kann, ist, welche Organismen enthalten sind. Dazu wurde in dieser Arbeit der Double-Indexed k-mer Taxonomic Classifier DIAMER entwickelt. DIAMER kombiniert zwei Ansätze der im Feld bewährten Programme DIAMOND und Kraken, um DNA *reads* auf Protein-Level taxonomisch zu klassifiziert. Der *double-indexing*-Ansatz von DIAMOND, zusammen mit einer *lowest common ancestor* Datenbank von Kraken, ermöglichen es *reads* auf einem Laptop zu klassifizieren. Zur Klassifizierung mit DIAMER kann eine große Referenzdatenbank verwendet werden, auch wenn wenig Arbeitsspeicher zur Verfügung steht. Trotz des schnellen Wachstums der Datenbanken, wird so auch in Zukunft die Klassifizierung auf weniger leistungsstarken Rechnern ermöglicht.

Acknowledgments

First of all, I would like to thank Prof. Daniel Huson for providing me with this interesting topic that perfectly matched what I had in mind for my thesis. I am also grateful to Anupam Gautam and Julia Fischer for helping me with the questions I had. Special thanks to my girlfriend Sophie for supporting me during this project and proofreading the thesis. Additionally, I would like to thank Prof. Nadine Ziemert for agreeing to review this thesis.

Contents

1	Introduction	8
1.1	Metagenomics	8
1.1.1	Analysis of Metagenomic Data	9
1.2	DIAMER - A Memory-Efficient Translation-Based Long-Read Classifier .	10
1.3	Thesis Structure	12
2	Background	13
2.1	Methods of Sequence Algorithms	13
2.1.1	K-mer based Methods	13
2.1.2	Spaced Seeds	14
2.1.3	Minimizer	15
2.1.4	Reduced Amino Acid Alphabets	16
2.1.5	Sort-Merge Join & Double Indexing	17
2.2	Abundance Estimation & Read Classification	18
2.2.1	DIAMOND	20
2.2.2	Kraken	21
2.2.3	Long-read Classification Algorithms	22
3	Long-Read Classification with DIAMER	24
3.1	Methods	24
3.1.1	Reference Database & Test Datasets	24
3.1.2	Reduced Amino Acid Alphabets	25
3.1.3	Hardware	26
3.1.4	Use of Third-Party Software	27
3.2	System Description	27

3.2.1	Reading the Taxonomy	28
3.2.2	Preprocessing	29
3.2.3	DIAMER Index	30
3.2.4	Reduced Amino Acid Alphabets & Unknown Characters	31
3.2.5	Database Indexing	32
3.2.6	Read Indexing	33
3.2.7	Read Classification	33
3.2.8	Output Files	34
3.2.9	Suggested Design Improvements	35
3.3	Implementation Details & Challenges	36
3.3.1	Programming principles	36
3.3.2	Reading Sequence with the SequenceSupplier class	37
3.3.3	Challenges in Text File Reading	37
3.3.4	k-mer Encoding, Extraction & Filtering	38
3.3.5	Bucket-wise k-mer Collection	40
3.3.6	Concurrency in DIAMER	44
3.3.7	Delta encoding	45
3.3.8	Code Documentation and Unit Tests	45
3.3.9	Possible Improvements	46
4	Results and Evaluation	47
4.1	Methods for Statistical Analysis	47
4.1.1	Taxon-based Analysis	48
4.1.2	Read-based Analysis	48
4.2	Parameter Search	49
4.2.1	Reference Database	49
4.2.2	Classification Algorithms & Thresholds	50
4.2.3	K-mer Length & Spaced Seeds	53
4.2.4	Reduced Amino Acid Alphabet	55
4.2.5	K-mer Filtering, Minimizers & Index Size	57
4.3	Performance and Application	59
4.3.1	Performance	59
4.3.2	Comparative Analysis	63

4.3.3 Application	65
5 Discussion and Outlook	66
5.1 Limitations	66
5.2 Key Insights	67
5.3 Future Directions	68
6 Availability	70
References	78
A Supplementary Information	79
A.1 Use of AI Methods	79
A.2 Supplementary Figures	80
Erklärung	84

Chapter 1

Introduction

1.1 Metagenomics

This chapter will summarize the origin, progress, and goal of metagenomic research to provide context for possible applications of DIAMER.

Microbes affect our everyday life. Whether in the form of the vital symbiotic bacteria that we carry around in and on us every day [1], the bacteria that fixate the nitrogen for the food we are eating [2], infections caused by microbes, or the bacteria-produced medicine we use to cure the diseases again [3]. Especially the relevance for medicine makes studying microbes, their interaction with us, each other, and the environment an important field of research. Thus, analyzing the genomes of these microorganisms is a fundamental and comprehensive way to better understand how they can hurt or help us.

Early studies that attempted to decipher the genomes of the first phages, bacteria, and archaea in the 1970s were limited to organisms with small genomes that could be cultivated in the lab [4]. Even though studying the taxonomic relationships of uncultured bacteria became possible via 16s sequencing in the 1980s [5, 6], the genomes of the unculturable majority of bacteria remained undiscovered. The culturability of bacteria introduced a bias into available reference sequence databases, which still exists today [7]. The analysis of the sequences of single strains of bacteria isolated from a complex microbiome in a specific environment allows us to study most metabolic processes happening within one cell. However, it can at best permit a glimpse at the manifold interactions between the members of a whole microbiome and between the microbiome and its environment.

In the 2000s, the decreasing sequencing costs and the development of second-generation sequencing made it economically feasible to advance from only the investigation of amplicon sequences to shotgun sequencing of all DNA that can be extracted from a microbiome [8, 9, 4]. The DNA sequences of a collective of organisms were termed "metagenome" [10], and the study of it was soon established as "metagenomics". However, the analysis of the data provided by this new opportunity posed a challenge to the existing computational tools available for sequence analysis [4, 9]. Metagenomic shotgun sequencing reads are a highly fractured sample of a usually unknown number of organisms. The amount of sequence data generated by one experiment is extremely high, and still, one cannot be sure that the genome of every organism is covered by the reads. A more recent advance in metagenomics is the use of third-generation sequencing methods (long-read sequencing) [11, 12]. While the much longer reads of partially more than 100 kb, compared to a few hundred bases for second-generation sequencing, reduce the fragmentation of the sample DNA, the higher error rate introduces additional problems to the analysis.

1.1.1 Analysis of Metagenomic Data

Despite the additional challenges in analyzing metagenomic shotgun sequencing reads compared to reads from an isolated organism in the lab, if interpreted successfully, metagenomics can help answer many interesting questions about a microbiome [8]. The most fundamental question is what organisms are present in the sample. While taxonomic abundance estimation in a microbiome was already possible before metagenomic shotgun sequencing via 16s analysis. While this might still be beneficial in some cases, the taxonomic classification of shotgun reads or assembled contigs provides a much higher resolution and does not suffer from biases that can be introduced by 16s-based methods [13, 14]. Similar to 16s analysis, one strategy used in bioinformatics tools to determine the abundance of organisms from shotgun data is the search for specific marker genes in the input data (marker-based approach). The output of these *profilers* is an estimate of the taxonomic composition of a sample [15]. A second approach is not limited to marker genes, but compares all sequence data from the input sequences to a database with reference sequences via k-mers, the Burrows–Wheeler transform, or alignments. These *classifiers* output an assignment of as many input sequences as possible to nodes of a taxonomic tree, which can be converted to abundance estimations if necessary. Due to

the low frequencies of marker genes, profilers usually only classify a small percentage of input sequences, while the goal for classifiers is to assign as many reads as possible. Further details on different profilers and classifiers are described in Chapter 2.2.

Another aspect that can be analyzed with a metagenome is how the collection of organisms cooperate and interact [8]. What is the purpose of each single organism in the context of the community? And how do the members interact with each other and with the environment? This fundamentally new type of analysis could not have been done based on 16s rDNA or the genomes of lab-grown bacteria alone. Answering these questions now relies on the functional annotation of the genes found in the DNA sequences. There are three main approaches for analyzing metagenomic data: alignment-based, machine learning-based, and marker gene-based. Sequence alignment is the most common method to annotate their functions based on the similarity to known genes in reference databases. The alignment of metagenomic DNA to reference databases can be done on the level of raw reads or contigs after assembly. Additionally, the sequences can be aligned on the DNA level or the level of translated proteins. A well-established tool for integrating the analysis of taxonomic diversity with various annotations about its functional capacity based on sequence alignments is MEGAN [11]. Other tools that can be used for further analysis of contigs are antiSMASH[16] and ARTS[17], which analyze the biosynthetic potential of the secondary metabolism to find potential antibiotics.

Finally, metagenomes in combination with metadata about the sampling site and conditions can be used for comparative analyses [18, 19]. The comparison of samples across time and space allows us to uncover the mechanics behind microbial ecosystems. In summary, metagenomics is a very diverse field of research with numerous applications in medicine and biology-related fields.

1.2 DIAMER - A Memory-Efficient Translation-Based Long-Read Classifier

One strategy to measure the abundance of different taxa in metagenomic shotgun sequencing data is to assign every read to a taxon and estimate relative abundance based on the number of reads assigned to each taxon [9]. Numerous tools have been developed for this task and follow different approaches to compare each individual read to

a database with reference sequences at either DNA or protein level. All read classifiers have in common that they require a lot of memory during execution to hold the index of the reference databases ([20]). In its translational search mode, the widely used read classifier Kraken 2 requires about 139 GB of RAM for the NCBI non-redundant Protein BLAST database (NR). The high memory demand already excludes the use of hardware other than high-performance computing clusters today, and given the rapid growth of sequence databases, this problem will get even worse in the future [21].

The Double-Indexed k-mer Taxonomic Classifier DIAMER developed in this project provides a solution for the high memory demand of classification tools. DIAMER can generate an index of a large protein reference database, such as NR, on a high-performance server. The generated index file can be used for the classification of 1M metagenomic sequencing reads in under 20 minutes on any platform that supports Java with a fast solid-state drive (SSD), reasonable computing power (e.g, 16 Threads at ~ 4.5 GHz), and 16 GB of RAM. The results have a recall comparable to Kraken 2 and generally better precision. The approach of DIAMER makes use of the fast m.2 SSDs that are part of most modern mainstream laptops (500 - 1,000 €). The precalculated index of the reference database contains a list of k-mer-taxonomic ID tuples that are constructed with the k-mer-lowest common ancestor database approach of Kraken [22]. The list is split into buckets and sorted according to the approach used in DIAMOND [23]. A second index with sorted k-mer-read ID pairs of the long-read dataset can be generated with the resources provided by a laptop or PC and compared to the database index. With the double index approach of DIAMOND, both lists can be traversed in parallel to find matching k-mers and classify the input sequences. The strategy to split the indexes into buckets and the linear data access required while traversing both indexes allows for using the full available read speed of the drive, even with low RAM and computing power. DIAMER shifts the memory requirements for a large reference database from the limited and expensive RAM to the much larger, cheaper, and increasingly fast SSDs. Through this approach, DIAMER will keep up with the rapid growth of reference databases without depending on more and more RAM.

1.3 Thesis Structure

This thesis is divided into four chapters: [Background](#), [Long-Read Classification with DIAMER](#), [Results and Evaluation](#), and [Discussion and Outlook](#). The Chapter [Background](#) describes all theoretical concepts and strategies used in DIAMER. Additionally, the chapter gives an overview of the tools available for sequence classification and abundance estimation, and the techniques used in DIAMOND and Kraken. All details related to the implementation of DIAMER and the methods used are explained in the Chapter [Long-Read Classification with DIAMER](#). The Section [System Description](#) gives a broad overview of the structure and function of DIAMER. The following Section [Implementation Details & Challenges](#) describes the most important and unique features and algorithms on the level of its implementation. In the Chapter [Results and Evaluation](#), the capabilities of DIAMER are explored and evaluated while testing different input parameters. Additionally, the results generated by DIAMER are compared to the results of Kraken 2X. Finally, the limitations, key insights, and possible future development of this Tool are discussed in Chapter [Discussion and Outlook](#).

Chapter 2

Background

2.1 Methods of Sequence Algorithms

This section describes important methods and strategies that are frequently used in bioinformatics algorithms for sequence analysis and find application in DIAMER.

2.1.1 K-mer based Methods

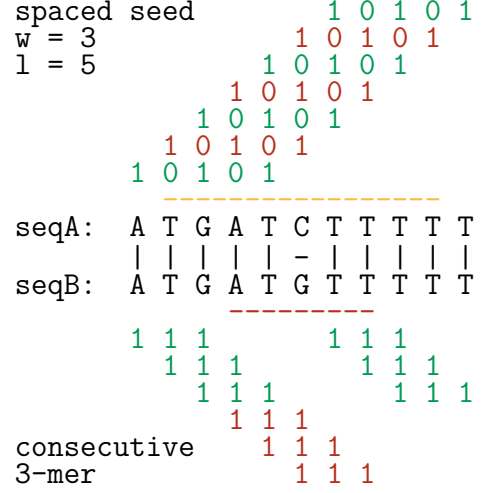
The transformation of a sequence into all subsequences of length k (k-mers) is an essential tool in bioinformatics. The fixed size of a k-mer makes it the perfect tool for computational methods. Especially in applications that require the comparison of multiple sequences, comparing exact k-mer matches rather than computing alignments of whole sequences is much easier. Thus, k-mers can be found in most bioinformatics software that deals with sequences. Most commonly, k-mers are used as a preprocessing step to find possible locations for a local sequence alignment in a *seed-and-extend* approach, as, for example, in the homology search and alignment tools BLAST [24], DIAMOND [23] or MMseqs2 [25]. On the other hand, some tools use k-mers without a subsequent alignment step, as the taxonomic classifiers Kraken 2 [26] or the sequence distance estimator Mash [27].

Formally, a k-mer can be defined as a substring $S_{i,j} = s_i, \dots, s_j$ of the sequence $S = s_1, \dots, s_n$ with the length of the k-mer $k = |S_{i,j}|$ and the length of the sequence $|S| = n$. The multiset T of all k-mers (including duplicates) of S has size $|T| = n - k + 1$ and if all characters in S come from an alphabet Σ with $s_{l \in \{1 \dots n\}} \in \Sigma$, the number of distinct k-mers in the set of all k-mers U (without duplicates) is limited by $|U| \leq |\Sigma|^k$ for any n .

2.1.2 Spaced Seeds

When using k-mers to compare sequences or perform database searches, one is faced with a trade-off between sensitivity and runtime that depends on the length of the k-mer. Longer k-mers decrease the probability of random matches between unrelated sequences and lower the computational effort to process all k-mer hits in possible post-processing steps. In comparison, shorter k-mers permit hits between more distantly related sequences, but simultaneously increase the number of random matches and the computational load on subsequent processing steps. With the release of the homology search tool PatternHunter in 2002 [28], the concept of *spaced seeds* as opposed to *consecutive seeds* in the seeding phase of k-mer based homology search for DNA was introduced. In contrast to a k-mer, where k consecutive characters of two sequences are compared, a spaced seed with *weight* w compares w characters within a window of l characters [28]. The relative positions of the characters that are compared to each other are defined in the *shape* (or *model*) of the seed, which is commonly specified by a string consisting of 1 and 0, where 1 marks the positions compared and 0 the spaces [28, 29]. The creators of PatternHunter showed that the use of spaced seeds of weight w increases the sensitivity and decreases the hit probability in homology searches compared to the use of a traditional k-mer of length $k = w$ [28]. Both properties help in homology search: The decreased hit probability leads to fewer hits in post-processing, and the increased sensitivity results in detecting more distantly related homologs. The decreased hit probability can be explained by the larger window size occupied by a spaced seed with weight w , compared to a k-mer without spaces [28]. Accordingly, a k-mer has $l - k$ more possible shifts of its window compared to the window of a spaced seed of length l and weight w . Consequently, more k-mers than spaced seeds exist for a sequence of the same length, which raises the hit probability for k-mers compared to spaced seeds. The higher sensitivity of spaced seeds can be explained with a simple example: If two sequences only differ in one position s_i somewhere in the middle of the sequence, the pairwise comparison of all k k-mers in the region $s_{i-(k-1)}, \dots, s_{i+(k-1)}$ will fail due to the difference in one character and subsequently, the equality of the remaining $2k - 2$ characters in that window will not be discovered. On the other hand, with a spaced seed of weight $w = k$, length $l > k$ and an arbitrary shape, the comparison of some of the characters in the window $s_{i-(k-1)}, \dots, s_{i+(k-1)}$ is still possible for all shift positions where a space masks the non-matching position (see Figure 2.1).

Figure 2.1: Example of an alignment with spaced seeds and consecutive k-mers. One mismatch between seqA and seqB causes all 3-mers (shown below the alignment) in a region around the mismatch (indicated with red dashes) to be different. The exact matches of spaced seeds with weight 3 and window length 5 are also disrupted around the mismatch (indicated with orange dashes), but the mismatches of the seeds are distributed over a larger region, and the similarity between nucleotides close to the mismatch can still be discovered.



The number of seeds that fail to match is k for the contiguous and the spaced seed. However, in the case of the spaced seed, the non-matching seeds are distributed over the larger region $s_{i-(l-1)}, \dots, s_{i+(l-1)}$ and thus the similarity between characters around s_i can still be discovered [28].

2.1.3 Minimizer

Minimizers can be used to sub-sample the k-mers extracted from a sequence. Roberts et al. first introduced the minimizer concept in 2004 to reduce the storage requirements of all k-mers of the ever-growing sequence databases [30]. The authors defined a local rule to decide which of m consecutive k-mers spanning a region of size $o = m + (k - 1)$ to keep. The retained k-mers are called minimizer of the region $o = m + (k - 1)$ and are defined as the *smallest* of all m k-mers according to an underlying *ordering*. Keeping at least one of m consecutive k-mers guarantees sequences with a matching subsequence of size o to have at least one minimizer in common. The ordering can be defined arbitrarily, but must be the same for all compared sequences. Roberts et al. suggest using an ordering that favors rare k-mers to become minimizers, since this leads to a higher statistical significance for a minimizer hit. In the case $m \leq k$, there are no spaces between adjacent minimizers. However, parts in the beginning and the end of a sequence can lie outside of any minimizer. Therefore, the concept of *end-minimizers* was developed, which adds additional k-mers at the beginning and end of a sequence to the set of minimizers [30].

2.1.4 Reduced Amino Acid Alphabets

The very nature of amino acids suggests grouping them by their physicochemical properties. Possible grouping criteria are hydrophobicity, aromaticity, size, or charge. All of those properties are important for how multiple amino acid side chains interact and lead to the complex three-dimensional structure of proteins. The similarity of some amino acid side chains is also visible in the secondary and tertiary structures of proteins. After the first protein structures were published in the second half of the 20th century, it became evident that the structural diversity of proteins is much more limited than the sequence diversity, since many different sequences can yield the same protein fold [31]. Driven by these observations, people started to investigate whether all 20 amino acids are needed to form nature-like structures, or if a subset of amino acids is sufficient for complex interactions. Experiments on different proteins showed that most amino acids can be replaced by only a few representative amino acids, while still retaining the fold of the more diverse original sequences, as reviewed by Peter Wolynes [32] and Hue Sun Chan [33]. Complementing those experimental efforts, theoretical models for the clustering of proteins were developed based on emerging sequence and structure databases. In 1996, Paul Thomas et al. developed a reduced amino acid alphabet from a statistical potential derived from PDB structures and successfully used a 10-letter alphabet for gapless threading (sequence-structure alignment) [34]. In 2000, Murphy et al. clustered the amino acids based on the BLOSUM50 substitution matrix and proved that 10 to 12 amino acid clusters are enough to predict the fold class of proteins [35]. Using a reduced amino acid alphabet lowers the complexity of protein sequences while adding information about the similarity of amino acids within one group at the same time. This not only simplifies the computational complexity of tasks but can also increase the specificity and sensitivity, as shown in a study on fold recognition [36]. In 2022, Liang et al. counted 672 different reduced amino acid alphabets, which have been used for all kinds of problems, mainly for structure and function prediction of proteins, but also for sequence alignment tasks [37]. The two prominent sequence alignment tools that are built around reduced amino acid alphabets are RAPSearch [38] or RAPSearch2 [39] and DIAMOND [23]. RAPSearch and the more memory-efficient RAPSearch2 provide an alternative to protein BLAST and BLASTX utilizing a reduced amino acid alphabet of 10 amino acids and suffix arrays (or a hash table in the case of RAPSearch2) to speed up the seeding

phase of the BLAST algorithm, which results in a 20 to 90 fold speedup compared to BLAST [38, 39]. DIAMOND uses an 11-letter reduced amino acid alphabet for protein-protein or DNA-protein database search and alignment to achieve a speedup of up to 20.000 times the speed of BLAST [23]. In both cases, reducing the amino acid alphabet allows specific "mismatches" in the seeding phase that are encoded in the reduced amino acid alphabet and based on the observations of which amino acids can be replaced by which other amino acids without changing the structure and function.

2.1.5 Sort-Merge Join & Double Indexing

Sort-merge join is an algorithm used for the JOIN operation in relational database management systems. For illustration, consider a database close to the structure of DIAMER with two tables, A and B. Table A contains two columns: the column "k-mer" that contains all k-mers that occur in a database of protein sequences, and the column "tax_id" that contains the taxonomic ID of the organism the k-mer was extracted from. Table B contains similar data from a set of sequencing reads. Here, the column "k-mer" harbours all k-mers that occur in the translation of the sequencing reads, and the column "read_id" contains the ID of the corresponding reads. Since both tables contain the column "k-mer", a JOIN operation of tables A and B on the extracted protein k-mers would result in a table with the columns "tax_id", "k-mer", and "read_id" where each taxonomic ID is associated with a read ID if they share the same k-mer. Internally, the JOIN operation can be realized with the merge-sort join algorithm. The idea is to first sort both tables by their k-mer column in lexicographic order and, in a second step, traverse both sorted tables simultaneously and return joined rows where tables A and B share the same k-mer. In the case of *double indexing*, both tables are already indexed on the k-mer column. Thus, the sorting step can be skipped, and the index order can be used directly for the JOIN operation. The sort-merge join algorithm is used in this project on the pre-sorted lists of k-mers and IDs of a protein database and a dataset of long-reads to identify matching k-mers.

