BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF COMPUTER SYSTEMS
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ

BIOINFORMATIC TOOL FOR CLASSIFICATION OF BACTERIA INTO TAXONOMIC CATEGORIES BASED ON THE SEQUENCE OF 16S RRNA GENE

BIOINFORMATICKÝ NÁSTROJ PRO KLASIFIKACI BAKTERIÍ DO TAXONOMICKÝCH KATEGORIÍ NA ZÁKLADĚ SEKVENCE GENU 16S RRNA

MASTER'S THESIS
DIPLOMOVÁ PRÁCE

AUTHOR **Bc. NIKOLA VALEŠOVÁ**
AUTOR PRÁCE

SUPERVISOR **Ing. STANISLAV SMATANA**
VEDOUCÍ PRÁCE

BRNO 2019

Department of Computer Systems (DCSY) Academic year 2018/2019

Master's Thesis Specification

21517

Student:	**Valešová Nikola, Bc.**
Programme:	Information Technology Field of study: Bioinformatics and biocomputing
Title:	**Bioinformatic Tool for Classification of Bacteria into Taxonomic Categories Based on the Sequence of 16S rRNA Gene**
Category:	Biocomputing

Assignment:

1. Study basic principles of metagenomics and its applications to the analysis of microbiome using the 16s rRNA amplicon sequencing.
2. Research existing methods for classification of 16S rRNA gene sequences into taxonomic categories.
3. Propose a new classification tool that will be able to classify 16S rRNA gene sequences into taxonomic categories. The assignment should be made on every taxonomic rank.
4. Implement the proposed tool and evaluate its performance on an appropriate testing dataset.
5. Evaluate achieved results and discuss possibilities of future continuation.

Recommended literature:

- Based on instructions from the supervisor.

Requirements for the semestral defence:

- Completion of tasks 1 and 3 from the specification.

Detailed formal requirements can be found at http://www.fit.vutbr.cz/info/szz/

Supervisor:	**Smatana Stanislav, Ing.**
Head of Department:	Sekanina Lukáš, prof. Ing., Ph.D.
Beginning of work:	November 1, 2018
Submission deadline:	May 22, 2019
Approval date:	October 26, 2018

Abstract

This thesis deals with the problem of automated classification and recognition of bacteria after obtaining their DNA by the sequencing process. In the scope of this work, a new classification method based on the 16S rRNA gene segment is designed and described. The presented principle is constructed according to the tree structure of taxonomic categories and uses well-known machine learning algorithms to classify bacteria into one of the classes at the lower taxonomic level. A part of this thesis is also dedicated to the implementation of the described algorithm and evaluation of its prediction accuracy. The performance of various classifier types and their settings is examined and the setting with the best accuracy is determined. The accuracy of the implemented algorithm is also compared to several existing methods. During validation, the implemented KTC application reached more than 45 % accuracy on genus prediction on both BLAST 16S and BLAST V4 datasets. At the end of the thesis, there are mentioned several possibilities to improve and extend the current implementation of the algorithm.

Abstrakt

Tato práce se zabývá problematikou automatizované klasifikace a rozpoznávání bakterií po získání jejich DNA procesem sekvenování. V rámci této práce je navržena a popsána nová metoda klasifikace založená na základě segmentu 16S rRNA. Představený princip je vytvořen podle stromové struktury taxonomických kategorií a používá známé algoritmy strojového učení pro klasifikaci bakterií do jedné ze tříd na nižší taxonomické úrovni. Součástí práce je dále implementace popsaného algoritmu a vyhodnocení jeho přesnosti predikce. Přesnost klasifikace různých typů klasifikátorů a jejich nastavení je prozkoumána a je určeno nastavení, které dosahuje nejlepších výsledků. Přesnost implementovaného algoritmu je také porovnána s několika existujícími metodami. Během validace dosáhla implementovaná aplikace KTC více než 45% přesnosti při predikci rodu na datových sadách BLAST 16S i BLAST V4. Na závěr je zmíněno i několik možností vylepšení a rozšíření stávající implementace algoritmu.