2.2 Abundance Estimation & Read Classification

The most fundamental way to study a microbiome is to examine the composition of organisms it consists of. Traditionally, the genes of the ribosomal RNA, such as the 16s rDNA, have been used to identify organisms for almost 50 years now [40]. The combination of conserved and more variable regions in the rDNA made it the perfect tool for identifying organisms in environmental samples [5]. The advantage of taxonomic classifications with rRNA genes is that only these genes must be amplified and analyzed. However, with decreasing costs for shotgun sequencing, taxonomic assessment of a microbiome no longer relies on rDNA. Metagenomic data can serve as a more comprehensive starting point for taxonomic abundance estimation [8, 9, 13].

All tools available for taxonomic characterization of metagenomic datasets rely on a database with either DNA or protein reference sequences and an underlying taxonomy. Popular choices are the NCBI taxonomy with the RefSeq or the BLAST non-redundant (NR/NT) databases [41], the GTDB taxonomy and database [42], or marker gene-specific databases. The second input for the tools is raw reads or contigs, the choice of which can influence the quality of the results [13]. The output can be roughly divided into two groups: taxonomic sequence classification (or binning) and taxonomic profiling [15]. Accordingly, a tool can be called *classifier* if it classifies each input sequence (raw read or contig) to a specific taxon and *profiler* if it only outputs an estimation of the abundance of different taxa, without trying to classify each sequence individually. Importantly, the output of a classifier can be converted into an abundance estimation by comparing the relative number of sequences assigned to each taxon. However, aspects such as different genome sizes and input sequence lengths have to be considered in the calculation. Furthermore, the taxa reported by profilers or classifiers are not necessarily tied to one taxonomic rank, like species or genus only, but it makes sense to also include higher ranks, depending on the conservation of the sequences. Another classification of the available tools can be based on the methods used. This results in one category of **marker-based** methods for all taxonomic profilers and three categories for the classifiers: **alignment-based**, based on **exact matches** via k-mers or the Burrows–Wheeler transform (BWT), or **machine learning-based**.

The **marker-based** profilers rely on identifying either marker genes or gene-independent marker regions on the input sequences. This approach is similar to the established use of

the 16s genes for taxonomic analysis. However, since not only one gene/region is used, marker-based profiling is a more general approach and can provide higher resolution [13, 14]. MetaPhlAn, for example, uses nucleotide BLAST search against its database of clade-specific marker genes for profiling [43]. Since the selection of marker genes is only about 4% of microbial genes, this is much faster than BLASTing against all available sequences. Similarly, the mOTU profiler relies on 10 universal, single-copy marker genes identified by one Hidden Markov model per gene [44]. The tool GOTTCHA, on the other hand, uses the BWT of gene-independent, unique regions of length 30 of the reference genomes to find exact matches to the reads and calculate taxonomic abundance [45].

The first, **alignment-based** category of sequence classifiers relies on the alignment of the input sequences to a reference database. Based on these alignments, tools such as MEGAN [11] can be used as a classifier to assign a read to the lowest common ancestor (LCA) of the organisms that received an alignment. While programs of the BLAST family can be used for the alignment part of this pipeline, they are not designed for these large amounts of data [8]. A new set of alignment tools was developed to keep up with the growing reference databases and metagenomic datasets. RAPSearch and RAPSearch 2 [39] were developed as a fast alternative for short read alignment to a protein database. With a hash table that stores 10-mers in a reduced amino acid alphabet, it is up to 90 times faster than BLASTX. DIAMOND [23] also uses a reduced alphabet with spaced seeds and double indexing to achieve a speed-up of up to 20,000 compared to BLASTX. Compared to other classifiers and profilers, the alignments generated by these tools can be used not only for sequence classification but also for gene annotation and functional analysis. However, for read classification alone, explicit alignments are not required and slow down alignment-based pipelines compared to other approaches.

The group of classifiers based on **exact matches** uses either k-mers or the BWT to find exact sequence matches between input and reference sequences. Similar to alignment-based methods, input sequences might have more than one hit in the reference database, which makes an algorithm necessary that comes up with a final assignment. Kraken, its successor Kraken 2 [26], and CLARK [46] are examples of k-mer based classifiers. Kraken 2 uses a hashmap of all minimizers of either a DNA or a protein reference database to classify reads, while CLARK uses only unique, rank-specific DNA k-mers. The classifier Kaiju employs the BWT of a reference protein database to find exact matches to the

translation of input sequences [47].

Another approach for classifying sequences is **machine learning-based**. The sequence classifier NBC, for example, uses Naïve Bayes Classification to assign a read to a specific taxonomic node [48]. Furthermore, analyzing the members of a metagenomic sample does not necessarily depend on existing reference genomes. Another completely different approach is *taxonomy-independent* binning. Features such as the sequence itself or the coverage can be used to cluster reads or contigs with various clustering algorithms as reviewed by Sedlar et al. [49]. The resulting clusters of sequences are not limited by or dependent on known taxa.[26]

Since DIAMER applies many methods and strategies from DIAMOND and Kraken 2, those tools will be described in more detail in the next chapters.

2.2.1 DIAMOND

DIAMOND is a sequence alignment tool developed to replace BLASTX for the large datasets produced by shotgun sequencing of environmental samples [23]. It is up to 20,000 times faster than BLASTX, while still keeping a comparable sensitivity, making translated alignment feasible for huge datasets and databases. This chapter summarizes the methods used to achieve this speedup.

Similar to the alignment tools of the BLAST family, DIAMOND is based on a seed-and-extend approach and extends exact seed matches to local alignments. The seeds in DIAMOND consist of four different, spaced seed shapes in a reduced amino acid alphabet with 11 amino acids. To compare all seeds of a reference database with the seeds of the reads, the tool uses a double-indexing strategy. All seed-location pairs extracted either in the database generation step or the read assignment step are clustered by a hash on the seed into 1024 partitions (*buckets*). Each bucket is sorted lexicographically by a compressed version of the seed. Distributing all seeds over 1024 buckets enables easy parallelization of all computational steps. Furthermore, in low memory environments, the seeds can be extracted one bucket at a time. Thus, only enough memory for one bucket must be available. For identifying matching seeds, pairs of buckets from the database and the read index can be loaded and compared sequentially. Since both are already sorted, the linear data access efficiently uses the memory structure of modern computers. In other approaches where only the database is indexed, as in Kraken 2, which uses a hash

table for the database index, the seeds are extracted and queried against the database in the order they appear in the reads. This leads to a suboptimal, more or less random memory access. For seed hits, DIAMOND initiates banded Smith-Waterman alignments, depending on calculations that avoid the computation of an alignment twice when run in parallel. In summary, DIAMOND splits spaced seeds of a reduced alphabet into multiple sorted index buckets, which can be processed efficiently in parallel, and the number of parallel index generations can be adjusted according to the available memory.

2.2.2 Kraken

Kraken is a tool for the taxonomic classification of shotgun sequencing reads. Its first version was published in 2014 [22] to provide a fast alternative to alignment or marker-based read classification techniques. The updated, more memory-efficient Kraken 2 was released in 2019 [26]. This chapter summarizes the methods used to achieve a high classification speed.

The core of Kraken is a database that maps k-mers to the lowest common ancestor (LCA) of all organisms in which the k-mer occurs. To classify a read, Kraken first finds all exact k-mer matches between the k-mers of a read and the database and extracts a pruned subtree from the taxonomy containing all taxons associated with the discovered k-mers. The nodes of the subtree are weighted by the k-mer hits, and finally, the read is assigned to the leaf of the highest-scoring root-to-leaf path (details in Section 2.2.3). In Kraken 1, the database that contains the k-mer LCA pairs is divided into regions, each containing k-mers with one common minimizer that is extracted from the k-mer itself. These regions are sorted and can be loaded into the CPU cache during a k-mer query. K-mers that are extracted consecutively from a read are likely to share a minimizer and consequently, the database region in cache can be reused multiple times to save loading time. In Kraken 2, the database was replaced by a probabilistic, compact hash table, now containing minimizers instead of k-mers. The use of minimizers and the hash table reduces the memory requirements of the Kraken 2 database by 85% compared to Kraken 1. To enable read assignment based on a protein database, a new translation search mode (Kraken 2X) with a reduced amino acid alphabet of 16 amino acids was introduced. Additionally, Kraken 2 introduced an option to use very basic spaced seeds instead of consecutive k-mers, as this was found to increase sensitivity in some scenarios

[50]. Overall, the Kraken tool was proven to be a sensitive and very fast read classifier compared to other alignment or marker gene-based tools [51, 52].

2.2.3 Long-read Classification Algorithms

To get from alignments of reads or k-mer hits against a reference database to a taxonomic label that can be assigned to a read, an algorithm is needed, since either method usually yields hits with more than one taxon [9, 22]. In the master thesis of Julia Fisher, titled "A Novel K-mer Algorithmic Approach for Taxonomic Classification of Metagenomic Long Reads", different algorithms for taxonomic assignment of long reads based on k-mer matches to a protein database are explored [53]. These algorithms are similar to the assignment algorithms used in MEGAN [9] and Kraken [22], but focus on being more fine-tunable concerning the assignment depth, which helps with finding thresholds to balance sensitivity and false positive rate [53]. This chapter will summarize the *PushDown* algorithm from Fischer's thesis, since it is based solely on k-mer hit counts between reads and taxa and finds application in DIAMER. Algorithms that are based on reference sequence or read coverage cannot be used in DIAMER, since this information is lost during indexing. The *PushDown* algorithm operates on a pruned subtree of the taxonomic tree, with each node being weighted by the cumulative k-mer hit count of the read that is classified. Starting at the root, the algorithm walks down the tree until it finds the node that is assigned to the read. At each node, three cases can occur: (1) The node is a leaf and the read is classified by its taxon, (2) the node only has one child and the algorithm proceeds to that child or (3) the node has more than one child and a decision has to be made, whether to proceed to the child with the highest weight c_{max} or to stop and assign the current node. Fischer describes two strategies to come to a decision: Compare the weight $w(c_{max})$ of c_{max} to the sum of weights $\sum w(c_i)$ of all other children in a one-vs-all (OVA) approach, or compare it only to the weight $w(c_{2nd})$ of the node c_{2nd} with the second highest weight one-vs-one (OVO). For the weight comparison, a threshold x can be defined that allows the *PushDown* algorithm to proceed to c_{max} if its weight is larger than x times the weight to compare to. This results in the following rules for continuing to c_{max} : OVA $w(c_{max}) > x \sum w(c_i)$ and OVO $w(c_{max}) > x \cdot w(c_{2nd})$. To restrict the threshold values to values between 0 and 1, the threshold t used in DIAMER has been

defined as the reciprocal of x , which leads to these rules:

$$\begin{aligned}
\text{OVA:} \quad & t \cdot w(c_{max}) > \sum w(c_i) \\
\text{OVO:} \quad & t \cdot w(c_{max}) > w(c_{2nd})
\end{aligned} \tag{2.1}$$

For t close to zero, all classifications that encounter case three will stop at the first node with multiple children in the pruned subtree. For $t = 1$ the algorithm proceeds down the tree until a leaf or the case $w(c_{max}) = \sum w(c_i)$ or $w(c_{max}) = w(c_{2nd})$, respectively, is reached. The OVA algorithm with t close to zero, the classification is similar to the algorithm used in the first version of MEGAN for read classification [9]. The OVO setting with $t = 1$ is equivalent to the classification algorithm of Kraken [22].

Chapter 3

Long-Read Classification with DIAMER

The **Double-Indexed k-mer Taxonomic Classifier** DIAMER is a command-line program for the taxonomic classification of long reads written in Java. It uses a protein reference database and a long-read dataset to find matching k-mers, spaced seeds, or minimizers in a customizable reduced amino acid alphabet. The identified sequence matches are used to assign the input sequences to a taxonomic node based on two assignment algorithms with customizable thresholds. This chapter describes the methods that were used in the development process, the ideas behind all tasks that can be performed by DIAMER, and the implementational details of unique features.

3.1 Methods

This section describes all hardware, software, and data resources used in this project.

3.1.1 Reference Database & Test Datasets

Both the NCBI non-redundant Protein BLAST database (NR) (~700M sequences) and the associated NCBI taxonomy [41] were used for this project. The clustered versions NR90 (~240M sequences) and NR50 (~50M sequences) which are clustered at 90% and 50% sequence identity, respectively, and are provided as part of the beta version of MEGAN7¹ [11], were used for testing on smaller scales.

¹<https://software-ab.cs.uni-tuebingen.de/download/megan7>

Two long-read sequencing datasets of microbial mock communities were used to test the assignment capabilities of DIAMER. The first dataset² consists of 1.16M reads and was sequenced in 2020 on an ONT GridION with R10.3 chemistry [54]. Basecalling was performed with Guppy. The second dataset³ was sequenced this year on an ONT PrometION and the bases were called with Dorado and the most recent base call model dna_r10.4.1-e8.2-400bps-sup@v5.0.0 [55].

Name	Sample	Device	Flowcell	Basecaller	# of Reads
ZymoMock	D6300	GridION	R10.3	Guppy 3.2.4	1.16M
ZymoOral	D6332	PromethION	R10.4.1	Dorado 0.8	1.14M

Table 3.1: Long-read datasets used for testing.

3.1.2 Reduced Amino Acid Alphabets

For the development of DIAMER, different reduced amino acid alphabets were tested. The amino acid clusters used will be described here.

Extended Amino Acid Alphabet

Many sequence databases not only support the 20 standard amino acids, but instead an extended amino acid alphabet that uses all 26 letters of the alphabet. All alphabets used in this project were either created directly with all 26 letters or the clusters of the alphabets from other sources were extended with the missing letters. The logic behind assigning the additional 6 letters to clusters of existing alphabets is shown in Table 3.2. All alphabets that have been used for testing are listed in Table 3.3.

Letter	Meaning	Assignment
B	D or N	to D or N
J	L or I	to L or I
O	pyrrolysine	to K
U	selenocystein	to C
X	unknown amino acid	to N, since it is the most common
Z	E or Q	to E or Q

Table 3.2: Additional amino acids of the extended amino acid code and their assignment to the standard amino acids for the reduced amino acid alphabets in this project.

²<https://lomanlab.github.io/mockcommunity/r10.html>

³<https://www.ebi.ac.uk/ena/browser/view/ERX13528856>

Construction of Uniform Reduced Amino Acid Alphabets

While testing DIAMER with different reduced amino acid alphabets, some *uniform* alphabets were constructed. The clustering in these alphabets does not follow any biological patterns on purpose. The only goal during construction was to cluster the amino acids in a way that all clusters have about the same likelihood to occur in an amino acid sequence, based on the amino acid frequency observed in NR.

Name	Clusters
Solis11[56]	[ILMVJ] [NPQSXB] [DEZ] [A] [HR] [G] [T] [KO] [WY] [F] [CU]
Solis15[56]	[ILJ] [NQSXB] [DEZ] [A] [MV] [G] [T] [R] [P] [KO] [F] [Y] [H] [W] [CU]
Solis15S[26]	[ILJ] [NQSXB] [DEZ] [A] [MV] [G] [T] [R] [P] [KO] [F] [Y] [H] [W] [CU] [*]
Solis16[56]	[ILJ] [NQSXB] [DEZ] [A] [M] [V] [G] [T] [R] [P] [KO] [F] [Y] [H] [W] [CU]
Etchebest11[57]	[G] [P] [IV] [FYW] [A] [LMJ] [EQRKZOX] [NDB] [HS] [T] [CU]
DIAMOND[23]	[BDEKNOQRXZ] [AST] [IJLV] [G] [P] [F] [Y] [CU] [H] [M] [W]
Uniform11	[L] [A] [GC] [VWUBIZO] [SH] [EMX] [TY] [RQ] [DN] [IF] [PK]
Uniform11S	[L] [A] [GC] [VWUBIZO*] [SH] [EMX] [TY] [RQ] [DN] [IF] [PK]
Uniform16	[L] [A] [G] [V] [S] [E] [T] [R] [D] [I] [PUBJZO] [KX] [FC] [NW] [QH] [YM]

Table 3.3: Table of all reduced amino acid alphabets used to test DIAMER. The alphabet Solis15s is the one that is used by Kraken 2X. The alphabets without a source behind the name were constructed in this thesis.

3.1.3 Hardware

For the development and testing of DIAMER, mainly two different sets of hardware were used. The platform, which will be called ***PC*** in subsequent chapters, consists of a Windows 11 system, running on an AMD Ryzen 7 5700 G (8 cores, 16 threads, up to 4.6 GHz) with 32 GB of DDR4 RAM (2400 MHz). Additionally, the ***server*** runs on Ubuntu 22.04.5 with two AMD EPYC 9654 processors (192 cores, 384 threads, up to 3,7 GHz) and 2.3 TB of memory. Additional devices were used for some tests during development, as, for example a laptop with specifications similar to the *PC* but only 16 GB of RAM. All plots in this thesis are based on data from the *server* only. *Italic* font will be used from here on to indicate the use of the two described platforms *PC* and *server*.

3.1.4 Use of Third-Party Software

The Java source code for this project was written in JetBrains IntelliJ IDEA 2024.3 Ultimate Edition. The GitHub Copilot plugin (Version 243) was used for AI assisted code completion with the setting to block "suggestions matching public code". The code completion speeds up the coding enormously and is especially useful for repetitive tasks, where the writing patterns can be easily picked up. For monitoring the performance of DIAMER within the Java Virtual Machine on the *server* and locally, VisualVM version 2.1.10 was used. Furthermore, DIAMER depends on the following Java libraries: JetBrains annotation package (org.jetbrains.annotations version 26.0.2), the Junit package for tests (junit version 4.13.2), the CLI package for command line input (commons-cli, version 1.9.0) and the sqlite-jdbc package (org.xerial.sqlite-jdbc, version 3.47.1) for SQLite drivers. The data plots in this thesis were generated in Jupyter Notebooks with Python 3.10 and the packages JupyterLab 4.3.4, Matplotlib 3.10.0, Numpy 2.2.1, Pandas 2.2.3, Scikit-learn 1.6.1 and their dependencies. FastQC⁴ was used to analyze the test datasets. This thesis was written in Overleaf with the Grammarly browser extension (version 8.929.0) for spelling and grammar correction. Diagrams were created with Microsoft PowerPoint (Version 2504).

3.2 System Description

This section summarizes the capabilities of the software DIAMER, the purpose of the most important components, and the standard workflow to use it. A more detailed description of smaller components can be found in the Section 3.3. A comprehensive overview of the parameters and the command line syntax for DIAMER can be found on the GitHub repository⁵.

The workflow for the taxonomic classification of reads with DIAMER is divided into three main steps: **database indexing**, **read indexing**, and **read assignment** (see Figure 3.1). (1) A protein reference database in FASTA format is preprocessed and a **database index** is created. During preprocessing, all sequence headers are replaced with a header that only contains the taxonomic ID (TaxId) of the lowest common ancestor

⁴<https://www.bioinformatics.babraham.ac.uk/projects/fastqc/>

⁵<https://github.com/husonlab/diamer1>

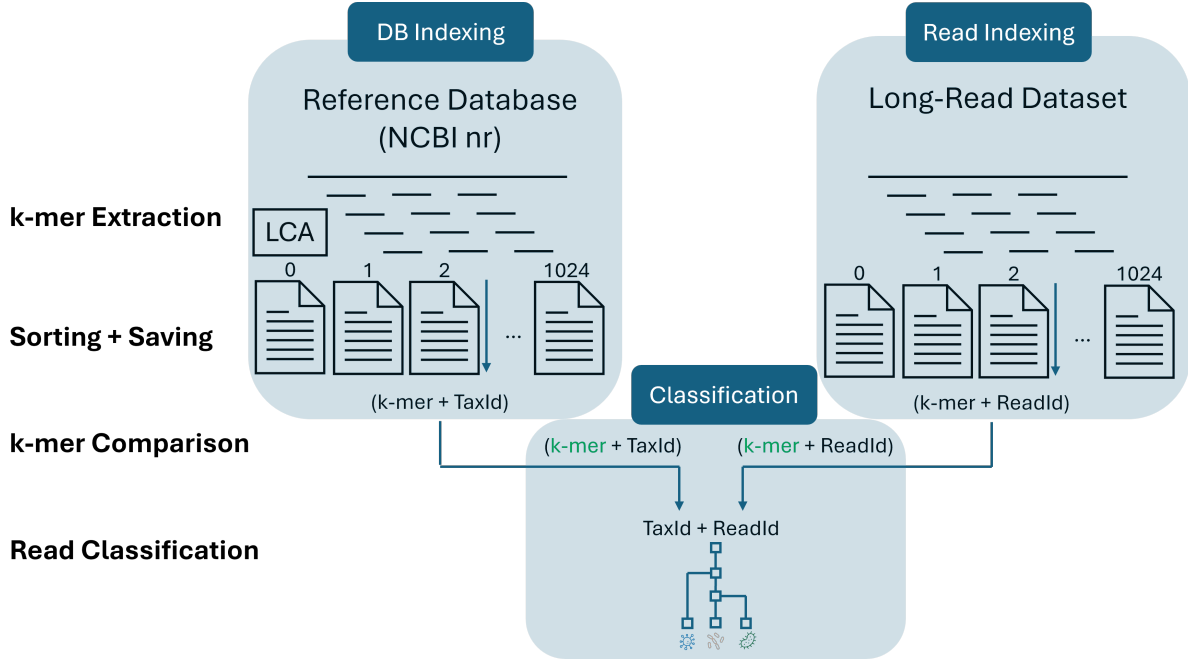


Figure 3.1: Overview of the DIAMER workflow. A database index is created, consisting of a sorted list of k-mer-TaxId-of-LCA pairs, divided into 1024 buckets. The k-mers from a long-read dataset are stored in 1024 buckets with k-mer-ReadId pairs (read index). Both indexes are traversed in parallel to find matching k-mers and thus matching ReadId-TaxId pairs. The matching taxa per read are used for final classification.

(LCA) of all organisms that contain this sequence. Next, a database index is created using the preprocessed database. All k-mers are extracted from the reference database and stored in an index folder on disk. **(2)** The second step processes the reads in the same way and stores the **read index** containing all k-mers on disk. **(3)** Finally, both indexes are compared to find matching k-mers between the database and the reads. Based on the taxa of the matching k-mers in the reference database, a final **classification** to one taxon of the reference database is performed for every read.

3.2.1 Reading the Taxonomy

All three steps of DIAMER require a taxonomic tree to perform LCA calculations and assignments. Currently, only the NCBI taxonomy is supported. For the preprocessing and database indexing step, the taxonomy is read from the *taxdump* files provided by NCBI. Two files are required for this: the *nodes.dmp* file that contains all nodes and edges of the taxonomic tree and the *names.dmp* file that contains the labels (e.g., the species name) for the taxonomic nodes. To preprocess and index the database, those files have to be supplied via the `-no <nodes.dmp>` and `-na <names.dmp>` arguments.

During database indexing, DIAMER stores the taxonomic tree in an adjacency-list-like format in the index folder. For read indexing, no taxonomic information is required, and for read classification, the stored taxonomic tree from the database index folder is used.

3.2.2 Preprocessing

DIAMER can either use the raw NR database that can be downloaded from NCBI, or it can use the clustered databases NR90 and NR50 provided by MEGAN. During preprocessing, DIAMER replaces the header of each sequence with the taxonomic ID (TaxId) of the lowest common ancestor (LCA) of all organisms associated with the sequence. The relation between sequence and taxon that is needed for this can either come from a MEGAN mapping file or the NCBI mapping files. In the case of the clustered NR90 and NR50, only the MEGAN mapping files can be used, since the sequences in these databases only contain a MEGAN-specific IDs. For preprocessing the whole NR database, either the mapping file from MEGAN7 or the *accession2taxid* files from NCBI can be used. The advantage of the MEGAN mappings is that they are stored in a SQLite database, which can be queried easily. In the current implementation of DIAMER, the NCBI mapping files have to be read into a hash table for querying, which requires a lot of memory. However, the NCBI mapping files contain almost all sequence IDs of NR, while the MEGAN mapping file contains only about half of all sequence IDs of the NR database. The syntax for database preprocessing is shown below.

```
java -jar diamer.jar --preprocess -no <nodes.dmp> -na <names.dmp>  
    <DB FASTA> <output FASTA> <mapping file(s)>
```

Preprocessing NR90 & NR50

Each entry in a MEGAN database FASTA file has one sequence ID in the header. Additionally, there are SQLite files that map each sequence ID to a taxonomic ID (TaxId) of the NCBI taxonomy. DIAMER reads through the FASTA file, looks up the encountered sequence IDs in the SQLite database, and replaces them with the corresponding TaxIds. The SQLite database is queried in batches of 100,000 sequence IDs at a time to reduce the number of database hits. Not all sequence IDs have a NCBI TaxId in the MEGAN mapping files, but about 37% of the clusters in NR90 and 33% in NR50 map to GTDB

TaxIds. All sequences that have no valid NCBI TaxId are ignored by DIAMER and written into a separate file for skipped sequences.

Preprocessing NR

In the NR database FASTA file, each entry can have multiple sequence IDs in the header if the sequence occurs in more than one organism. Headers with multiple sequence IDs are replaced by the taxonomic ID (TaxId) of the lowest common ancestor (LCA) of all associated organisms by DIAMER. For the NR database, NCBI provides three different *accession2taxid* files. The default mapping file holds about 1.4 billion entries for all sequences with a GenInfo Identifier. However, the file provides mappings for only about 54% of NR sequences. For DIAMER, the use of the *full* mapping file (~ 7.5 B entries) together with the *dead* mapping file (~ 160 M entries) is recommended, because this way over 99% of the sequences in NR can be used. Collecting the content of both files in one hash table for fast lookup is too memory-intensive, since a hash table with only the default mapping file alone requires almost 200GB of RAM. Therefore, DIAMER reads over the NR database twice. In the first iteration, all sequence accessions are extracted from the headers. Then, only the required mappings can be read from the mapping files into a hash table. Finally, in a second iteration over the NR database, the headers are replaced, and sequences without mapping are diverted into another file. Under low-memory conditions, DIAMER can use the MEGAN mapping file for the whole NR, but it contains only mappings for about 50% of the sequences in NR.

3.2.3 DIAMER Index

Read Index Entry 74 bit			DB Index Entry 74 bit		
Bucket Name	Bucket Entry 64 bit		Bucket Name	Bucket Entry 64 bit	
k-mer I 10 bit	k-mer II 42 bit	Read ID 22 bit	k-mer I 10 bit	k-mer II 42 bit	TaxId 22 bit

Figure 3.2: Scheme of the Index Entries of the read index and the database index. Each Index Entry is stored with 10 bits in the name of a bucket file and with 64 bits (Bucket Entry) sorted in a bucket. The bits that hold the k-mers are divided into 10 bits in the bucket name and 42 bits in the Bucket Entry. The 22 least significant bits correspond to either read ID or taxonomic ID (TaxId).

Similar to DIAMOND [23], DIAMER indexes both the reference database and the long reads as queries in a double indexing-based approach (see Section 2.1.5). A DIAMER index file (*bucket*) is a compressed, sorted list of 64-bit numbers (*Bucket Entries*). A DIAMER *index* is a folder containing 1024 buckets, named from 0 to 1023. The additional 10 bits provided by the name of each bucket result in a total size of 74 bits per *Index Entry* (see Figure 3.2). In the index of a reference database, each Index Entry holds one k-mer and the lowest common ancestor (LCA) of all taxa that have the k-mer in a protein sequence, similar to the index of Kraken [26]. Each Index Entry of the read index holds a k-mer and the read ID the k-mer is associated with. In the database index, each k-mer can only occur once, since all equal k-mers are collapsed to one entry with the LCA of the taxa that contain this k-mer. In the read index, on the other hand, each k-mer can appear multiple times for different reads. The 52 most significant bits that are reserved for k-mer encoding in both Index Entries are enough to encode, for example, 15-mers in an 11-letter reduced amino acid alphabet, 13-mers in a 16-letter alphabet, or 12-mers in a 20-letter alphabet. The remaining 22 least significant bits are enough to encode up to 4M read IDs or taxonomic IDs, respectively. The NCBI taxonomy currently holds about 2.6M taxonomic nodes; thus, the 22 bits are enough to also encode taxonomies with increasing numbers of nodes in the future. Datasets that have more than 4M reads have to be split into batches before they can be processed with DIAMER.

3.2.4 Reduced Amino Acid Alphabets & Unknown Characters

By default, DIAMER uses a reduced amino acid alphabet with 11 amino acids. However, it supports the use of a custom reduced alphabet as long as the alphabet-k-mer-length combination fits into 52 bits. A custom alphabet can be supplied using the command line option `--alphabet <Alphabet>`. The string that defines the alphabet has the following format: `[L][A][GC][VWUBIZO*][SH][EMX][TY][RQ][DN][IF][PK]` and contains groups of amino acids in square brackets, where each group defines one reduced amino acid or cluster. Each letter that is not defined but occurs in a sequence is treated as an unknown character, leading to a separation of the sequence at this position. As a result, if the character for stop codons `*` is not part of the reduced amino acid alphabet used, all translated DNA reads will be split at stop codons. Furthermore, the letters B, J, O, U, X, and Z of the extended amino acid alphabet can be either treated as unknown or added

to the reduced amino acid alphabet, as appears appropriate. Additionally, the parser for custom amino acid alphabets is case sensitive. This allows, for example, to exclude lower-case characters for the filtering of input sequences. The NCBI tool *Segmasker*, which is used in Kraken 2 to filter out low complexity regions from protein sequences [26], masks these regions with lower case letters. Consequently, if DIAMER is run on a reference database that was processed with Segmasker and a custom amino acid alphabet that only contains upper case letters, all masked out lower case letters will be treated as unknown and not be used for k-mer extraction.

3.2.5 Database Indexing

To create an index of a reference database, DIAMER needs a preprocessed FASTA database as input. It iterates multiple times over the input database and collects the k-mers that fall into a subset of the 1024 buckets in each iteration until the k-mers of all buckets are collected. The number of buckets processed in each iteration depends on the available memory and the size of the input database. Each iteration can be divided into three different processing steps. First, all k-mers that correspond to the selected subset of buckets are collected from the database. Next, the Bucket Entries are sorted by their (unsigned) bit value. Finally, the lowest common ancestor (LCA) of equal k-mers from different taxa is calculated, and the buckets are written to disk. During the writing process of the bucket, all k-mers that are found for one taxon in the taxonomic trees are counted. These k-mer counts are saved together with the taxonomic tree in the database index folder and can be used to normalize k-mer hit counts in the read classification process. The minimal command to run database indexing with default settings is shown below.

```
java -jar DIAMER.jar --indexdb -no <nodes.dmp> -na <names.dmp>  
    <DB FASTA> <DB Index Dir>
```

3.2.6 Read Indexing

The current implementation of DIAMER can only index reads in FASTQ format. The indexing process is very similar to the indexing of the database. The main difference is that no lowest common ancestors have to be calculated, and all k-mers remain in the buckets and are written to disk. Since this step is independent of the taxonomy, no files containing the taxonomy have to be supplied:

```
java -jar DIAMER.jar --indexreads <FASTQ> <Read index Dir>
```

3.2.7 Read Classification

To assign DNA reads to taxa in a taxonomic tree, DIAMER needs an index of the reference database and an index of the reads to classify. The classification step can be divided into two phases. In the first phase, all pairs of matching k-mers between the reference database and the reads are identified. Based on the resulting k-mer hit numbers, a specific taxon is assigned to each read in the second phase. To find matching pairs of k-mers, two corresponding buckets of database and reads are traversed in parallel. DIAMER is configured to process one bucket per thread. Accordingly, the number of buckets processed in parallel depends on the number of available threads. In practice, however, the speed of DIAMER in this phase does not seem to depend much on the available threads, but is rather limited by the maximum read speed of the disk that holds the indexes. A k-mer hit is found if the 52 most significant bits of both Index Entries are the same (see Figure 3.2). The corresponding read ID and taxonomic ID (TaxId) that are encoded in the remaining 22 bits of each Index Entry are the information that will be stored for each k-mer hit, and are used for read classification in phase two. During read classification, the k-mer hit numbers are stored as tuples of the form (TaxId, k-mer count) for each taxon that has a matching k-mer with a read. The one-vs-one (OVO) and one-vs-all (OVA) classification algorithms that are implemented in DIAMER (for details see Section 2.2.3) find a taxonomic assignment for a read based on a weighted subtree of all taxa that have matching k-mers with the read. The weight of each node v_i in a subtree in DIAMER can be either the raw k-mer hit count k with $w_r(v_i) = k$ or the k-mer hit count normalized by the total number of k-mers that map to the taxon in the reference database k_d with $w_n(v_i) = \frac{k}{k_d}$. The classification algorithms and their

threshold parameter t that should be used in the classification step of DIAMER can be specified via command line options. The default algorithm used is OVO with $t = 1$, which corresponds to the Kraken classification algorithm. The command for the default setup is shown below.

```
java -jar DIAMER.jar --assignreads <DB Index Dir> <Read Index Dir>
    <output Dir>
```

3.2.8 Output Files

DIAMER produces four different output formats in the read classification step. The first file created after all k-mer matches have been identified is labeled *raw_assignments.tsv* and contains a tab-separated table with one read per row that consists of its header together with a space-separated list of the taxonomic ID-k-mer-count tuples, similar to the standard output format of Kraken 2[26]. With this file, the assignment algorithms for raw k-mer counts could be run with different settings again, without repeating the k-mer search itself. However, in the current implementation, there is no command-line parameter to trigger this process. The second output file, labeled *per_read_assignment.tsv*, has one column for the read ID and one column for every combination of assignment algorithm, raw k-mer counts or normalized k-mer counts, and the specified thresholds. Each entry in the classification columns contains the taxonomic rank of the taxon, its label, and the taxonomic ID (TaxId). The third output file (*per_taxon_assignment.tsv*) contains a list of all taxa in the taxonomy with columns for the TaxId, the rank, the label of the taxonomic node, the number of k-mers that were collected in the database for the taxon, the number of k-mers that matched with reads in the dataset, and the normalized number of k-mers that matched with reads. Additionally, there are two columns for every algorithm column in the *per_read_assignment.tsv*. One contains the number of reads that were directly assigned to the taxon in question, and the other contains the cumulative count of reads that were assigned to the subtree that is rooted at the current node. Additionally, DIAMER saves a file (*megan.tsv*) that only holds the TaxId and the raw k-mer hit count to reads, which can be parsed and viewed in MEGAN. If the input reads need to be binned, the *per_read_assignment.tsv* is the file that holds this information. If DIAMER is used to estimate taxon abundance, the *per_taxon_assignment.tsv* can be used.

Log Files

For each of the four processing steps in DIAMER (preprocessing, database indexing, read indexing, read classification), a log file is produced. This file contains all settings that were used for this specific computational task, as well as further information about the data that has been processed or errors that occurred, depending on the task. The preprocessing log file records how many sequences have been successfully annotated with a taxonomic ID (TaxId), how many could not be associated with a taxon, and how many sequences per taxonomic rank are in the preprocessed database file. Both indexing steps report the frequency of each character of the reduced amino acid alphabet that was used, the size of each bucket, the total number of extracted k-mers, and the number of sequences (or translated sequences) that have been processed or skipped because they were too short for the selected k-mer size. The classification log file lists the number of reads that were assigned or not assigned. The settings in the log files are important for the repeatability of experiments, while the additional information aids with debugging and provides some useful statistics.

Console Logging

For every task performed by DIAMER, it logs the current task to the console. Timestamps, runtime, and a simple progress bar are shown for most processes so that the user can estimate the runtime and see the current status of a computation. Each log is tagged with the name of the class that produces the log and with either [INFO], [WARNING], or [ERROR] to facilitate debugging.

3.2.9 Suggested Design Improvements

The current implementation of DIAMER is especially suitable for trying out different algorithms and changing all the parameters in the code. However, several things can be improved for a better user experience. During development, it was important to keep the database preprocessing step separate from the database indexing step to avoid needing a sequence ID to taxonomic ID (TaxId) map for every creation of an index file. The same preprocessed database could be used repeatedly to generate index files with different parameters. If DIAMER were to be used in real experiments, this feature

would be less useful, but it introduces one additional step in the pipeline. Therefore, combining the preprocessing and indexing steps into one task for DIAMER would make sense. Furthermore, DIAMER is restricted to the use of the NCBI taxonomy for now. It would make sense to support other taxonomies, such as the GTDB [42] taxonomy. The use of only FASTQ files as input format is another constraint for users. DIAMER could be changed to allow the use of FASTA input files as well. Storing DIAMER indexes as loose files in an index directory is not optimal, since individual files could be lost. A possible solution is to pack the individual files into an archive or to write all buckets into one large file. Another feature that is missing at this stage is the selection of one specific output format and the columns that should be generated in this file, to avoid computing and storing all four output files.

3.3 Implementation Details & Challenges

DIAMER is implemented in Java 23 and can be run on any platform with a recent Java Runtime installation. This chapter illustrates the most important design decisions, Java classes, and algorithms used to build DIAMER, as well as some problems and limitations that emerged in the process. For all classes that are not described in this section in more detail, please refer to the documentation in the source code.

3.3.1 Programming principles

The Java classes and packages of DIAMER are organized according to the *Separation of Concerns* principle. Therefore, the different computational tasks of DIAMER, such as database indexing, read indexing, or read classification, are encapsulated in separate classes. Furthermore, the *Single-responsibility principle* finds application in the *IO* package and the *Encoder* and *GlobalSettings* classes. If a class of DIAMER needs to be written to and read from a file - examples would be the database or read index, the buckets, the taxonomic trees, or the raw k-mer assignments - both operations are set up in the same class. Accordingly, only one class has to be adapted if the file formats were to be changed. The *Encoder* class holds all functions that determine how the k-mers, taxonomic IDs, and read IDs are encoded into an Index Entry. If one needed to shrink the bits of a Bucket Entry from 64 to 32, only the *Encoder* class would need to be extended. The *GlobalSet-*

tings class holds all input parameters, the hard-coded parameters, such as queue sizes for concurrent operations, and the maximum number of input/output threads. These settings can influence performance and can be fine-tuned in this class.

3.3.2 Reading Sequence with the `SequenceSupplier` class

The *SequenceSupplier* is one of the most important classes responsible for reading sequences from different input formats (FASTA or FASTQ), converting them to a desired reduced amino acid alphabet, and keeping the sequences cached if necessary. The class uses different instances of the *SequenceReader* class to decode different input formats, particularly FASTA or FASTQ files with either strings or IDs as headers. During initialization, a *Converter* can be specified to convert the input sequences into different reduced amino acid alphabets. Different *Converters* are used to convert the protein sequences of the reference database into the reduced amino acid alphabet, or to simultaneously translate and convert the input DNA reads into all six possible reading frames and the reduced alphabet. The conversion and caching of sequences is mediated by the *FutureSequenceRecord*. The *SequenceSupplier* does not return a *SequenceRecord* object directly (which is used to handle header-sequence pairs in DIAMER), but a *FutureSequenceRecord* object. Only another call to a function of the *FutureSequenceRecord* returns the converted *SequenceRecords*. This decouples the reading operation from the conversion (and caching) and allows the conversion to be performed *lazily* by a thread other than the reading thread. The method call that triggered sequence conversion by the *FutureSequenceRecord* is additionally used to cache the converted sequence in a list of the *SequenceSupplier*. If the command line flag `--keep-in-memory` is set, the *SequenceSupplier* maintains a list with all lazily converted *SequenceRecords* so that the reading operation and conversion must only be performed in the first iteration over a sequence input file. Furthermore, the interposed *FutureSequenceRecord* class is not restricted to returning one converted *SequenceRecord*, but it returns a list of sequences, which is particularly useful for the translation in six reading frames, or for dividing a sequence at stop codons.

3.3.3 Challenges in Text File Reading

DIAMER supports the use of uncompressed and Gzip-compressed sequence files as input. All sequence reader classes in DIAMER depend on the *BufferedReader.readLine()* method

to read sequence strings. This method produces a new *String* object for every line it reads. Reading compressed files offers the advantage that the information which is encoded in the same amount of disk space is much larger. Consequently, slower file systems can supply sequences at a higher rate than would be possible with an uncompressed file. However, Java’s default Gzip package seems to limit the read speed a lot. Read speeds of 60 to 80 mb/s could not be exceeded in any test environment. It is also not possible to decode compressed files with multiple threads, since each decoding step depends on previous information. In summary, the use of only one reader thread with *BufferedReader.readLine()* and the speed limiting decompression are the most important factors that slow down the reading of sequence files in the current implementation of DIAMER.

3.3.4 k-mer Encoding, Extraction & Filtering

All k-mers are encoded by interpreting each reduced amino acid as a digit in a number system with the same base as amino acids in the alphabet. The extraction of k-mers is handled by two classes in DIAMER, the *KmerEncoder* and the *KmerExtractor*, and follows a sliding window approach. The encoding of the k-mers is handled by the *KmerEncoder* class, while filtering is done by the *KmerExtractor*.

k-mer Encoding

The *KmerEncoder* represents the sliding window of a k-mer or spaced seed, which can be slid forward by adding a letter to it. This causes the byte array that represents the window internally to shift its content one position to the left and write the byte representing the new amino acid to the now free, last position of the array. Depending on whether the mask used for k-mer extraction has spaces or not, the calculation of the value that represents the k-mer can happen in two different ways. In a consecutive mask, the value of k-mer $S_{i,j} = s_i, \dots, s_j$ of length $k = |S_{i,j}|$ in an alphabet of base b is always encoded in a variable $v(S_{i,j}) = b^{k-1}s_i + b^{k-2}s_{i+1} + \dots + b^0s_j$ that is updated once a letter is added to the window. With these definitions, the value of the next k-mer $v(S_{i+1,j+1})$ is given by this recursive update rule: $v(S_{i+1,j+1}) = b \cdot (v(S_{i,j}) - b^{k-1}s_i) + s_{j+1}$. This allows DIAMER to calculate the next k-mer recursively using the value of the old k-mer very efficiently. The most time-consuming step is updating the array that keeps track of the

amino acids currently in the window. For spaced seeds as k-mers, the calculation is more difficult, since there is not only one letter that moves out of the window and another one that enters the window, but there are multiple letters that move in and out of the masked regions. In that case, DIAMER maintains a second array containing the letter values that are already multiplied by b^l , where l is the position in the mask. For each window shift, every non-masked position of the window is multiplied by the corresponding value of b^l and summed. The $s_i \cdot b^l$ results are precalculated in a two-dimensional array for speed up. In the case that the alphabet in use has a base that is a power of 2, all multiplications with the base or powers of the base can be replaced with bitshift operations, which would further simplify the calculation. However, this case is not handled separately in the current implementation.

k-mer Extraction & Filtering

The *KmerExtractor* class extracts all k-mers from a sequence by adding one amino acid after the other to the *KmerEncoder*. At the same time, it can access different properties of the k-mers provided by the *KmerEncoder* that enable it to filter out k-mers based on specified thresholds. Furthermore, the *KmerExtractor* can be configured to extract only minimizers.

Two methods of the *KmerEncoder* allow the computation of a sequence complexity measure and a measure of the probability of a k-mer based on the reduced amino acid frequencies. The complexity measure is the number of distinct amino acids in the k-mer or the non-masked region of the spaced seed. This value can be used to filter out k-mers with fewer than four different amino acids. In the NR database, for example, multiple sequences have large regions of unknown amino acids, as indicated by the symbol x. If the amino acid alphabet used encodes the letter x, this leads to the procession of many k-mers with the same value by DIAMER. By filtering out k-mers with a very low complexity, negative influences on the running time of subsequent steps can be avoided. The default filtering setting in DIAMER is a complexity larger than three. Discarding k-mers with fewer than three different amino acids could effectively decrease the amount of total extracted k-mers, while keeping almost the same amount of extracted distinct k-mers.

For the probability measure, the *KmerEncoder* returns the product of the individual amino acid probabilities encoded in the current k-mer. This requires knowledge of the probability of each amino acid in the input dataset. This information has to be supplied to the *KmerEncoder* if the probability measure is to be used.