Keywords

Machine learning, metagenomics, bacteria classification, phylogenetic tree, taxonomy, 16S rRNA, DNA sequencing, scikit-learn

Klíčová slova

Strojové učení, metagenomika, klasifikace baterií, fylogenetický strom, taxonomie, 16S rRNA, sekvenování DNA, scikit-learn

Reference

VALEŠOVÁ, Nikola. *Bioinformatic Tool for Classification of Bacteria into Taxonomic Categories Based on the Sequence of 16S rRNA Gene.* Brno, 2019. Master's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Ing. Stanislav Smatana

Rozšířený abstrakt

Když se lidé narodí, jejich tělo obsahuje pouze jejich vlastní eukaryotické lidské buňky. Jak ale vyrůstáme a interagujeme s ostatními lidmi a věcmi kolem nás, povrch naší kůže, stejně jako naše střeva, jsou kolonizovány různými bakteriemi, viry a houbami. Komunita těchto dalších buněk se nazývá lidský mikrobiom. Může dosáhnout téměř desetkrát vyššího počtu buněk, než je počet buněk přirozeně lidských. [40]

Obvykle jsou tyto mikroby neutrální nebo dokonce prospěšné pro naše tělo, pomáhají nám trávit potravu a mají důležitou úlohu pro náš imunitní systém. Nicméně dysfunkční lidský mikrobiom byl již spojen s mnoha onemocněními, jako je diabetes, zánětlivé onemocnění střev a infekce rezistentní vůči antibiotikům. [28], [40]

Po dlouhou dobu bylo možné analyzovat bakterie v lidském mikrobiomu pouze jejich kultivací. Mnohé druhy bakterií jsou však nekultivovatelné, a proto nebyly vůbec zjistitelné. Díky nedávnému pokroku v sekvenování s vysokou průchodností je nyní možné účinně vyšetřovat mikrobiální komunity a analyzovat druhy bakterií, které se v nich nacházejí. Se získanými poznatky se pozornost mnoha vědců zaměřila na výzkum vztahu mezi lidským mikrobiomem a lidským zdravím. Vzhledem k tomu, že oblast výzkumu lidského mikrobiomu je nová, není dobře prozkoumána a týká se lidského zdraví, je to velmi slibné odvětví výzkumu. [28], [40]

Cílem této práce je navrhnout a popsat novou metodu klasifikace bakterií, která je založena na genovém segmentu 16S rRNA. Sekvence 16S rRNA je odlišná pro každý rod a může obsahovat několik mutací, inzercí a delecí, proto různé sekvence 16S rRNA mohou mít různou délku. To by mohlo způsobit nepříjemnosti, jelikož algoritmy strojového učení vyžadují, aby jejich vstupní vektory měly stejné rozměry. Pro překonání těchto obtíží je nejprve ze vstupní sekvence extrahováno k-merové spektrum, které se následně použije pro klasifikaci. S využitím k-merového spektra je možné transformovat každou sekvenci 16S rRNA, která je ve formě řetězce, na numerický vektor, kde každá hodnota představuje počet výskytů odpovídajícího dílčího řetězce v původní sekvenci.

Prezentovaný princip klasifikace je postaven na stromové struktuře taxonomických kategorií. Celý klasifikátor se skládá ze stromu dílčích klasifikátorů s topologií respektující taxonomický strom. Klasifikace vstupního vzorku začíná v horním klasifikátoru rozlišujícím mezi bakteriemi a archaea a vstupní sekvence sestupuje stromem podle predikovaných taxonomických kategorií. Dílčí klasifikátory představují dobře známé metody strojového učení (jako například SVM, rozhodovací strom a k-NN) a jejich cílem je klasifikovat dané bakterie a přiřadit jim třídu na nižší taxonomické úrovni.