The *KmerExtractor* can filter all k-mers with the minimizer concept. For prioritizing k-mers within a specified window, either the k-mer probability or the k-mer complexity can be used. Therefore, either a **probability minimizer** or a **complexity maximizer** is extracted from a window. Compared to the original concept of minimizers [30], DIAMER only extracts exactly one minimizer per window shift, even if the scoring criterion yields two maximally scoring k-mers. For ambiguous cases, the rightmost k-mer is used, since it will remain in the window the longest. Furthermore, the concept of *end minimizers* [30] is not implemented in DIAMER. Adding this feature would increase performance for alphabets that encode stop symbols, since the stop symbols would always be included. If minimizers are enabled, DIAMER keeps track of the highest-scoring k-mer S_{max} in the current window via an array of all k-mers in the window and the index i_{max} of the currently highest-scoring k-mer in that array. For a window shift, three cases can occur. (1) The new k-mer has a higher score than S_{max} of the old window. Hence, i_{max} is updated to the position of the new k-mer. (2) The old S_{max} is no longer in the window ($i_{max} < 0$). This triggers a method that re-computes i_{max} . (3) The old S_{max} still has the highest score and is in the window.

During the k-mer extraction process in index generation with spaced seeds, the translation or conversion of the sequence, the encoding of the k-mer, and the calculation of the filtering measures need about one-third of the total computation time each. The fastest k-mer extraction can be achieved with consecutive k-mers without any filtering, and for even more speed, a cached sequence supplier with already translated/converted sequences.

3.3.5 Bucket-wise k-mer Collection

One of the key features of DIAMER, similar to DIAMOND [23], is that a subset of k-mers that belong to a specific range of buckets can be collected in one iteration. With this strategy, the required memory can be limited to the memory required for one of the 1024 buckets, allowing processing large databases with limited amounts of RAM. In part, the

k-mer collection process for the database index and the read index are quite similar. In both cases, k-mers have to be collected and sorted in the bucket they belong in, together with the taxonomic ID (TaxId) or read ID. In the end, all Bucket Entries have to be sorted and written to disk. The largest difference that complicates the k-mer collection for the database index is that, in contrast to the read index, not every k-mer can be kept. If the same k-mer is collected multiple times from different taxa, the bucket entries have to be merged to one entry on the lowest common ancestor (LCA) of all containing taxa, similar to the hash table in Kraken 2 [26]. Furthermore, thought has to be put into how to determine in which bucket a specific k-mer belongs. If the k-mers are distributed unequally across the buckets, the memory and processing time per bucket can not be distributed optimally.

Equal k-mer Distribution across Buckets

Each k-mer extracted with DIAMER has at most 52 bits (see Figure 3.2). 10 bits of the k-mer have to be used to determine in which bucket the k-mer falls. These bits will be removed from the k-mer encoding, but can be restored from the name of the bucket. This ensures that all information encoded in the k-mer is preserved. Different approaches have been tested on how to extract bits from the 52 bits of a k-mer in a way that distributes them equally across all buckets. It makes sense to avoid using the most significant bits of a k-mer encoding as a bucket name, since they might not always contain information. For example, a 16-letter reduced amino acid alphabet with a k-mer of length 12 would only require 48 of the available 52 bits, which would leave all except for the first 16 buckets empty. The simplest approach is to use the 10 least significant bits as the bucket index. This works reasonably well for amino acid alphabets with a prime number of amino acids. However, for alphabets with other bases or especially alphabets that are divisible by a multiple of two, the unequal distribution of reduced amino acids interferes with the bucket assignment, which leads to some very full buckets and some that are almost empty. Even extracting bits with a random pattern and shuffling them leads to biased results. The method that seems to work best is to use the 10 least significant bits paired with an XOR operation that flips every second bit. This technique has been used in Kraken 2 to extract unbiased minimizers [26] and is also used in DIAMER to avoid unequal bucket sizes.

Arrays for k-mer Collection

During the development of DIAMER, different data structures were used to collect the k-mers during database and read indexing. The first datastructure that was used to store k-mer-read ID pairs for read indexing was the default *ConcurrentLinkedQueue* class of Java. This seemed to make sense at first, since the queue allows concurrent operations and has no fixed size. For the database k-mer collection, the Java *ConcurrentHashMap* was used in the beginning, since it ensures that a k-mer is only stored once and the rediscovery of a known k-mer can be used to trigger a lowest common ancestor (LCA) algorithm. However, multiple tests on the *server* showed, that the Java Virtual Machine needed almost as much time for garbage collection as it needed to extract k-mers. This was probably due to the need to store objects in both of these generic classes, which overloaded the garbage collector. In the current version of DIAMER, the class *FlexibleBucket* is implemented and replaces the *ConcurrentLinkedQueue* and the *ConcurrentHashMap*. This class uses *long* arrays to store Bucket Entries. It can be supplied with an estimate of the size of a bucket in the beginning, and generates a single array of that size. If the initial array is not large enough, more arrays are added to a list with a predefined size to extend the bucket. To enable concurrent operations on those arrays without locks, all threads that need to write to the *FlexibleBucket* have to request an index range called *contingent* from the flexible bucket first. In this way, only the *getContingent* method requires synchronized operations. As long as the threads only write to the index range of their contingent, no errors will happen. If there is a thread out of indexes, it can call the method to get a new contingent. The only drawback of the current implementation is that a *FlexibleBucket* does not know which parts of a contingent that was handed to a thread have actually been used. So there will be some unused fragments of arrays after each k-mer collection phase. This is handled by initializing the arrays with the maximal long value, which will be one coherent block after sorting and can be ignored by subsequent processing steps.

The downside of using arrays for the database k-mer collection is that all k-mers have to be collected, even though only unique k-mers are needed. The uniqueness of k-mers is restored during the writing process of each database index bucket. Once all equal k-mers have been clustered together by the sorting, only one entry per cluster is written to the file with the LCA of the source taxa.

Bucket Size & Amino Acid Probability Estimation

For the initialization of the *FlexibleBucket*, the bucket sizes have to be estimated. Furthermore, the *KmerEncoder* needs to know the probability of each reduced amino acid. To estimate both values, DIAMER analyzes the first 10,000 FASTA entries or the first 1,000 FASTQ entries, respectively. Using this information, the expected maximal bucket size can be calculated from the total file size and the amino acid distribution is calculated based on this sample. Certainly, the first sequences might not be a representative sample of the whole sequence file and there might be instances where this estimation does not yield proper results. However, the results are recorded in the log file and can be reviewed for correctness. Additionally, DIAMER uses the maximum bucket size estimation to suggest the number of buckets that can be processed in parallel depending on the available memory. If the number of buckets is not set by the user with the option `-b <number>`, the suggestion is used. In this case, if the estimation is too far off, DIAMER might run out of memory because of incorrect bucket size estimation. Furthermore, the suggested number of buckets does not consider whether sequences should be cached by the *SequenceSupplier* or not (option `--keep-in-memory`), which could also lead to a wrong estimation of the required memory.

Parallel Binary Radix Sort

Since DIAMER uses almost all available memory to collect as many buckets as possible in parallel, an in-place sorting algorithm is necessary to avoid doubling storage requirements during sorting. Binary radix sort is implemented in DIAMER for that purpose. It sorts the contents of a *FlexibleBucket* by one specific bit per iteration. Starting with the most significant bit of each Bucket Entry, it collects all numbers with a 0 in this position in the first half of the *FlexibleBucket*, and all numbers with a 1 in the second half. Then, the following bit can be sorted in the same way in both parts of the array independently. The necessary computations can be modeled as a divide-and-conquer approach, since the problem can be divided into two subproblems with each iteration. Therefore, a Java *ForkJoinPool* can be used to submit the sorting tasks to, which can fork off other tasks by themselves. In this way, all available threads (or threads specified by the user) are used for sorting as soon as the number of array parts to sort is larger than the thread number. Due to the fixed number of 64 bits per Bucket Entry, the runtime of this sorting algorithm

depends linearly on the number of Bucket Entries. In practice, the performance seems to depend heavily on the fragmentation of the *FlexibleBucket* that is sorted. If the initial chunk size is chosen well, the sorting happens on only one array, which was much faster than when the *FlexibleBucket* had to expand multiple times and the Bucket Entries to sort are spread over multiple arrays.

3.3.6 Concurrency in DIAMER

All computation steps in DIAMER, except for the preprocessing, are set up to use multiple threads, if available. The next two sections will go into more detail about how concurrency is achieved in each step.

k-mer Extraction, Sorting & Writing

All three steps of database and read indexing are set up to distribute computations between multiple threads. The k-mer collection phase of both indexing steps follows a *producer-consumer pattern*. One reader thread produces batches of *FutureSequenceRecords* objects and adds them to a *BlockingQueue*. A variable number of consumer threads is set up to retrieve the batches from the blocking queue and extract all k-mers from the sequences. The k-mers are stored in *FlexibleBuckets* that are shared among all threads. Thread safety in the *FlexibleBuckets* is achieved through dedicated index contingents, which each consumer thread is allowed to write to. Once one iteration over the input file is complete, all *FlexibleBuckets* are sorted with the parallel implementation of binary radix sort.

Read Classification

During read classification, the comparison of one pair of buckets is considered one task that can be processed by one thread. In the *ReadAssigner* class, a *ThreadPoolExecutor* is set up to process all bucket comparison tasks. The found k-mer matches are stored in one *ReadAssignment* class, which allows for thread-safe collection of all matching taxonomic IDs per read. While running assignment algorithms on the k-mer matches that were found for one read, the classification of one read is considered one task. All assignment algorithms that are required to run are submitted as one task per read to another *ThreadPoolExecutor* and can be processed in parallel.

3.3.7 Delta encoding

The content of the bucket files in a DIAMER index is a list of sorted, unsigned 64-bit numbers. The values alone can be quite large, and every single value needs 64 bits of storage. However, since the values are sorted, the difference between subsequent numbers is usually smaller and does not require 64 bits of storage. For this reason, the buckets are written in *delta encoding*, where every number is encoded as the sum of its predecessor and the difference to its own value (delta). The encoding only helps with a format that can use a flexible number of bits to encode numbers. Therefore, DIAMER writes all subsequent deltas in 8-bit packages, where the first bit is a continuity flag that remains on until the final byte of the current delta is reached. With this technique, the index file sizes of DIAMER could be reduced by about 40%, and an index with an 11-letter alphabet of NR only needs 369 GB of storage instead of 611 GB without delta encoding.

3.3.8 Code Documentation and Unit Tests

The source code of DIAMER is documented with varying richness of detail, depending on how often the code has been changed. Code that is in the project from the beginning is documented in high detail, while segments that changed frequently might not be documented sufficiently for others to understand them.

Throughout the development of DIAMER, many test classes have been created to test single classes or larger units of the software. However, most classes were altered and optimized so often that it became unfeasible to also keep testing classes up to date with the rest of the project. The only testing class that remained through all code changes and was crucial in discovering bugs and assuring the correct function of DIAMER is a class that runs all steps of the program on a test dataset. The true intermediate results and final results of this test dataset are known and can also be derived by hand, since the dataset is very small. The *CompleteRunTest* creates the output files of every step of DIAMER with a consecutive and a spaced seed and compares the computed *test_results* with the folder *expected_results* to assure correct function of DIAMER. Since this class only depends on the command line input and the final result files, it was easier to maintain than class-specific test classes.

It would have been possible to create documentation and test classes for the code with

AI tools, such as GitHub Copilot, which was used throughout the project. However, if this project is going to be continued at some point, future AI assistance will probably be even better in explaining and analyzing the code than the models that could have been used now. Therefore, no AI tools were used for this purpose in the hope that better AI models will help reuse and maintain the code in the future.

3.3.9 Possible Improvements

Throughout the project, code was constantly optimized for faster computation times. Still, there is much that can be improved. Some ideas that will improve performance but could not be realized so far are discussed in this section. First of all, all *Nodes* of the *Tree* class for taxonomic trees are implemented as objects. While this facilitates the implementation of graph algorithms, it is not the best choice for performance and memory optimization. Storing information about nodes, their connectivity, and their properties in arrays would be much more memory efficient. This could be realized, for example, with a connection table-based approach. Another limitation is the use of the *BufferedReader.readLine()* method in all sequence parsers. This method generates *String* objects with each call that have to be cleared by the garbage collector again at some point. A more efficient way of reading could be to directly implement a byte buffer in every sequence reader, which can be copied and parsed into a char array without the intermediate generation of *String* objects. Additionally, at least for uncompressed input files, multi-threaded file reading could be considered. Another useful feature would be the combination of complexity and probability filtering. The current implementation allows the selection of one filtering option only. However, longer stretches of unidentified amino acids in NR - indicated by the letter x - have to be filtered out via the complexity filter. The probability filter might keep k-mers that consist of consecutive x, since the filtering only depends on the individual frequency of reduced amino acid groups within the whole dataset. The support of *end-minimizers* would make sense especially for reduced amino acid alphabets with stop codons, but is not included in the current version. Furthermore, it might be useful to be able to supply a custom maximal bucket size and custom amino acid probabilities for the initialization of *FlexibleBucket* and the filtering in the *KmerExtractor*. This would allow the user to overwrite wrong estimates that are based on possibly unrepresentative sequences at the beginning of a FASTQ or FASTA file.

Chapter 4

Results and Evaluation

4.1 Methods for Statistical Analysis

Apart from measuring the running time and memory required, there are several different methods to evaluate the performance of a taxonomic classification or abundance estimation tool. In most cases, read datasets of known composition are used for evaluation [52, 15, 20]. These can be either simulated reads or reads from real experiments of a known sample. Knowledge about what species should be in the sample can then be used to calculate statistical parameters, such as precision and recall.

The evaluation method that comes closest to a real-world project, with e.g., aquatic or soil samples, is a *strain exclusion experiment* [26]. For this, a dataset of known species composition is created together with a reference database, where all species of the sample are removed. This makes sense because in metagenomics, the major part of most samples will consist of species that can not be found in any reference database [58]. Evaluating the performance of an algorithm at a rank higher than species level allows for drawing conclusions about its precision and accuracy for novel species [26].

For the analysis of metagenomic samples from a better-studied environment, such as the characterization of the human gut microbiome [59], more reference sequences are available, and the identification of novel organisms is less important. For the evaluation of a classification tool for this scenario, unmodified reference databases can be used. In this project, only unmodified reference databases were used for testing.

4.1.1 Taxon-based Analysis

In the literature, two different methods are applied to calculate precision and recall for datasets of known compositions. One possibility is to calculate these values based on the identified taxa [52]. For a read classifier, a taxon would be considered as identified once the number of reads assigned to it lies above a defined threshold. The threshold can be lowered from a value that is too high for any taxon to be considered identified to a value where all taxa that are in the sample are considered identified. For each threshold u where a new taxon is considered identified, precision and recall can be calculated based on true positive TP , false positive FP , and false negative FN identified species [52]:

$$precision_u = \frac{TP_u}{TP_u + FP_u} \quad (4.1)$$

$$recall_u = \frac{TP_u}{TP_u + FN_u} \quad (4.2)$$

For all values of u between $TP_u + FP_u = 0$ and $FN_u = 0$, a precision-recall curve can be plotted and used for the evaluation of a tool. This kind of plot is especially useful when comparing sequence classifiers to taxonomic abundance estimators, since a low read assignment count is not punished. However, for this project, the plots are not very informative (see Figure A.1). Thus, only read-based statistics are used here.

4.1.2 Read-based Analysis

An alternative to calculating precision and recall based on the identified taxa is to use the read counts directly [22]. Accordingly, true positives (TP) are defined as the number of reads assigned to a taxon that is expected to be in the sample, false positives (FP) are the reads assigned to taxa that should not be in the sample, and false negatives (FN) are all unassigned reads. Precision and recall can be calculated depending on one taxonomic rank:

$$precision = \frac{TP}{TP + FP} \quad (4.3)$$

$$recall = \frac{TP}{TP + FN} \quad (4.4)$$

In this thesis, the data is either visualized as precision-recall plots or only the recall is shown in a stacked bar chart across all taxonomic ranks.

4.2 Parameter Search

DIAMER allows the user to specify many parameters during index generation and read classification. During database and read index generation, the reference database, the reduced amino acid alphabet, as well as the k-mer shape and k-mer filtering options, can be customized. For the classification process, the user can choose the classification algorithm and the threshold. In this section, the results of classification runs using different parameters will be discussed to test the capabilities of DIAMER and to provide guidelines on which parameters to use for which scenario.

4.2.1 Reference Database

The reference database used has an unavoidable influence on the results of any read classification tool [15]. To illustrate this, the full NR database and two clustered versions of NR were used here. All reference databases and the ZymoMock dataset were indexed with a consecutive 13-mer and the Uniform11S alphabet. With these parameters, the index of the full NR with about 700M sequences is about 369 GB, the index of NR90 (~ 240 M sequences) is 171 GB, and the index of NR50 is 46 GB. The read-based precision and recall are plotted in Figure 4.1 for the assignment algorithm one-vs-one (OVO) with threshold $t = 1$ and four different taxonomic ranks.

As expected, the classification results are the best for the full NR, with a precision of over 80% and a recall of over 60% at species level. This can be explained by the differences in reference database size. The largest reference database performs best, since the probability of k-mer matches between the dataset and the database is much higher. The index of the full NR holds about 266B k-mers, while the index of NR50 holds less than 4% of this amount (~ 10 B). This can also be seen in the number of matching k-mers between the database and the dataset. For the full NR, this is 753M k-mers and only 224M for NR50.

Despite the far smaller index size of NR50 (only 49 GB compared to 369 GB for NR), the precision at lower taxonomic ranks is not affected much by the size difference. At the family and order ranks, the precision for NR50 exceeds 85%, which might be sufficient in some scenarios. This is consistent with the results of the Kraken 1 publication [22]. High precision does not require a large database, and can be even better with a smaller

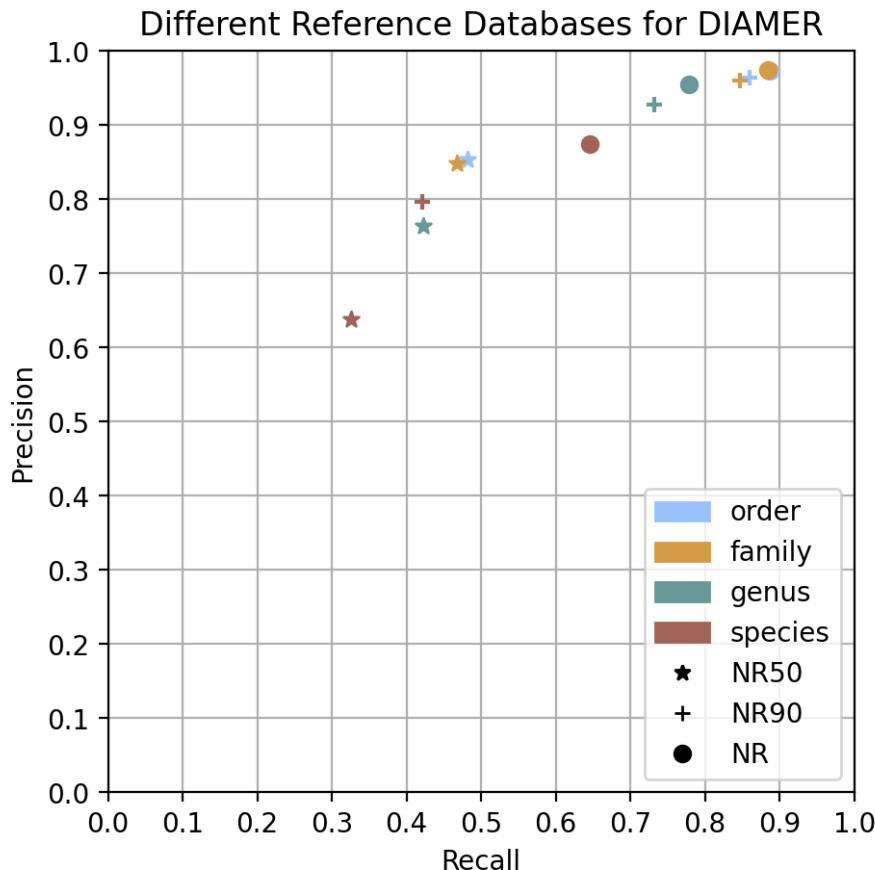


Figure 4.1: Comparison of the performance of DIAMER with different reference databases. The database index files for NR (green), NR90 (blue), NR50 (red), and the index file for the ZymoMock dataset were created with a consecutive 13-mer in the Uniform11S alphabet and default filtering. Shown are the read-based precision and recall for the ranks *species*, *genus*, *family*, and *order* with increasingly light colors. The one-vs-one (OVO) assignment algorithm with $t = 1$ was used.

database. Consequently, it makes sense to use a smaller database for tasks where a small database and short computation time are beneficial, and only the abundance estimation without assignment of each single read is required. However, using a large reference database can not be avoided for tasks where high recall and precision on low taxonomic levels are required.

4.2.2 Classification Algorithms & Thresholds

DIAMER supports the use of two *PushDown* algorithms developed by Julia Fischer [53] (see Chapter 2.2.3) for the final classification of the input reads. The algorithms operate on a weighted subtree of the reference taxonomy, and DIAMER runs each user-specified algorithm-threshold combination on four different weighted subtrees. The subtrees can be

weighted with the **k-mer count**, which is the number of k-mers that are shared between the sequences of a taxon in the reference database and the input read. Additionally, these weights can be accumulated along the taxonomic tree, resulting in a tree weighted with the **k-mer count (cumulative)**. These two weights can also be computed from the output of Kraken 2X. Trees weighted with the **normalized k-mer count** and the **normalized k-mer count (accumulated)**, however, can only be generated with DIAMER. For the normalized k-mer count, the number of matching k-mers is divided by the number of different k-mers in the reference database for a specific taxon. This normalization is supposed to suppress the bias towards well-studied model organisms in the reference databases.

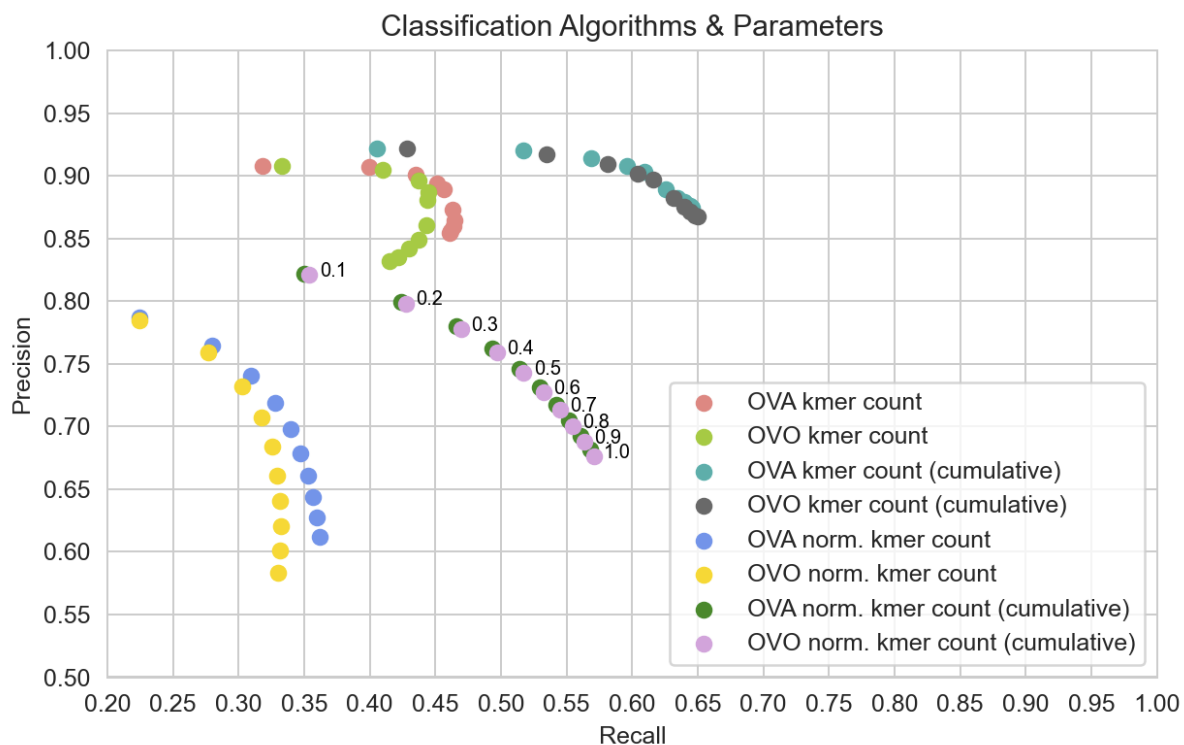


Figure 4.2: Comparison of the classification algorithms one-vs-one (OVO) and one-vs-all (OVA) with different threshold values and on differently weighted subtrees. Precision and recall are read-based and were calculated on species level from the classification of the ZymoMock dataset with the full NR database as reference and an index with consecutive 13-mers in the Uniform11S alphabet. Both classification algorithms are run on subtrees weighted with the count of k-mer matches, the cumulative k-mer match count, the k-mer count normalized by the number of k-mers extracted from the reference database per taxon, and the cumulative normalized k-mer count. Threshold values are indicated for the OVO algorithm with a subtree weighted by the cumulative normalized k-mer count. Be aware of the differently scaled axes.

A read-based precision-recall plot on species level for both algorithms with different

threshold values and differently weighted trees is shown in Figure 4.2. The results from subtrees that are weighted only with the raw k-mer match count without accumulation always perform worse than the algorithms that use the accumulated values. This is to be expected, since there is no biological reason for not accumulating k-mer counts across a taxonomic tree. This weighting system was included in DIAMER as a control and is only shown for completeness. When comparing the results for the cumulative weights, the not-normalized k-mer count outperforms the normalized k-mer count in both precision and recall for any threshold. Since the normalized k-mer count was introduced to mitigate the bias of reference databases towards model organisms, the advantage of the normalization might not show up for the dataset used, because all species contained are well-studied. Furthermore, the simple division of k-mer counts by the number of k-mers in the database per taxon might boost random hits to species with a low k-mer footprint in the database. Additionally, the calculation is numerically unstable, which can distort the results. Other normalization strategies should be tested here.

The best-performing subtree uses the cumulated k-mer counts as weights. Both algorithms, one-vs-one (OVO) and one-vs-all (OVA), perform similarly across different threshold values (see cyan and gray points in Figure 4.2). Notably, the OVA algorithm always has a slightly better precision, but a much worse recall compared to the OVO algorithm with the same threshold parameter. This is expected because the OVA algorithm will never walk further down the taxonomic tree than the OVO algorithm. As a result, the classifications of the OVA algorithm will always be on the same, or a higher taxonomic rank, which leads to a higher precision (fewer false positives) but a lower recall (more high-rank classifications) on species level.

The increased threshold parameter for both classification algorithms leads to a higher recall at the cost of precision. This is because a higher threshold allows the algorithms to walk further down the taxonomic tree and to classify at a lower taxonomic level. This increases recall because every additional classification on species level (in this case) can only increase the true positives. At the same time, reads might be assigned to species level that do not have enough support for a reliable, more specific classification, which leads to more false positives and decreases precision. Overall, changing the threshold from $t = 0.1$ to $t = 1$ increases the recall by over 20%, while the precision drops by a little more than 5%. Since the precision is still much higher than the recall, for most

applications, the higher recall will have higher benefits than the higher precision. For most scenarios, a threshold of $t = 1$ is advisable for DIAMOND and Kraken 2X (see Figure A.2), which results in the Kraken classification algorithm for the OVO algorithm.

4.2.3 K-mer Length & Spaced Seeds

The user can specify the length and shape of the k-mers used by DIAMER. However, the 52 bits available for storing a k-mer should not be exceeded when using large k-mers. Figure 4.3 compares different k-mer sizes for the ZymoMock dataset with the NR90 database as reference and the Uniform11S reduced amino acid alphabet.

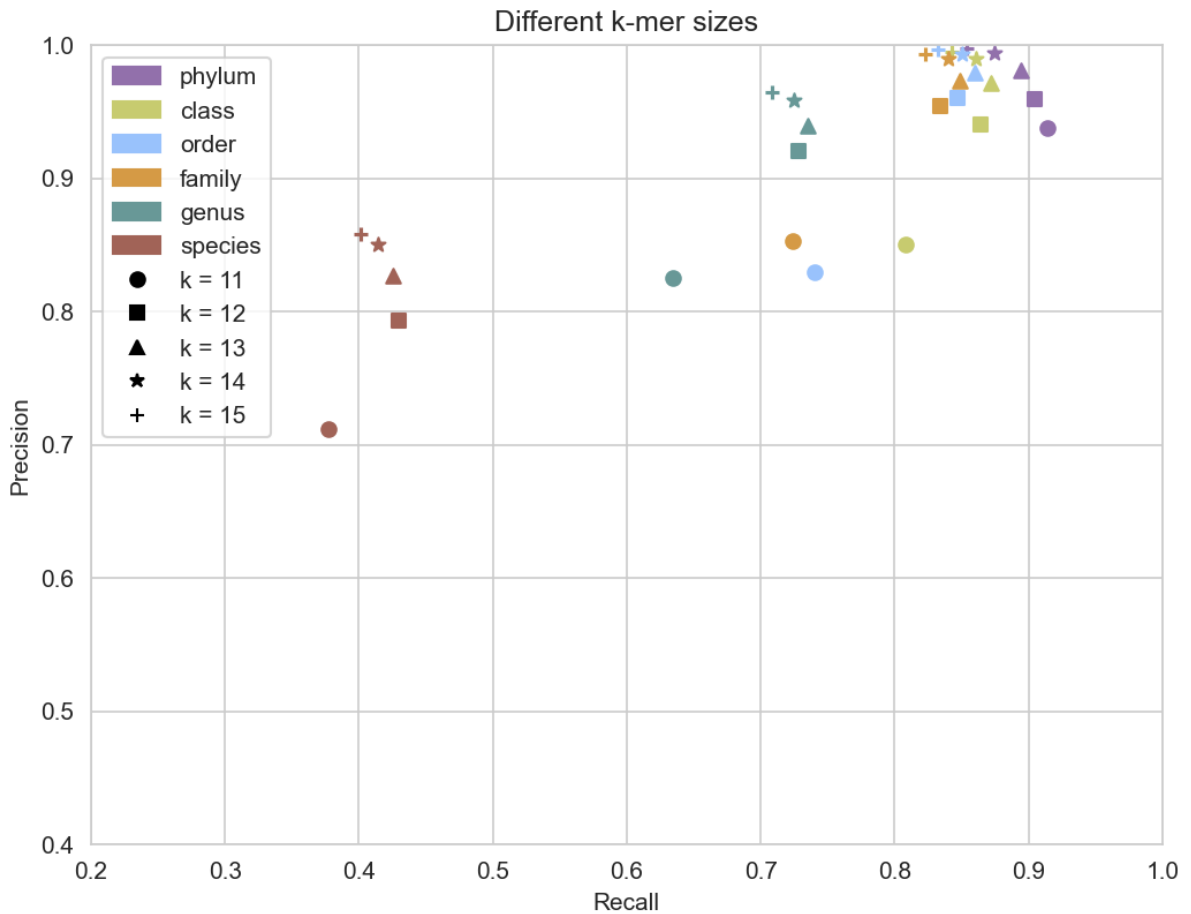


Figure 4.3: Comparison of different k-mer sizes for DIAMER. Read-based precision and recall are shown at different taxonomic ranks and with different k-mer lengths for the ZymoMock dataset with NR90 as the reference database. The Uniform11S reduced amino acid alphabet was used.

Precision and recall are both influenced by the k-mer size (see Figure 4.3). As expected, a larger k-mer increases the precision. This can be explained by the fact that a shorter exact sequence match is more likely to occur by chance than the match of a

longer sequence. However, the recall has an optimal k-mer length of $k = 12$ at species level and $k = 13$ for genus, family, order, and class. Again, this can be explained by the higher probability of finding a short k-mer match by chance. A shorter k-mer has a higher probability of finding a match in the database. This increases the chance that a read is classified, but at the same time, it increases the risk that it is assigned based on a random match. This tradeoff is optimal for the recall for $k = 12$ or $k = 13$.

For the newer ZymoOral dataset, with the most recent base call model and much higher read quality, the k-mer length does not seem to have a large influence on the results (see Figure A.3). However, there seems to be something wrong with the data for this plot, since the rank order has a lower recall than the ranks family and genus, which is not possible. Therefore, this result can not be trusted.

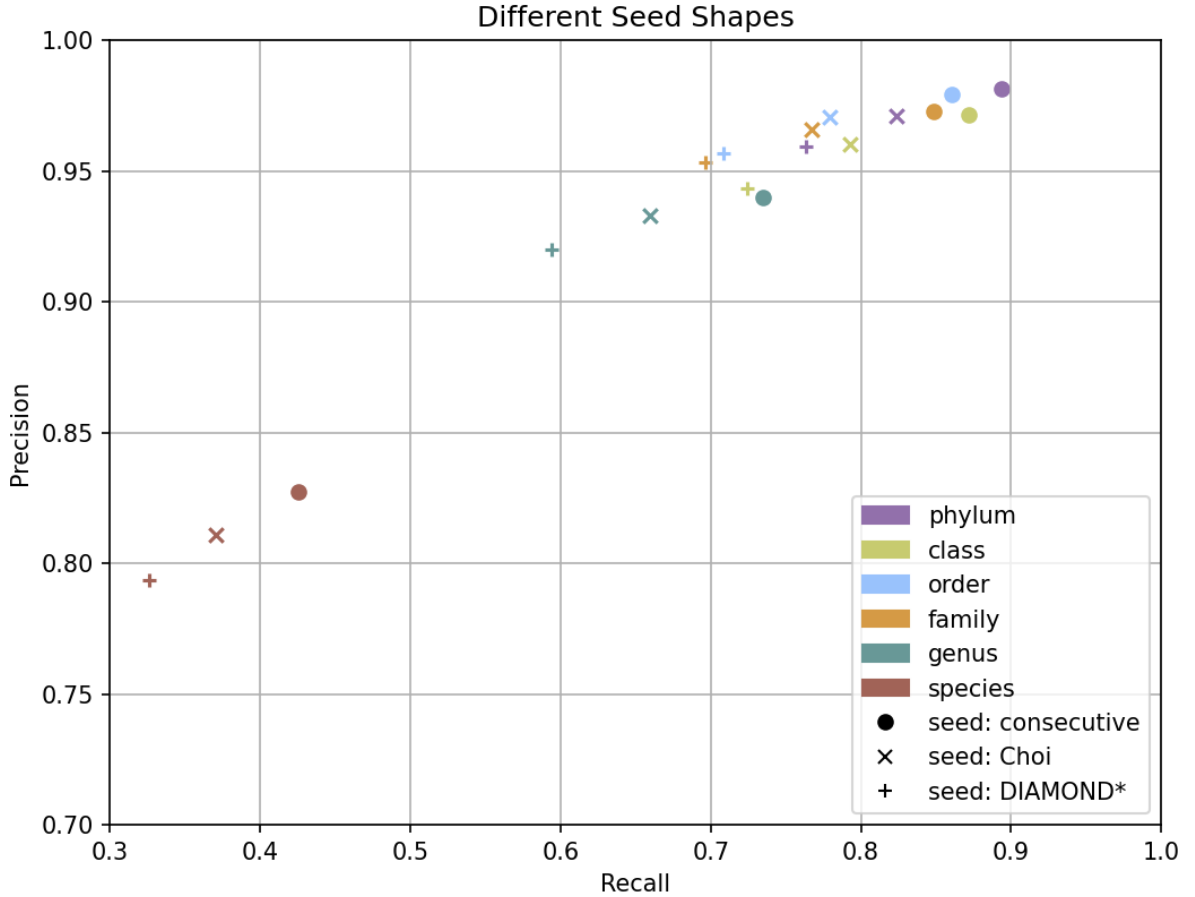


Figure 4.4: Comparison of spaced seeds of different shapes to a consecutive k-mer. The dataset ZymoMock was classified with the Uniform11S alphabet and the NR90 reference database, with different seed shapes of weight $w = 13$. The *consecutive* seed (circles) corresponds to a k-mer with $k = 13$. The seed *Choi*[60] (x) has length $l = 20$ and the following shape: 11101001101001110111. The seed *DIAMOND**[23] (+) has length $l = 25$ and shape: 1111001010000100100101111.

DIAMER supports the use of spaced seeds instead of consecutive k-mers. Spaced seeds are reported to increase sensitivity in homology search [28, 60], but also read classification on ranks higher than species [50]. Figure 2.1 compares two different spaced seed shapes with weight $w = 13$ to the consecutive k-mers of length $k = 13$ for classification of the ZymoMock dataset with NR90 and the Uniform11S alphabet. The seed shape labeled *Choi* in the figure was proposed by Choi et al. for homology search [60]. It has a length of $l = 20$ and is the best-scoring seed for the high-similarity dataset they used for comparison. The seed labeled *DIAMOND** is adapted from one of the seeds used by DIAMOND. To change it from the native $w = 12$ of DIAMOND to $w = 13$, a single 1 was added at the end.

Figure 2.1 shows worse precision and recall for both spaced seeds compared to the consecutive k-mers on all taxonomic levels. The performance seems to depend directly on the length of the shape, since the seed with $l = 25$ performs even worse than the seed with $l = 20$. This can be explained by the dominant errors for long-read sequences, which are insertions and deletions [11, 61]. Spaced seeds are ideal for dealing with differences in one character of the compared sequences (see Chapter 2.1.2), such as, for example, point mutations of DNA. However, frame-shifts caused by insertions and deletions can not be corrected by spaced seeds. The spaced seed with $l = 25$ spans 75 nucleotides at the DNA level. Hence, a single insertion or deletion at the DNA level potentially leads to only wrong seeds from a region spanning 149 nucleotides. This shows that spaced seeds are not suitable for translated long-read classification.

4.2.4 Reduced Amino Acid Alphabet

DIAMER uses a reduced amino acid alphabet that can be changed by the user for k-mer extraction. Figure 4.5 shows the result of nine different reduced amino acid alphabets in comparison. Surprisingly, the number of amino acid clusters in the alphabet does not affect performance much. Both the worst-performing alphabet (DIAMOND) and the alphabet with the highest recall (Uniform11S) have 11 amino acid clusters. The additional amino acid clusters in the alphabets Solis15, Solis15S, Solis16, and Uniform16 only seem to have a slight advantage regarding precision. This is surprising, since the information content of one k-mer is much higher for alphabets with more clusters. An 11-letter alphabet with a k-mer size of 13 requires only 45 bits per k-mer, while a 16-letter

alphabet with $k = 13$ uses all 52 bits available for k-mers in DIAMER.

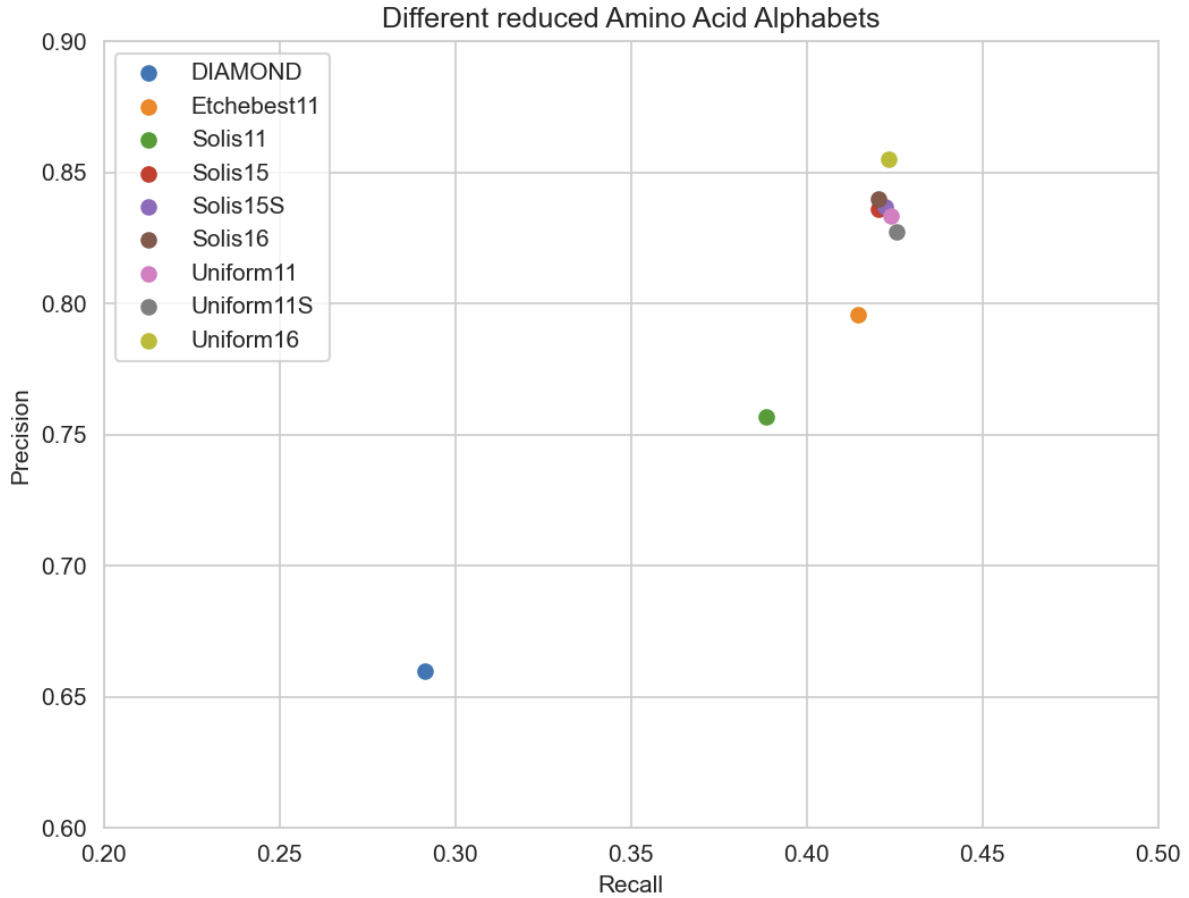


Figure 4.5: Comparison of different reduced amino acid alphabets for the classification of the ZymoMock dataset with NR90 and a k-mer of length $k = 13$ at species level. The amino acid clusters for the reduced amino acid alphabets can be found in Table 3.3. The numbers in the names of the alphabets indicate the number of amino acid clusters. The DIAMOND alphabet has 11 clusters.

Another interesting finding is that the alphabets by Etchebest et al. [57], Solis et al. [56], and Buchfink et al. [23], which were created based on biological data, are not better than the *uniform* alphabets that were created in this project. The uniform alphabets are not based on any biological observations, but only try to make each amino acid cluster equally likely to be found in NR. This leads to a more uniform distribution of the k-mers across the space of possible k-mers. Since not much information gets lost in one cluster with a high probability to occur (e.g., a cluster consisting of many likely amino acids), the uniform alphabets maximizes the information that can be encoded in one k-mer from a perspective of information theory. This would also explain the worse performance of the DIAMOND alphabet, which collects seven of the 20 standard amino acids in one

cluster. However, uniform alphabets might still not be perfect in preserving the biological information encoded in the amino acids. Alphabets based on biological data should be better at preserving the biological information. Consequently, based on the results in Figure 4.5, the preservation of biological information might not be as important for read classification as the preservation of sequence information in general. However, the strong performance of uniform alphabets compared to biology-based alphabets might also be due to the fact that the test dataset does not contain novel organisms. For the detection of similarity between novel sequences and similar sequences in the reference database, the preservation of biological data might be a more important property of a reduced amino acid alphabet.

4.2.5 K-mer Filtering, Minimizers & Index Size

DIAMER provides diverse options to filter the k-mers during database and read indexing. These options can help reduce the size of the database index. Figure 4.6 shows the recall of all filtering strategies depending on the database index size. Different parameters for the complexity and probability thresholds and the window sizes of minimizers and maximizers have been used. All filtering strategies, except for the probability filtering, seem to perform comparably well in reducing the database size of the NR90 index from 171 GB to about 100 GB, concerning the recall. The probability filtering performs the worst, which is caused by the alphabet. The Uniform11S alphabet used for this classification is designed specifically to reduce differences between the probabilities of different amino acid clusters. Selecting k-mers with low probability as minimizers might not have the desired effect of selecting k-mers with higher statistical significance in this case. For index size reductions below 100 GB, the complexity maximizer performs better than any other method. With complexity maximization in a window of size $o = 15$, the database is only half the original size, and the recall drops by less than 5%. Selecting k-mers that maximize the sequence complexity is the best choice for reducing the size of database indexes.