Tréninková metoda prezentovaného klasifikátoru může být rozdělena do dvou částí. Nejprve proti sobě soutěží všechny typy klasifikátorů s různými nastaveními, aby bylo možné určit nastavení klasifikátoru dosahující nejvyšší přesnosti. Pro provedení soutěže je celý proces tvorby stromu klasifikátorů, tréningu a validace zabalen do cross-validace. V každé iteraci jsou na aktuálním souboru trénovacích dat natrénovány všechny typů klasifikátorů v různých konfiguracích a jejich přesnost je pak vyhodnocena na souboru validačních dat. Výsledkem každé iterace je součet přesných predikcí získaných během validace.

Po celém procesu cross-validace je určen klasifikátor, který dosáhl nejvyšší celkové přesnosti klasifikace. Ten je poté natrénován na všech dostupných datech a uložen jako finální model pro další použití. Díky tomuto přístupu je možné získat pro každou použitou datovou sadu ten klasifikátor, který dosahuje nejvyšší přesnosti.

Díky použití taxonomické stromové struktury a konceptu postupné klasifikace by mělo být možné snížit celkovou klasifikační chybu ve srovnání s přímou predikcí nejnižší taxonomické úrovně. Tento přístup také nabízí možnost prezentace kompletní taxonomické klasifikace od domény až po rod s hodnotami předpokládané přesnosti pro kategorie na každé taxonomické úrovni.

Během porovnání různých velikostí k-meru bylo dosaženo nejvyšší přesnosti predikce při použití k-meru o velikosti 5 a 6. Zkoumání přesnosti predikce s použitím jednotlivých hypervariabilních regionů ukázalo, že regiony s nejlepší přesností se významně liší pro obě použité datové sady. Celkově byla nejlepší přesnost dosažena při použití regionů V1, V3 a V8. Během validace dosáhla implementovaná aplikace KTC přibližně 47,3% přesnosti při predikci rodu na datové sadě BLAST 16S a přesnosti 45,5 % na datové sadě BLAST V4. Aplikace byla testována také na databázi hub ITS, kde získaná přesnost na úrovni rodu byla přibližně 75,3 %.

Bioinformatic Tool for Classification of Bacteria into Taxonomic Categories Based on the Sequence of 16S rRNA Gene

Declaration

Hereby I, Nikola Valešová, declare that this thesis entitled "Bioinformatic Tool for Classification of Bacteria into Taxonomic Categories Based on the Sequence of 16S rRNA Gene" was prepared as an original author's work under the supervision of Ing. Stanislav Smatana. All the relevant information sources, which were used during preparation of this thesis, are properly cited and included in the list of references.

. . . .
Nikola Valešová
May 21, 2019

Acknowledgements

I would like to express my sincere gratitude to my advisor, Ing. Stanislav Smatana, for the continuous support of my study and for his motivation, valuable advice and patience. His guidance was highly appreciated during the work on this thesis.

Contents

Chapter 1

Introduction

When people are born, their body contains only their own eukaryotic human cells. However, as we grow up and interact with other people and things around us, the surface of our skin, as well as our gut, becomes colonized by various bacteria, archaea, viruses, and fungi. The community of these additional cells is called the human microbiome. It can reach almost ten times higher number of cells than the count of natural human ones. [40]

Usually, these microbes are neutral or even beneficial for our body, they help us digest food and have an important role in our immune systems. However, the dysfunctional human microbiome has been also already linked to many diseases, such as diabetes, inflammatory bowel disease, and antibiotic-resistant infection. [28], [40]

For a long time, it was possible to analyse bacteria in the human microbiome only by their cultivation. However, many bacteria species are unculturable and therefore were not detectable at all. Thanks to recent advance in high-throughput sequencing, it is now achievable to efficiently investigate microbial communities and analyse the bacteria species found in them. With the obtained knowledge, the attention of many scientists has been drawn to the research of the relationship between the human microbiome and human health. As the field of human microbiome investigation is quite new, not well explored and concerns human health, it is a very promising branch of research. [28], [40]

The aim of this thesis is to design and describe a new bacteria classification method, which is based on the 16S rRNA gene segment. The presented principle is constructed according to the tree structure of taxonomic categories and uses well-known machine learning methods to classify bacteria into one of the classes at a given taxonomic level. In the scope of this project, the described method is implemented and its accuracy is evaluated and compared with other existing bacteria classification tools. Lastly, the presented method is analysed in order to define its weak points and possibilities for improvement.