Figure 4.7 shows how the precision depends on the database size for different k-mer filtering strategies. Surprisingly, almost all filtering strategies lead to an increase in precision with a reduced database size. The only exception is the probability filtering, which could also be caused by the use of a uniform reduced amino acid alphabet. The observa-

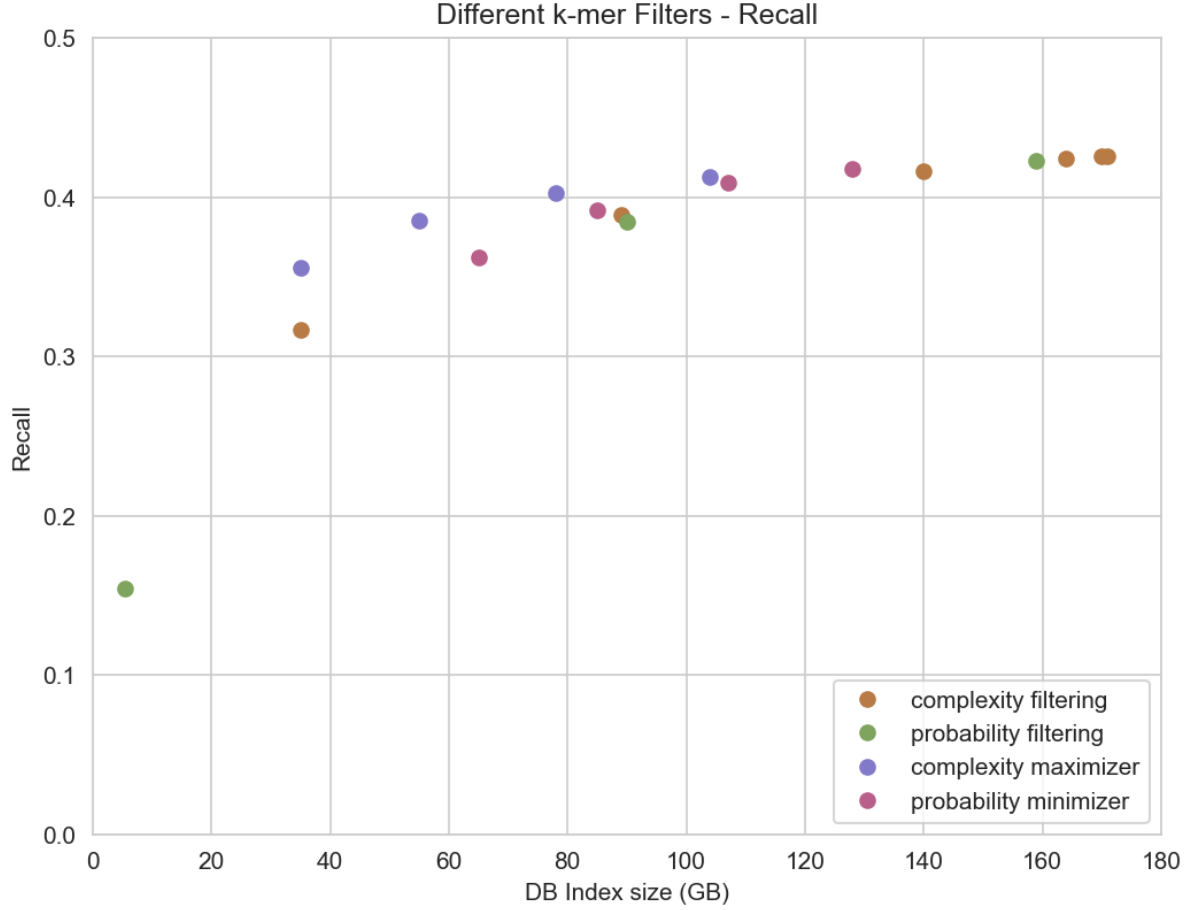


Figure 4.6: Read classification of the ZymoMock dataset with different k-mer filtering options using the Uniform11S alphabet, the NR90 reference database, and a k-mer size of 13. The read classification recall is plotted over the database index size at species level. Applied filtering thresholds for complexity c , probability p and the minimizer/maximizer window sizes o are from left to right: *complexity filtering* $c > [8, 7, 6, 5, 4, 3]$, *probability filtering* $p < [2 \cdot 10^{-14}, 3 \cdot 10^{-14}, 4 \cdot 10^{-14}]$, *complexity maximizer* $o = [21, 17, 15, 14]$, *probability minimizer* $o = [21, 17, 15, 14]$.

tion that a smaller database size increases precision is also described in the publication of Kraken 1 [22]. The effect results from the definition of precision. Since precision does not take into account the number of classified sequences, but only the ratio of true positives to total classifications, fewer classified sequences do not have a negative influence on this measure. With a smaller reference database, the probability of being able to classify a read decreases, which does not affect the precision. However, at the same time, the probability of wrong k-mer matches and classifications decreases, which has a positive influence on the precision. Therefore, reduced database sizes can help to increase the precision, which might be useful in scenarios where speed is more important than recall.

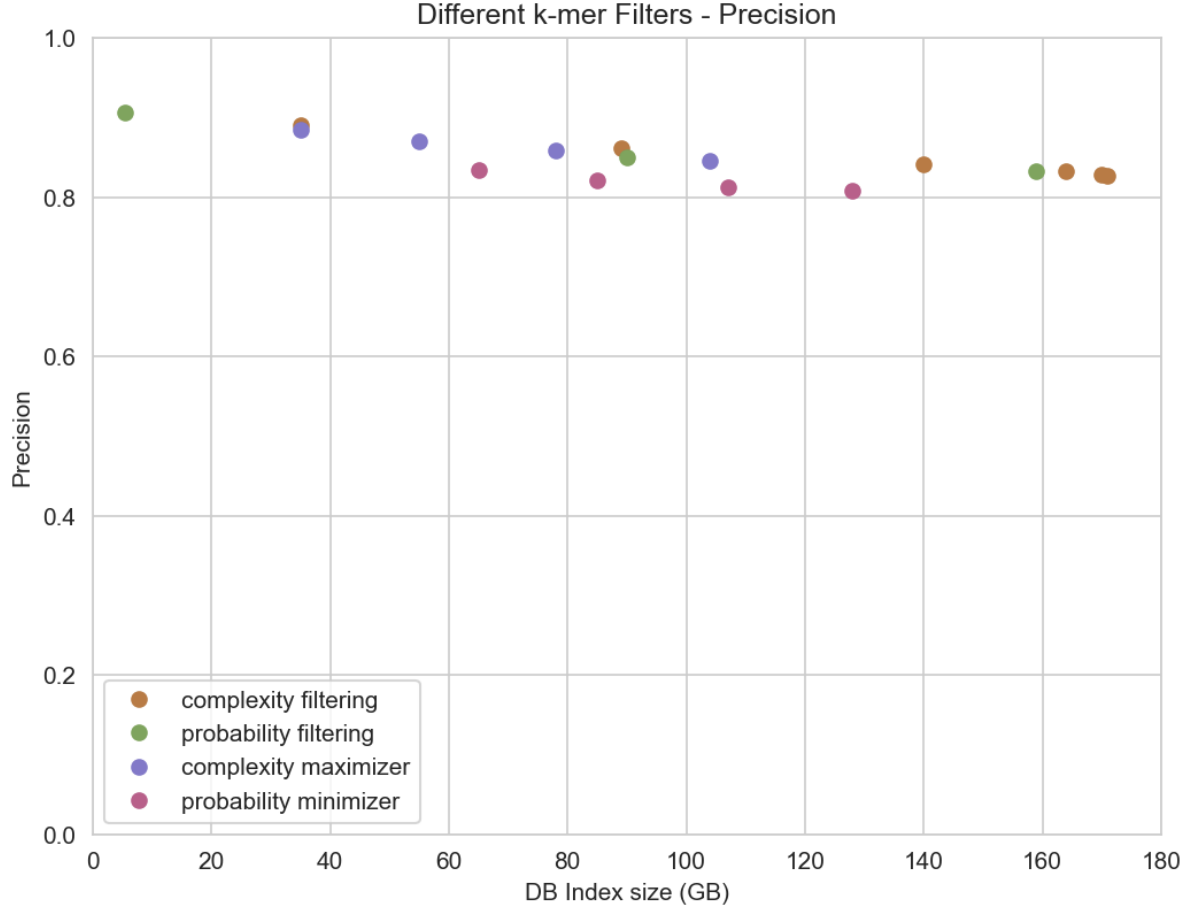


Figure 4.7: Read classification of the ZymoMock dataset with different k-mer filtering options using the Uniform11S alphabet, the NR90 reference database, and a k-mer size of 13. The read classification precision is plotted over the database index size at species level. Applied filtering thresholds for complexity c , probability p and the minimizer/maximizer window sizes o are from left to right: *complexity filtering* $c > [8, 7, 6, 5, 4, 3]$, *probability filtering* $p < [2 \cdot 10^{-14}, 3 \cdot 10^{-14}, 4 \cdot 10^{-14}]$, *complexity maximizer* $o = [21, 17, 15, 14]$, *probability minimizer* $o = [21, 17, 15, 14]$.

4.3 Performance and Application

In this section, the performance of DIAMER in terms of required processing power, RAM, and disk space will be evaluated on example runs. Based on these results, possible applications of DIAMER will be discussed.

4.3.1 Performance

DIAMER is able to analyze a dataset of 1.16M nanopore reads in under 20 minutes on a PC with 16 GB of RAM and a fast SSD, given a precomputed database index. This section will give an overview of the resources required to run the computation steps of

DIAMER on different platforms. The tool has a lot of options that can be changed by the user, especially for indexing tasks. All options influence the running time, required memory and disk space. Furthermore, not all steps of DIAMER are optimized for every platform. The database preprocessing and indexing should be run on a server with at least 500 GB available RAM, while read indexing and classification can be run on a modern laptop. The performance will be evaluated based on example runs on different platforms with specific parameters. For details on the platforms *PC* and *server*, please refer to Chapter 3.1.3.

Database Preprocessing & Indexing

Preprocessing and indexing require a lot of RAM to run in a reasonable time. Preprocessing the full NR with the *FULL* and the *dead* NCBI mapping file takes about 6.5 h on the *server* with ~700 GB of ram for the Java Virtual Machine. To reduce the high memory demand, DIAMER can use MEGAN mapping files for accession-to-taxonomic-ID mapping. This is possible for NR and the clustered NR50 and NR90. Since the SQLite database of MEGAN does not need to be kept in memory, this process requires less than 3 GB of RAM and can be run on a laptop. Preprocessing NR50 takes less than 1 h on the *PC*. Preprocessing NR90 on the *server* takes about 12 h and would probably not be slower on a PC or laptop (not tested). It has to be kept in mind that using a SQLite database makes the process strongly dependent on the speed of IO operations, which might slow down computation on the *server*. Since the database preprocessing step was not performed often during this project, not much time was invested in optimizing it. The command used to preprocess NR with the NCBI mapping files is the following:

```
java -jar diamer.jar --preprocess -no <nodes.dmp> -na <names.dmp>  
    <NR> <output file> <dead accession2taxid> <FULL accession2taxid>
```

The indexing process can, in theory, be run on any machine where at least one bucket can be kept in RAM. Indexing NR90 on the *PC* with 16 GB of RAM and 16 threads with the command below would take an estimated 100 h (not tested). The low RAM allows only five buckets to be collected at a time, and therefore DIAMER has to iterate 205 times over the database.

```
java -jar diamer.jar --indexdb -t 16 -b 5 --mask 11111111111111
--alphabet [L][A][GC][VWUBIZO*][SH][EMX][TY][RQ][DN][IF][PK]
-no <nodes.dmp> -na <names.dmp> <NR90> <index folder>
```

Indexing the whole NR on the *server* with 800 GB of RAM and 224 threads takes about 3 h with the following settings:

```
java -Xmx800g -jar diamer.jar --indexdb -b 128 -t 224
--keep-in-memory --mask 11111111111111
--alphabet [L][A][GC][VWUBIZO*][SH][EMX][TY][RQ][DN][IF][PK]
-no <nodes.dmp> -na <names.dmp> <NR> <index folder>
```

The clustered NR90 database only needs 20 minutes indexing time on the *server* with the following settings, since all 1024 buckets can be collected at once with 500 GB of RAM:

```
java -Xmx500g -jar diamer.jar --indexdb -t 224 -b 1024
--mask 1111111111111111
--alphabet [L][A][GC][VWUBIZO*][SH][EMX][TY][RQ][DN][IF][PK]
-no <nodes.dmp> -na <names.dmp> <NR90> <index folder>
```

The automatic or user-defined adjustment of how many buckets fit into RAM makes DIAMER very flexible for different computational environments. Preprocessing and indexing large databases on a server is possible in under 12 h and would even be possible on a PC. It should be noted that the required memory strongly depends on the k-mer size since the number of extracted k-mers increases for smaller values of k . The available options to filter k-mers can be used to reduce the memory requirements. Furthermore, the use of reduced amino acid alphabets with more amino acid clusters also requires more memory, since more unique k-mers are extracted.

Read Indexing

In contrast to the database preprocessing and indexing, the step of read indexing can be computed quickly without a server. Generating an index for the ZymoMock dataset on the *PC* with 12 GB RAM takes 13 minutes with the command below.

```
java -Xmx12g -jar diamer.jar --indexreads -t 16 -b 128
  --filtering c 0 --mask 11111111111111
  --alphabet [L][A][GC][VWUBIZO*][SH][EMX][TY][RQ][DN][IF][PK]
  <reads FASTQ> <index folder>
```

On the *server*, all buckets can be extracted in one run in less than 4 minutes using 100 GB of RAM and 64 threads:

```
java -Xmx100g -jar diamer.jar --indexreads -t 64 -b 1024
  --filtering c 0 --mask 11111111111111
  --alphabet [L][A][GC][VWUBIZO*][SH][EMX][TY][RQ][DN][IF][PK]
  <reads FASTQ> <index folder>
```

In the current version, the limiting factor for achieving even higher speeds on the *server* is the use of only one reader thread, even for uncompressed files. Speeding up the read speed will speed up the whole indexing process.

Read Classification

The only limiting factor for the speed of read classification is the maximum read speed of the disk on which the index files are stored. On the *PC*, collecting k-mer matches and running the default assignment algorithm on the index of the ZymoMock dataset and the full NR index with a k-mer size of 13 and the Uniform11s alphabet takes only 5 minutes with the following command:

```
java -Xmx12g -jar diamer.jar --assignreads -t 16
  <database index> <reads index> <output folder>
```

Comparing the 34 GB of the reads index to the 369 GB of the NR index is only possible with a very fast drive. In this case, the m.2 SSD of the *PC* has a maximum read speed of 2 GB/s.

Despite running on 64 threads, the exact same computation takes about 6 minutes on the *server*. The difference in computation time can be explained by a slower disk read

speed on the *server*.

```
java -Xmx50g -jar diamer.jar --assignreads -t 64  
    <database index> <reads index> <output folder>
```

The time DIAMER needs for the classification of reads almost entirely depends on the read speed of the disk the indexes are stored on. While the classification phase can not make full use of the resources provided by a high-performance server, it is adapted to the very fast SSDs of modern PCs and laptops and might be even faster on a PC than on a high-performance server.

4.3.2 Comparative Analysis

This chapter will compare the performance of DIAMER to Kraken 2X, since it is a widely used tool for read classification and its k-mer-based approach has a lot in common with DIAMER. For this comparison, DIAMER was used with an NR index with the Uniform11S reduced amino acid alphabet, $k = 13$, and default filtering. The following commands were used for classification with Kraken 2X:

```
kraken2-build --download-taxonomy --protein -db <name>  
kraken2-build --download-library nr --protein --db <name>  
kraken2-build --build --threads 64 --protein --db <name>  
kraken2 --db <name> <ZymoMock> > <output file>
```

Resource Requirements

DIAMER has two steps to get from the raw NR database to the reference database index that is required for classification. Both of these steps can be performed in under 12 h on a server with 1 TB or RAM. With default filtering, the NR index of DIAMER with $k = 13$ and the Uniform11S alphabet has a size of 369 GB. Kraken 2X needs about 12 h for the mapping and indexing process on 64 threads of the server. The resulting index file needs 140 GB on disk. While the recommended workflow for DIAMER is to use the *dead* and *FULL* mapping file of NCBI for processing 99% of sequences in NR, Kraken 2X only uses the default mapping file, which allows it to map about 54% of all sequences. For the actual classification of the reads of the ZymoMock dataset, Kraken 2x needs about 3 minutes with 16 threads and 139 GB of RAM on the *server*. DIAMER, on the other

hand, only needs 12 GB RAM and 16 threads to classify the ZymoMock dataset in 17 minutes on a PC or laptop. For this, DIAMER relies on a disk with fast read speeds. However, loading the 140 GB index file into memory makes the speed of Kraken 2X also dependent on disk speed. In conclusion, both tools need a lot of time and memory to prepare a database for indexing. Once the indexes are generated, DIAMER takes about six times as long to process the test dataset, but only needs 9% of the RAM needed by Kraken 2X. The database size of Kraken 2 can be reduced via hash-based sub-sampling for systems with less memory, however, this will decrease recall [26].

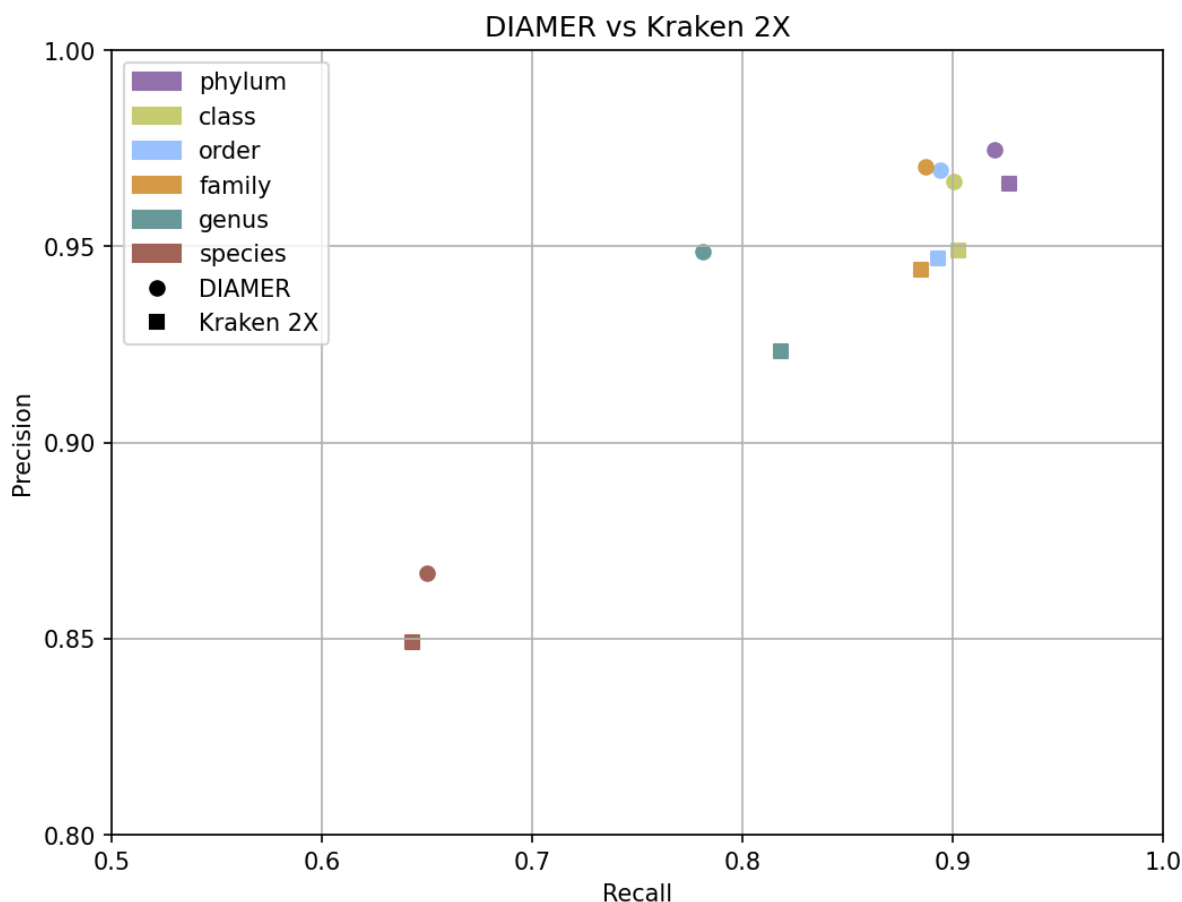


Figure 4.8: Comparison of the performance of Kraken 2X and DIAMER on the ZymoMock dataset with the NR reference database. Kraken 2X was run with default parameters. DIAMER used the Uniform11S alphabet with a k-mer size of 13 and default filtering.

Performance

A comparison of precision and recall for Kraken 2X and DIAMER is shown in Figure 4.8 for the ZymoMock dataset. DIAMER has a higher precision at all taxonomic levels, and

a higher recall at species level. Kraken 2X performs better at genus level recall and a little better at phylum level recall. The advantage of Kraken 2X for the recall at the genus and phylum level can be explained by the smaller k-mer size. Similarly, the higher precision of DIAMER might depend on the longer k-mers. It should be noted that DIAMER was developed with ZymoMock as a test dataset. Therefore, the settings chosen for this comparison might be tailored to produce good results specifically for this dataset. For the much newer ZymoOral dataset with higher read quality, Kraken 2X performs slightly better than DIAMOND (see Figure A.4).

4.3.3 Application

In its current state, DIAMER provides many customization options. This allows for the exploration of different reduced amino acid alphabets, different k-mer sizes, spaced seeds, and filtering techniques for the application in translated read classification. DIAMER can replace Kraken 2X in pipelines where long-read classification is necessary. However, because of the many customization options, the separation of database preprocessing and indexing, and no automatic download of the reference database and taxonomy, DIAMER is not as user-friendly as Kraken 2X. Nevertheless, in scenarios where only a limited amount of RAM is available, DIAMER might be the only option to classify long-read sequencing data with similar, or even better, precision and recall compared to Kraken 2X. Furthermore, a precomputed index for popular reference databases could be provided as a download, so that the computationally most expensive steps can be avoided. Consequently, a long-read dataset could be classified completely without the need for a supercomputer or high-performance cluster.

Chapter 5

Discussion and Outlook

Our lives depend on microbes, and with the ongoing development of DNA sequencing techniques, metagenomic research will become even more important tomorrow than it is today. The rapid growth of reference databases will enable researchers to analyze and annotate metagenomic samples in increasing detail and resolution. However, larger reference databases pose a challenge for existing tools for metagenomic analysis. The memory requirement for most programs that can be used for metagenomic read classification exceeds the resources of most PCs and Laptops, making their use dependent on high-performance computing. DIAMER provides an alternative to an expensive computing infrastructure. With its double indexing strategy, the tool can classify long reads in under 20 minutes on a laptop with 12 GB RAM while keeping precision and recall equal to the well-established Kraken 2X, which needs 11 times more memory for the same task.

5.1 Limitations

Despite its good performance in comparison with Kraken 2X, DIAMER is far from perfect. Further development and testing are needed to develop the concepts that have been shown to work in this thesis to a state where others can benefit from the tool.

The largest problem with the current version of DIAMER is insufficient testing. The tool has so far only been tested by one person and with only two test datasets. Much more testing is needed to make sure that no unexpected behavior influences the results and to ensure functionality across different datasets. Species exclusion experiments are an important test to ensure proper function in scenarios where a lot of unknown organisms

are expected in a sample. The use of only two datasets also reduces the significance of the parameter-search-related and resource requirement test results. Different datasets may result in completely different optimal parameters, or might require more processing time or memory.

Furthermore, DIAMER is not the most user-friendly software. The many options to customize index generation are needed to test different strategies, but might overload users with less experience in bioinformatics. The two steps required to build a database index are not ideal, and the restriction to FASTQ input files and the NCBI taxonomy are also suboptimal. Furthermore, a lot of features that were proven not to help the classification are still included in DIAMER. These are, for example, the probability filtering, the support of spaced seeds, and the normalization of k-mer counts.

Finally, DIAMER does not make use of the full potential that the algorithmic approach presented in this thesis has. A lot of algorithms can be further optimized, and some Java classes could be replaced by faster primitive data structures. Poor optimization for edge cases already shows during the use of very short k-mers or the use of reduced amino acid alphabets with many clusters. This implementation might be well suited to try out different algorithms, parameters, and software designs, but it is not yet ready for broad application.

5.2 Key Insights

The software DIAMER shows that the RAM required for read classification can be reduced drastically via the double indexing strategy of DIAMOND. Using the fast SSDs, which are part of any modern computer, metagenomic long-read shotgun sequences can be assigned to taxa with only 12 GB of RAM in reasonable time and with better precision compared to an established tool in the field. The parallelization of many processing steps and the ability to process the index in small batches (buckets) was shown to make use of the computing power available on a high-performance server, as well as a PC. While still not being perfect, the algorithmic approach was proven to be applicable to the problem of read classification.

Furthermore, the application of DIAMER to datasets of metagenomic mock communities could uncover which k-mer and seed-based bioinformatics methods help with the

task of translational long-read classification. It was shown that small reference databases are enough to classify a subset of reads at very high precision, which is consistent with observations during the development of Kraken [22]. Additionally, DIAMER showed that the maximization of k-mer complexity is the best strategy to filter k-mers in that setting. The superior recall that can be achieved by reducing the size of the reference database index with complexity maximizers suggests that this approach might be better at retaining important information compared to random, hash-based sub-sampling, which is performed by Kraken for the reduction of database size [26]. Further experiments need to be performed to compare the performance of a reduced Kraken database to a reduced database of DIAMER. Another surprising result is that spaced seeds can not help with protein-based long-read classification. This might be because insertions and deletions are the predominant sequencing errors in long-read technologies. The choice of a reduced amino acid alphabet has a huge influence on the performance of this type of classification. Surprisingly, the performance of different alphabets can not be explained by the number of amino acid clusters alone. The equal performance of uniform alphabets and biology-derived alphabets implies that the preservation of biological information might not be more important than the preservation of the sequence information in general. This should be confirmed with further experiments using different datasets.

This project proves that the concept of double indexing can be utilized efficiently for the task of translational long-read classification and, at the same time, gives some important hints on what bioinformatics strategies to use for this problem.

5.3 Future Directions

Even though DIAMER might not be very user-friendly, it could be used to replace Kraken 2X in certain scenarios. With a pre-calculated reference database index, it can classify reads on less powerful devices. Furthermore, the modularity and customizability of DIAMER can be used for further testing of what parameters are optimal for translational long-read classification. The insights provided by this thesis, together with additional tests on other datasets, can be used to improve DIAMER in the future. Additionally, the test results for the different strategies that have been tested in this thesis can be used to focus the future development of this tool on the features that have been shown

to improve performance, such as the complexity filtering and the uniform reduced amino acid alphabets.

The code developed in this project can be used as a base to further improve the indexing and classification speed. Several suggestions for possible changes that will lead to faster running time and lower memory consumption have been discussed in Chapter 3.

Because of the double-indexing setup of DIAMER, only its running time depends on the size of the reference database. The required RAM is not affected as long as one bucket fits in memory. This property can help to keep up with growing reference databases and will aid metagenomics studies with third-generation sequencing to analyze samples and make microbes work for - and not against us.

Chapter 6

Availability

All code generated in this project can be found on the github repository of DIAMER <https://github.com/husonlab/diamer1>. This includes the Jupyter Notebooks that were used to generate the plots. However, the data used in the plots is too large to be uploaded anywhere. The data for the plots can be found on the IBMI server of the university of Tübingen (/ceph/ibmi/ab/projects/kubach/thesis/dataset) or can be requested from me (<mailto:noel.kubach@student.uni-tuebingen.de>).

References

- [1] P. J. Turnbaugh, R. E. Ley, M. Hamady, C. M. Fraser-Liggett, R. Knight, and J. I. Gordon, “The human microbiome project,” *Nature* 2007 449:7164, vol. 449, pp. 804–810, 10 2007.
- [2] A. Soumare, A. G. Diedhiou, M. Thuita, M. Hafidi, Y. Ouhdouch, S. Gopalakrishnan, and L. Kouisni, “Exploiting biological nitrogen fixation: A route towards a sustainable agriculture,” *Plants* 2020, Vol. 9, Page 1011, vol. 9, p. 1011, 8 2020.
- [3] N. Ziemert, M. Alanjary, and T. Weber, “The evolution of genome mining in microbes – a review,” *Natural Product Reports*, vol. 33, pp. 988–1005, 7 2016.
- [4] J. C. Wooley, A. Godzik, and I. Friedberg, “A primer on metagenomics,” *PLOS Computational Biology*, vol. 6, p. e1000667, 2010.
- [5] D. J. Lane, B. Pace, G. J. Olsen, D. A. Stahl, M. L. Sogin, and N. R. Pace, “Rapid determination of 16s ribosomal rna sequences for phylogenetic analyses,” *Proceedings of the National Academy of Sciences of the United States of America*, vol. 82, p. 6955, 1985.
- [6] N. R. Pace, D. A. Stahl, D. J. Lane, and G. J. Olsen, *The Analysis of Natural Microbial Populations by Ribosomal RNA Sequences*, vol. 9, pp. 1–55. Springer, Boston, MA, 1986.
- [7] S. Albright and S. Louca, “Trait biases in microbial reference genomes,” *Scientific Data*, vol. 10, 12 2023.
- [8] J. A. Eisen, “Environmental shotgun sequencing: Its potential and challenges for studying the hidden world of microbes,” *PLOS Biology*, vol. 5, p. e82, 3 2007.

- [9] D. H. Huson, A. F. Auch, J. Qi, and S. C. Schuster, “Megan analysis of metagenomic data,” *Genome Research*, vol. 17, pp. 377–386, 3 2007. MEGAN is a program to explore metagenomic datasets as a post processing step after alignment. The results of the first version only take into account the hit number between reads and taxa.
- [10] J. Handelsman, M. R. Rondon, S. F. Brady, J. Clardy, and R. M. Goodman, “Molecular biological access to the chemistry of unknown soil microbes: a new frontier for natural products,” *Chemistry & biology*, vol. 5, 1998. First use of the term ”metagenomics”.
- [11] D. H. Huson, B. Albrecht, C. Bağci, I. Bessarab, A. Górski, D. Jolic, and R. B. Williams, “Megan-lr: New algorithms allow accurate binning and easy interactive exploration of metagenomic long reads and contigs,” *Biology Direct*, vol. 13, pp. 1–17, 4 2018.
- [12] R. Hamilton and S. Stephen, “Long-read sequencing for metagenomics in microbiology,” *Diagnostic Molecular Pathology: A Guide to Applied Molecular Testing, Second Edition*, pp. 39–48, 1 2024.
- [13] M. B. Scholz, C. C. Lo, and P. S. Chain, “Next generation sequencing and bioinformatic bottlenecks: the current state of metagenomic data analysis,” *Current Opinion in Biotechnology*, vol. 23, pp. 9–15, 2 2012.
- [14] J. Jovel, J. Patterson, W. Wang, N. Hotte, S. O’Keefe, T. Mitchel, T. Perry, D. Kao, A. L. Mason, K. L. Madsen, and G. K. Wong, “Characterization of the gut microbiome using 16s or shotgun metagenomics,” *Frontiers in Microbiology*, vol. 7, p. 180723, 4 2016.
- [15] D. M. Portik, C. T. Brown, and N. T. Pierce-Ward, “Evaluation of taxonomic classification and profiling methods for long-read shotgun metagenomic sequencing datasets,” *BMC Bioinformatics*, vol. 23, 2022.
- [16] K. Blin, S. Shaw, L. Vader, J. Szenei, Z. Reitz, H. Augustijn, J. D. Cedié-Becerra, V. de Crécy Lagard, R. Koetsier, S. Williams, P. Cruz-Morales, S. Wongwas, A. Segurado Luchsinger, F. Biermann, A. Korenskaia, M. Zdouc, D. Meijer, B. Terlouw, J. J. van der Hooft, N. Ziemert, E. N. Helfrich, J. Masschelein, C. Corre, M. Chevrette,

- G. van Wezel, M. Medema, and T. Weber, “antismash 8.0: extended gene cluster detection capabilities and analyses of chemistry, enzymology, and regulation,” *Nucleic Acids Research*, p. gkaf334, 4 2025.
- [17] M. D. Mungan, M. Alanjary, K. Blin, T. Weber, M. H. Medema, and N. Ziemert, “Arts 2.0: feature updates and expansion of the antibiotic resistant target seeker for comparative genome mining,” *Nucleic Acids Research*, vol. 48, pp. W546–W552, 7 2020.
- [18] S. J. Biller, P. M. Berube, K. Dooley, M. Williams, B. M. Satinsky, T. Hackl, S. L. Hogle, A. Coe, K. Bergauer, H. A. Bouman, T. J. Browning, D. D. Corte, C. Hassler, D. Hulston, J. E. Jacquot, E. W. Maas, T. Reinthaler, E. Sintes, T. Yokokawa, and S. W. Chisholm, “Marine microbial metagenomes sampled across space and time,” *Scientific Data 2018 5:1*, vol. 5, pp. 1–7, 9 2018.
- [19] H. Doré, U. Guyet, J. Leconte, G. K. Farrant, B. Alric, M. Ratin, M. Ostrowski, M. Ferrieux, L. Brillet-Guéguen, M. Hoebeke, J. Siltanen, G. L. Corguillé, E. Corre, P. Wincker, D. J. Scanlan, D. Eveillard, F. Partensky, and L. Garczarek, “Differential global distribution of marine picocyanobacteria gene clusters reveals distinct niche-related adaptive strategies,” *The ISME Journal*, vol. 17, pp. 720–732, 5 2023.
- [20] J. Marić, K. Križanović, S. Riondet, N. Nagarajan, and M. Šikić, “Comparative analysis of metagenomic classifiers for long-read sequencing datasets,” *BMC Bioinformatics*, vol. 25, p. 15, 12 2024.
- [21] D. J. Nasko, S. Koren, A. M. Phillippy, and T. J. Treangen, “Refseq database growth influences the accuracy of k-mer-based lowest common ancestor species identification,” *Genome Biology*, vol. 19, pp. 1–10, 10 2018.
- [22] D. E. Wood and S. L. Salzberg, “Kraken: Ultrafast metagenomic sequence classification using exact alignments,” *Genome Biology*, vol. 15, pp. 1–12, 3 2014.
- [23] B. Buchfink, C. Xie, and D. H. Huson, “Fast and sensitive protein alignment using diamond,” *Nature Methods 2014 12:1*, vol. 12, pp. 59–60, 11 2014.
- [24] S. F. Altschul, W. Gish, W. Miller, E. W. Myers, and D. J. Lipman, “Basic local alignment search tool,” *Journal of Molecular Biology*, vol. 215, pp. 403–410, 10 1990.

- [25] M. Steinegger and J. Söding, “Mmseqs2 enables sensitive protein sequence searching for the analysis of massive data sets,” *Nature Biotechnology* 2017 35:11, vol. 35, pp. 1026–1028, 10 2017.
- [26] D. E. Wood, J. Lu, and B. Langmead, “Improved metagenomic analysis with kraken 2,” *Genome Biology*, vol. 20, pp. 1–13, 11 2019.
- [27] B. D. Ondov, T. J. Treangen, P. Melsted, A. B. Mallonee, N. H. Bergman, S. Koren, and A. M. Phillippy, “Mash: Fast genome and metagenome distance estimation using minhash,” *Genome Biology*, vol. 17, pp. 1–14, 6 2016.
- [28] B. Ma, J. Tromp, and M. Li, “Patternhunter: faster and more sensitive homology search,” *Bioinformatics*, vol. 18, pp. 440–445, 3 2002.
- [29] U. Keich, M. Li, B. Ma, and J. Tromp, “On spaced seeds for similarity search,” *Discrete Applied Mathematics*, vol. 138, pp. 253–263, 4 2004.
- [30] M. Roberts, W. Hayes, B. R. Hunt, S. M. Mount, and J. A. Yorke, “Reducing storage requirements for biological sequence comparison,” *Bioinformatics*, vol. 20, pp. 3363–3369, 12 2004.
- [31] C. Chothia and A. M. Lesk, “The relation between the divergence of sequence and structure in proteins,” *The EMBO Journal*, vol. 5, pp. 823–826, 4 1986.
- [32] P. G. Wolynes, “As simple as can be?,” *researchgate.net*, vol. 4, pp. 871–874, 11 1997.
- [33] H. S. Chan, “Folding alphabets,” *Nature Structural Biology* 1999 6:11, vol. 6, pp. 994–996, 1999.
- [34] P. D. Thomas and K. A. Dill, “An iterative method for extracting energy-like quantities from protein structures communicated by,” *Biophysics*, vol. 93, pp. 11628–11633, 10 1996.
- [35] L. R. Murphy, A. Wallqvist, and R. M. Levy, “Simplified amino acid alphabets for protein fold recognition and implications for folding,” *Protein Engineering, Design and Selection*, vol. 13, pp. 149–152, 3 2000.

- [36] E. L. Peterson, J. Kondev, J. A. Theriot, and R. Phillips, “Reduced amino acid alphabets exhibit an improved sensitivity and selectivity in fold assignment,” *Bioinformatics*, vol. 25, p. 1356, 6 2009.
- [37] Y. Liang, S. Yang, L. Zheng, H. Wang, J. Zhou, S. Huang, L. Yang, and Y. Zuo, “Research progress of reduced amino acid alphabets in protein analysis and prediction,” *Computational and Structural Biotechnology Journal*, vol. 20, pp. 3503–3510, 1 2022.
- [38] Y. Ye, J. H. Choi, and H. Tang, “Rapsearch: A fast protein similarity search tool for short reads,” *BMC Bioinformatics*, vol. 12, pp. 1–10, 5 2011.
- [39] Y. Zhao, H. Tang, and Y. Ye, “Rapsearch2: a fast and memory-efficient protein similarity search tool for next-generation sequencing data,” *Bioinformatics*, vol. 28, pp. 125–126, 1 2012.
- [40] C. R. Woese and G. E. Fox, “Phylogenetic structure of the prokaryotic domain: the primary kingdoms,” *Proceedings of the National Academy of Sciences of the United States of America*, vol. 74, p. 5088, 1977.
- [41] N. A. O’Leary, M. W. Wright, J. R. Brister, S. Ciuffo, D. Haddad, R. McVeigh, B. Rajput, B. Robbertse, B. Smith-White, D. Ako-Adjei, A. Astashyn, A. Badretdin, Y. Bao, O. Blinkova, V. Brover, V. Chetvernin, J. Choi, E. Cox, O. Ermolaeva, C. M. Farrell, T. Goldfarb, T. Gupta, D. Haft, E. Hatcher, W. Hlavina, V. S. Joardar, V. K. Kodali, W. Li, D. Maglott, P. Masterson, K. M. McGarvey, M. R. Murphy, K. O’Neill, S. Pujar, S. H. Rangwala, D. Rausch, L. D. Riddick, C. Schoch, A. Shkeda, S. S. Storz, H. Sun, F. Thibaud-Nissen, I. Tolstoy, R. E. Tully, A. R. Vatsan, C. Wallin, D. Webb, W. Wu, M. J. Landrum, A. Kimchi, T. Tatusova, M. DiCuccio, P. Kitts, T. D. Murphy, and K. D. Pruitt, “Reference sequence (refseq) database at ncbi: current status, taxonomic expansion, and functional annotation,” *Nucleic acids research*, vol. 44, pp. D733–D745, 2016.
- [42] D. H. Parks, M. Chuvochina, D. W. Waite, C. Rinke, A. Skarszewski, P. A. Chaumeil, and P. Hugenholtz, “A standardized bacterial taxonomy based on genome phylogeny substantially revises the tree of life,” *Nature Biotechnology* 2018 36:10, vol. 36, pp. 996–1004, 8 2018.

- [43] N. Segata, L. Waldron, A. Ballarini, V. Narasimhan, O. Jousson, and C. Huttenhower, “Metagenomic microbial community profiling using unique clade-specific marker genes,” *Nature Methods* 2012 9:8, vol. 9, pp. 811–814, 6 2012.
- [44] S. Sunagawa, D. R. Mende, G. Zeller, F. Izquierdo-Carrasco, S. A. Berger, J. R. Kultima, L. P. Coelho, M. Arumugam, J. Tap, H. B. Nielsen, S. Rasmussen, S. Brunak, O. Pedersen, F. Guarner, W. M. D. Vos, J. Wang, J. Li, J. Doré, S. D. Ehrlich, A. Stamatakis, and P. Bork, “Metagenomic species profiling using universal phylogenetic marker genes,” *Nature Methods* 2013 10:12, vol. 10, pp. 1196–1199, 10 2013.
- [45] T. A. K. Freitas, P. E. Li, M. B. Scholz, and P. S. Chain, “Accurate read-based metagenome characterization using a hierarchical suite of unique signatures,” *Nucleic Acids Research*, vol. 43, pp. e69–e69, 5 2015.
- [46] R. Ounit, S. Wanamaker, T. J. Close, and S. Lonardi, “Clark: fast and accurate classification of metagenomic and genomic sequences using discriminative k-mers,” *BMC Genomics*, vol. 16, pp. 1–13, 3 2015. CLARK: extracts all k-mers from DB but only keeps the unique ones within one taxonomic rank (requires building of multiple indexes for multiple ranks). Assignment is based on only the target with the highest k-mer hit count.
- [47] P. Menzel, K. L. Ng, and A. Krogh, “Fast and sensitive taxonomic classification for metagenomics with kaiju,” *Nature Communications* 2016 7:1, vol. 7, pp. 1–9, 4 2016.
- [48] G. L. Rosen, E. R. Reichenberger, and A. M. Rosenfeld, “Nbc: the naïve bayes classification tool webserver for taxonomic classification of metagenomic reads,” *Bioinformatics*, vol. 27, pp. 127–129, 1 2011.
- [49] K. Sedlar, K. Kupkova, and I. Provaznik, “Bioinformatics strategies for taxonomy independent binning and visualization of sequences in shotgun metagenomics,” *Computational and Structural Biotechnology Journal*, vol. 15, pp. 48–55, 1 2017.
- [50] K. Brinda, M. Sykulski, and G. Kucherov, “Spaced seeds improve k-mer-based metagenomic classification,” *Bioinformatics*, vol. 31, pp. 3584–3592, 11 2015.

- [51] S. Lindgreen, K. L. Adair, and P. P. Gardner, “An evaluation of the accuracy and speed of metagenome analysis tools,” *Scientific Reports 2016 6:1*, vol. 6, pp. 1–14, 1 2016.
- [52] S. H. Ye, K. J. Siddle, D. J. Park, and P. C. Sabeti, “Benchmarking metagenomics tools for taxonomic classification,” *Cell*, vol. 178, pp. 779–794, 8 2019.
- [53] J. Fischer, “A novel k-mer algorithmic approach for taxonomic classification of metagenomic long reads,” Master’s thesis, Eberhard Karls Universität Tübingen, 5 2024.
- [54] J. Quick, “Assessing ultra-deep, long-read metagenomics on oxford nanopore promethion,” 9 2018.
- [55] S. Heidelberg, S. M. Dall, J. S. Bøjer, J. Nissen, L. N. van der Maas, M. Sereika, R. H. Kirkegaard, S. I. Jensen, S. J. Kousgaard, O. Thorlacius-Ussing, K. Hose, T. D. Nielsen, and M. Albertsen, “Nanomotif: Leveraging dna methylation motifs for genome recovery and host association of plasmids in metagenomes from complex microbial communities,” *bioRxiv*, p. 2024.04.29.591623, 1 2025. study where my zymo oral dataset is from.
- [56] A. D. Solis, “Amino acid alphabet reduction preserves fold information contained in contact interactions in proteins,” *Proteins: Structure, Function and Bioinformatics*, vol. 83, pp. 2198–2216, 12 2015.
- [57] C. Etchebest, C. Benros, A. Bornot, A. C. Camproux, and A. G. D. Brevern, “A reduced amino acid alphabet for understanding and designing protein adaptation to mutation,” *European Biophysics Journal*, vol. 36, pp. 1059–1069, 11 2007.
- [58] Y. Nishimura and S. Yoshizawa, “The oceandna mag catalog contains over 50,000 prokaryotic genomes originated from various marine environments,” *Scientific Data 2022 9:1*, vol. 9, pp. 1–11, 6 2022.
- [59] A. Zaky, S. J. Glastras, M. Y. W. Wong, C. A. Pollock, S. Saad, A. . Zaky, S. J. . Glastras, M. Y. W. . Wong, C. A. . Pollock, and S. Saad, “The role of the gut microbiome in diabetes and obesity-related kidney disease,” *International Journal of Molecular Sciences 2021, Vol. 22, Page 9641*, vol. 22, p. 9641, 9 2021.