This thesis can be notionally divided into two parts – theoretical and practical. In the first category, chapters 2 to 5 could be included, which aim to introduce the basic terms and concepts in the areas of bioinformatics and machine learning and to give an introduction into the problem of bacteria classification. Chapter 2 is focused on preliminary knowledge regarding the fields of bioinformatics and metagenomics, which is essential for understanding the core of this work. The concepts of DNA and RNA are introduced and briefly described. The last part of this chapter deals with the explanation of taxonomy. Chapter 3 describes other methods of digital processing of sequence data and explains the terms k-mer, k-mer spectrum and k-mer similarity. Chapter 4 provides an overview of the area of some well-known classification algorithms, which are used in the implemented classifier. Chapter 5 gives information about other existing methods addressing the bacteria classification problem.

The practical part includes chapters 6 to 8 and is devoted to the description of the implemented application. The focus of chapter 6 is on a detailed specification of the proposed method ranging from input rRNA sequence processing up to the unknown specimen classification. Chapter 7 aims to describe the application implementation and to list the required libraries and dependencies. Chapter 8 is dedicated to the evaluation of the proposed method and presenting the obtained results.

Chapter 9 concludes this work and contains also the proposal of suggestions for further development.

Chapter 2

Metagenomics

Metagenomics is a field of study focused on the microbial world. Its main characteristic is the investigation of bacteria, viruses and fungi in complex communities in which they usually exist, irrespective of whether they are culturable of not. Metagenomics tries to examine the DNA in a sample of a microbial community as a whole. [39], [62]

The aim of this chapter is to offer a brief introduction into the field of metagenomics and explain the major concepts that form the preliminary knowledge needed to fully understand the method proposed in the scope of this thesis.

This chapter is divided into three main parts. Section 2.1 contains basic introduction into DNA, such as information about what it consists of and what its structure looks like. Section 2.2 includes description of RNA, rRNA and 16S rRNA. There are also listed the differences between RNA and DNA. Section 2.3 provides a brief explanation of the taxonomy and structure of the taxonomic tree.

2.1 DNA

Deoxyribonucleic acid (or shortly DNA) is a macromolecule with hereditary information encoded in it describing recipes for making proteins and other functional molecules that the organism needs to survive. DNA can be found in almost all known organisms. Most cells in the human body share the same DNA. In a cell, DNA can be found in its nucleus (then it is called nuclear DNA), or in the mitochondria (which is called mitochondrial DNA). [22]

The DNA can be represented as a code consisting of four chemical bases – adenine (A), guanine (G), cytosine (C), and thymine (T). Information in DNA is encoded by combining these bases into long sequences. [22]

The basic building unit of DNA is called a nucleotide. It is composed of one base, a sugar and a phosphate. Nucleotides form a long sequence creating one strand and thanks to their capability of pairing up with another base (A pairs up with T and C with G), a long spiral is built called the double helix. [22]

DNA sequencing is the process of determining the nucleotide sequence of DNA. The obtained sequence gives the most fundamental knowledge of a gene or a genome. A genome is the complete set of DNA of an organism, which includes all of its genes. All needed information on how to build and maintain the organism can be found in its genome. [1], [21]

Today, it is impossible to sequence an entire genome or a single chromosome. It has to be broken down into smaller chunks that are easier to manage. On these partitions, various techniques which label the individual bases are applied to obtain the number of bases and their order. [69]

2.2 RNA

Ribonucleic acid (shortly RNA) is a nucleic acid consisting of a long strand of nucleotides. Similarly to DNA, each nucleotide contains a nitrogenous base, a sugar, and a phosphate. [38]

The main difference between DNA and RNA is that RNA has only one strand. Furthermore, the sugar found in RNA nucleotides is ribose while DNA nucleotides contain deoxyribose. Last important difference between RNA and DNA is in nucleic acids as RNA is also composed of nucleotides, however, instead of thymine, a different type of nucleotide is present – uracil. [38]

Comparison of the structure of DNA, which is a double helix, and RNA, which is a single helix, and of the nucleotides they are composed of can be seen in figure 2.1.