- [60] K. P. Choi, F. Zeng, and L. Zhang, “Good spaced seeds for homology search,” *Bioinformatics*, vol. 20, pp. 1053–1059, 5 2004.
- [61] E. Sacristán-Horcajada, S. G.-D. L. Fuente, R. Peiró-Pastor, F. Carrasco-Ramiro, R. Amils, J. M. Requena, J. Berenguer, and B. Aguado, “Aramis: From systematic errors of ngs long reads to accurate assemblies,” *Briefings in Bioinformatics*, vol. 22, p. bbab170, 11 2021.

Appendix A

Supplementary Information

A.1 Use of AI Methods

Name	Version	Purpose
GitHub Copilot plugin	243	code completion
Overleaf (free version)	-	spell/grammar correction
Grammarly browser extension (free version)	8.929.0	spell/grammar correction

Table A.1: List of all AI tools that were used for this project. No version for Overleaf could be found. Overleaf and Grammarly are not transparent about whether AI is used for spell and grammar correction in the free plan.

A.2 Supplementary Figures

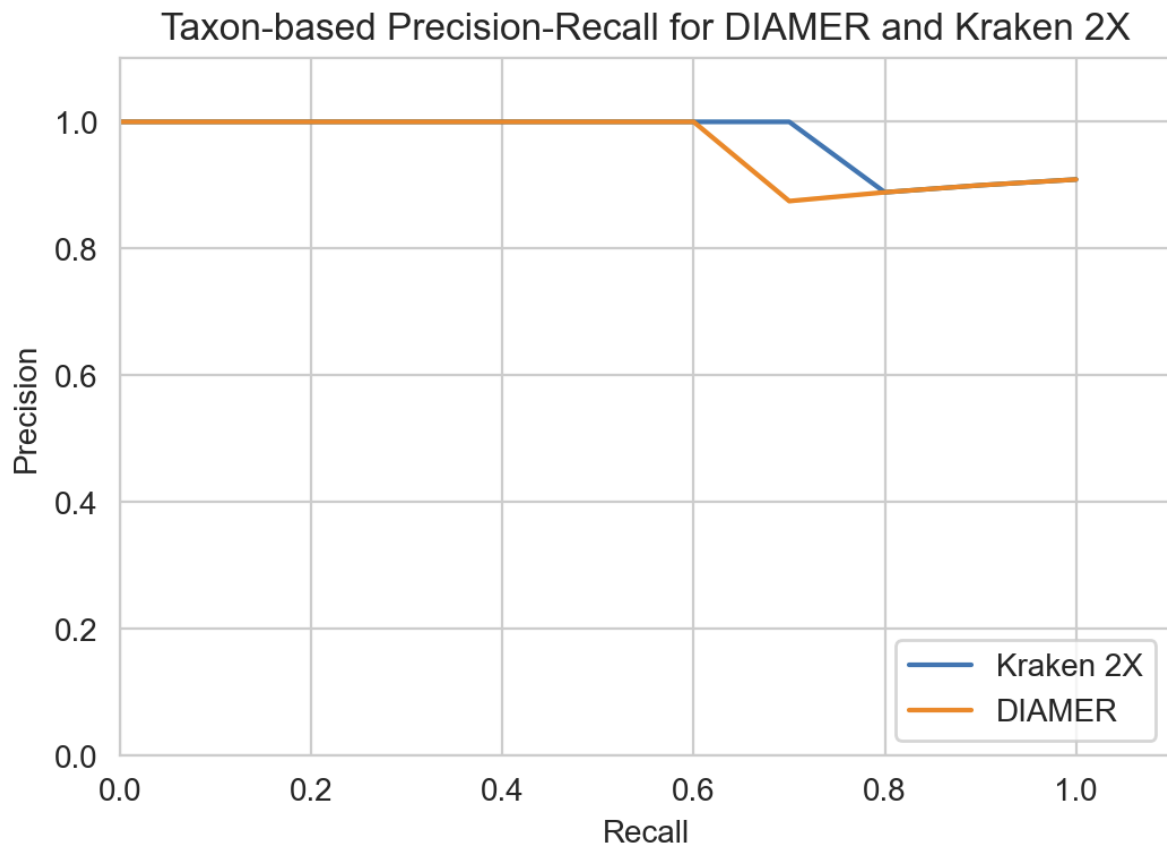


Figure A.1: Plot with taxon-based precision-recall calculation for different threshold values at species level. The reads of the ZymoMock dataset were classified based on accumulated k-mer counts from Kraken 2X and DIAMER with NR as reference database. The threshold parameter for the one-vs-one OVO algorithm was $t = 0.4$. This kind of plot only provides limited information about the performance of each tool.

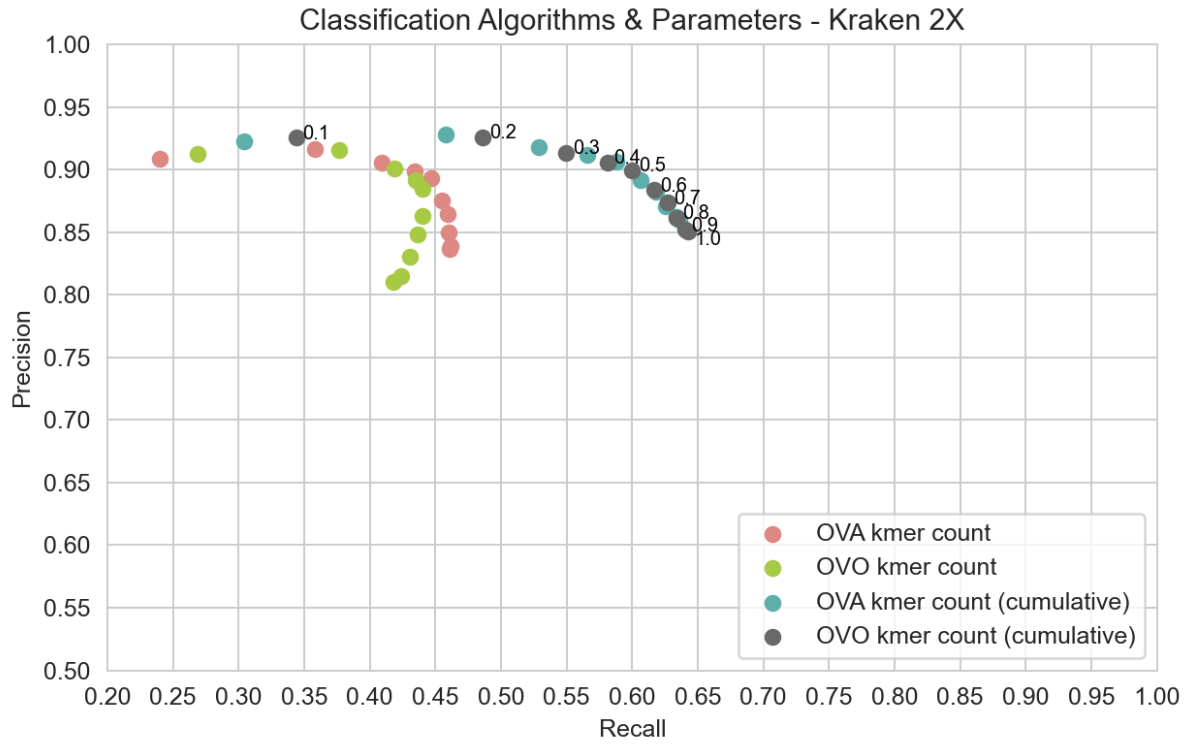


Figure A.2: Comparison of different classification algorithms for the k-mer matches produced by Kraken 2X with the NR database as reference. The one-vs-one (OVO) and one-vs-all (OVA) algorithms were used to classify the reads of the ZymoMock dataset based on weighted subtrees of the raw k-mer match counts or the accumulated k-mer match count. For both algorithms, different thresholds from $t = 0.1$ to $t = 1$ were used, as indicated for the OVO algorithm in gray.

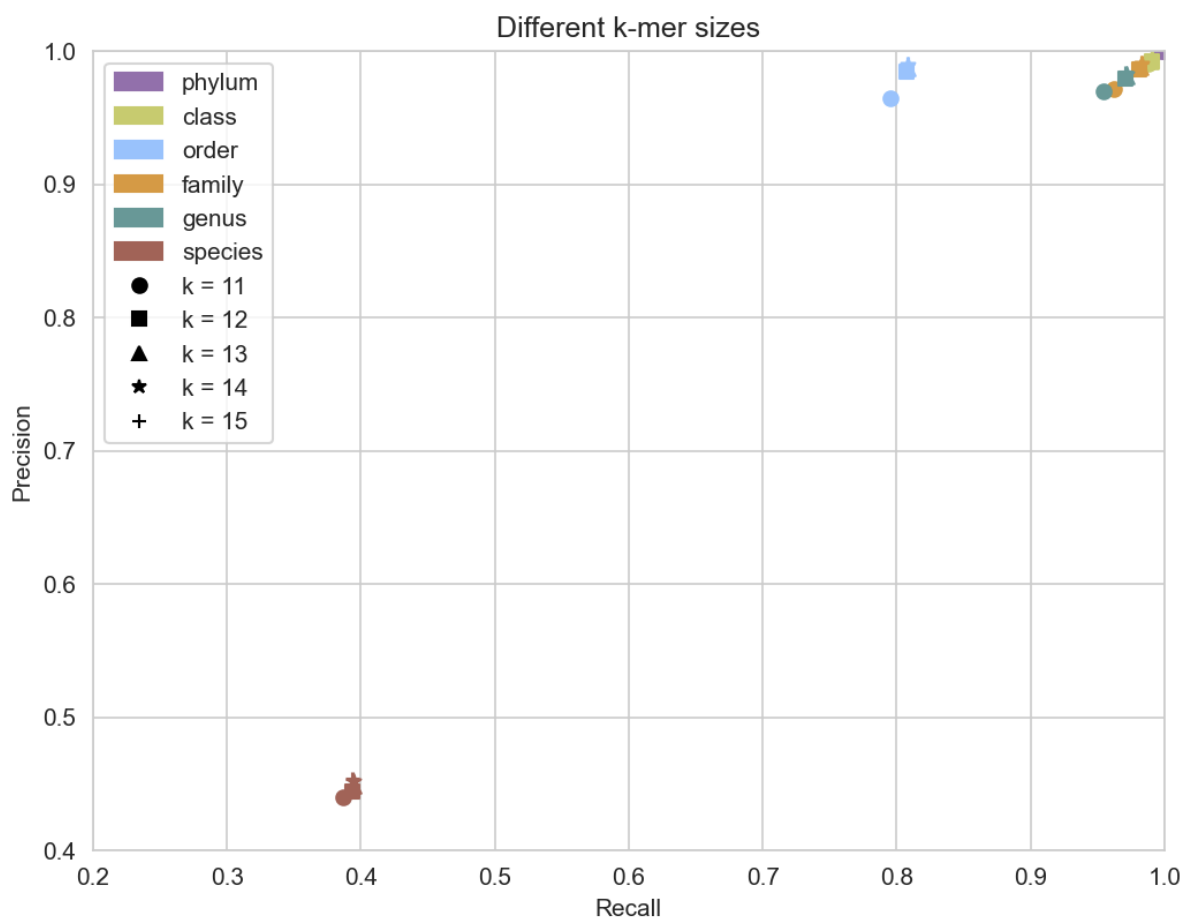


Figure A.3: Comparison of different k-mer sizes for diamer. Precision and recall are shown at different taxonomic ranks and with different k-mer lengths for the ZymoOral dataset with NR90 as the reference database. The Uniform11S reduced amino acid alphabet was used. The plot shows a worse recall at the rank order compared to family and genus. **This is not possible** and might be caused by an error during plot generation.

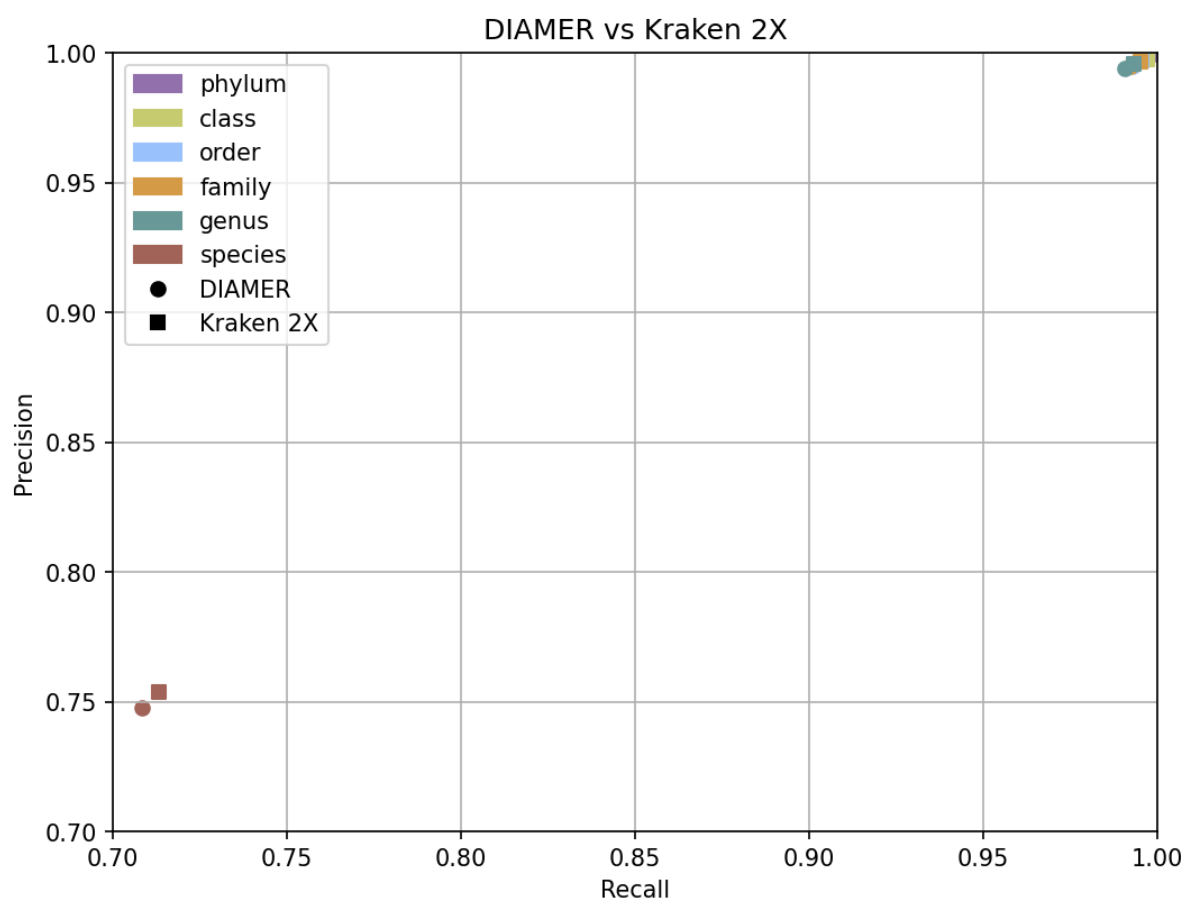


Figure A.4: Comparison of the performance of Kraken 2X and DIAMER on the ZymoOral dataset with the NR reference database. Kraken 2X was run with default parameters. DIAMER used the Uniform11S alphabet with a k-mer size of 13 and default filtering.

Erklärung

Laut Beschlüssen der Prüfungsausschüsse Bioinformatik, Informatik, Informatik Lehramt, Kognitionswissenschaft, Machine Learning, Medieninformatik und Medizininformatik der Universität Tübingen vom 05.02.2025. Gültig für Abschlussarbeiten (B.Sc./M.Sc./B.Ed./M.Ed.) in den zugehörigen Fächern. Bei Studienarbeiten und Hausarbeiten bitte nach Maßgabe des/der jeweiligen Prüfers/Prüferin.

1. Allgemeine Erklärungen

Hiermit erkläre ich:

- Ich habe die vorgelegte Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt.
- Ich habe alle wörtlich oder sinngemäß aus anderen Werken übernommenen Aussagen als solche gekennzeichnet.
- Die Arbeit war weder vollständig noch in wesentlichen Teilen Gegenstand eines anderen Prüfungsverfahrens.
- Falls ich ein elektronisches Exemplar und eines oder mehrere gedruckte und gebundene Exemplare eingereicht habe (z.B., weil der/die Prüfer/in(nen) dies wünschen): Das elektronisch eingereichte Exemplar stimmt exakt mit dem bzw. den von mir eingereichten gedruckten und gebundenen Exemplar(en) überein.

2. Erklärung bezüglich Veröffentlichungen

Eine Veröffentlichung ist häufig ein Qualitätsmerkmal (z.B. bei Veröffentlichung in Fachzeitschrift, Konferenz, Preprint, etc.). Sie muss aber korrekt angegeben werden. Bitte kreuzen Sie die für Ihre Arbeit zutreffende Variante an:

- ☒ Die Arbeit wurde bisher weder vollständig noch in Teilen veröffentlicht.
- ☐ Die Arbeit wurde in Teilen oder vollständig schon veröffentlicht. Hierfür findet sich im Anhang eine vollständige Tabelle mit bibliographischen Angaben.

3. Nutzung von Methoden der künstlichen Intelligenz (KI, z.B. chatGPT, DeepL, etc.)

Die Nutzung von KI kann sinnvoll sein. Sie muss aber korrekt angegeben werden und kann die Schwerpunkte bei der Bewertung der Arbeit beeinflussen. Bitte kreuzen Sie alle für Ihre Arbeit zutreffenden Varianten an und beachten Sie, dass die Varianten 3.4 - 3.6 eine vorherige Absprache mit dem/der Betreuer/in voraussetzen:

- ☐ 3.1. Keine Nutzung: Ich habe zur Erstellung meiner Arbeit keine KI benutzt.
- ☒ 3.2. Korrektur Rechtschreibung & Grammatik: Ich habe KI für Korrekturen der Rechtschreibung und Grammatik genutzt, ohne dass es dabei zu inhaltlich relevanter Textgeneration oder Übersetzungen kam. Das heißt, ich habe von mir verfasste Texte in derselben Sprache korrigieren lassen. Es handelt sich um rein sprachliche Korrekturen, sodass die von mir ursprünglich intendierte Bedeutung nicht wesentlich verändert oder erweitert wurde. Im Zweifelsfall habe ich mich mit meinem/r Betreuer/in besprochen. Alle genutzten Programme mit Versionsnummer sind im Anhang meiner Arbeit in einer Tabelle aufgelistet.

- ☒ 3.3. Unterstützung bei der Softwareentwicklung: Ich habe KI als Unterstützung beim Schreiben von Code in der Softwareentwicklung genutzt. Es handelt sich hierbei lediglich um Unterstützung und nicht um die automatische Generierung von größeren Programm-Teilen. Im Zweifelsfall habe ich mich mit meinem/r Betreuer/in besprochen. Alle genutzten Programme mit Versionsnummer sind im Anhang meiner Arbeit in einer Tabelle aufgelistet.
- ☐ 3.4. Übersetzung: Ich habe *nach vorheriger Absprache und mit Erlaubnis meines/r Betreuer/in* KI zur Übersetzung von mir in einer anderen Sprache geschriebenen Texte genutzt. Jede derartige Übersetzung ist im laufenden Text gekennzeichnet und der Anhang meiner Arbeit enthält eine Tabelle mit einem vollständigen Nachweis aller übersetzten Textstellen und der verwendeten Programme mit Versionsnummer.
- ☐ 3.5. Code-Generierung: Ich habe *nach vorheriger Absprache und mit Erlaubnis meines/r Betreuer/in* KI zur Erzeugung von Code in der Softwareentwicklung genutzt. Der Anhang meiner Arbeit enthält eine Tabelle mit einem vollständigen Nachweis aller derartigen Nutzungen, der verwendeten Programme mit Versionsnummer und der verwendeten Prompts.
- ☐ 3.6. Text-Generierung: Ich habe *nach vorheriger Absprache und mit Erlaubnis meines/r Betreuer/in* KI zur Erzeugung von Text in meiner Arbeit genutzt. Jede derartige Verwendung von KI ist im laufenden Text gekennzeichnet und der Anhang meiner Arbeit enthält eine Tabelle mit einem vollständigen Nachweis aller derartigen Nutzungen, der verwendeten Programme mit Versionsnummer und der verwendeten Prompts.

Falls ich in irgendeiner Form KI genutzt haben (siehe oben), dann erkläre ich:

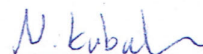
Mir ist bewusst, dass ich die Verantwortung trage, falls es durch die Verwendung von KI zu fehlerhaften Inhalten, zu Verstößen gegen das Datenschutzrecht, Urheberrecht oder zu wissenschaftlichem Fehlverhalten (z.B. Plagiaten) kommt.

4. Abschluss und Unterschrift(en)

Mir ist bekannt, dass ein Verstoß gegen diese Erklärung prüfungsrechtliche Konsequenzen haben und insbesondere dazu führen kann, dass die Prüfungsleistung mit „nicht ausreichend“ bzw. die Studienleistung mit „nicht bestanden“ bewertet wird und bei mehrfachem oder schwerwiegendem Täuschungsversuch eine Exmatrikulation erfolgen bzw. ein Verfahren zur Entziehung eines eventuell verliehenen akademischen Titels eingeleitet werden kann.

Noel Kubach

Tübingen, 17.05.2025



Vorname, Nachname
Student/in

Ort, Datum

Unterschrift

Die Punkte 3.4 - 3.6 erfordern eine Zustimmung des/r Betreuer/in. Sollten Sie einen dieser Punkte angekreuzt haben, dann sollte der/die Betreuer/in bitte hier unterschreiben:

Ich habe der oben genannten Nutzung von KI zur Erstellung der Arbeit zugestimmt.

Daniel Huson

7.5.25



Vorname, Nachname
Betreuer/in

Ort, Datum

Unterschrift