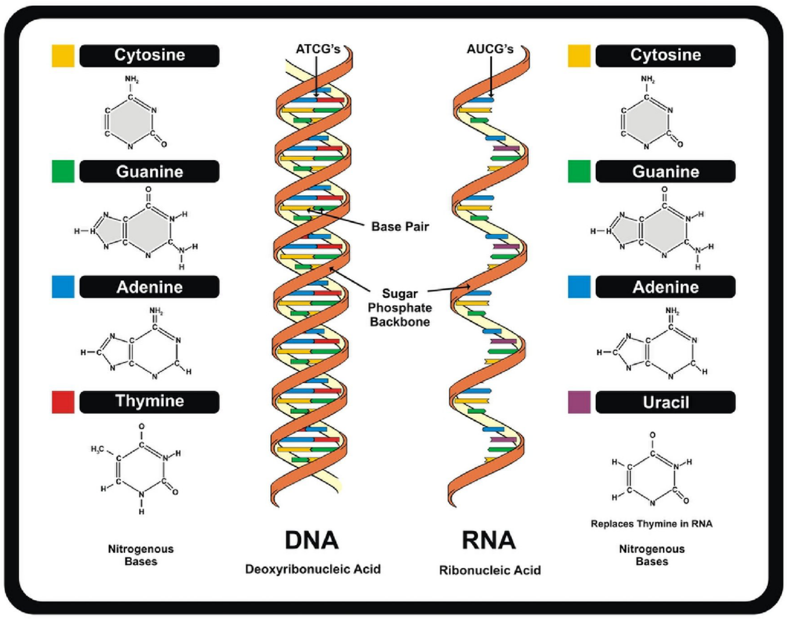


Figure 2.1: Comparison of the structure of DNA and RNA and of the nucleotides they consist of. This image was taken from the article *DNA: Definition, Structure & Discovery* by Rachael Rettner [50].

rRNA is one type of RNA called ribosomal ribonucleic acid. It is located in ribosomes, which are the catalysts of protein synthesis. Over sixty per cent of the ribosome consists of the ribosomal RNA which is a necessary part of all ribosome functions, such as binding to mRNA and urging the catalysis of the peptide bond formation between two amino acids. [36]

16S rRNA is a sequence of DNA which encodes the RNA component of the smaller subunit of the bacterial ribosome. It can be found in the genome of all bacteria species and a related form can be found in all cells. In the 16S rRNA, two types of regions can be determined. There have been detected portions which change very slowly during evolution and other parts that are variable and undergo rapid genetic changes. Therefore, they are suitable for determining the taxonomic classifications of bacteria. The 16S rRNA gene has been proved to have the most information regarding the examination of evolutionary relatedness. [1]

The number of hypervariable regions and their proportions can be seen in figure 2.2. The entire 16S rRNA sequence, which is approximately 1500 base pairs long, contains nine variable regions separated by ten conserved regions. [61]

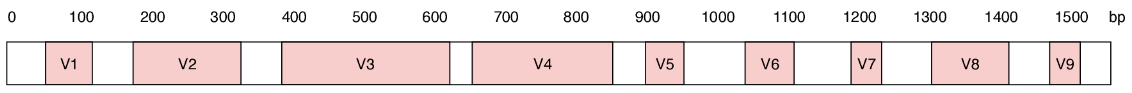


Figure 2.2: Visualisation of the 16S rRNA gene sequence with conserved (white) and variable (red) regions (created according to information found in the article by Singer et al. [61])

2.3 Taxonomy

Taxonomy is the part of science focused on naming and classification of all organisms which includes animals, plants, fungi and microorganisms. The system was first introduced in the 18th century by Swedish naturalist Carolus Linnaeus. He is the inventor of the principle of assigning each organism its genus and species name. He also developed a hierarchical classification system which is still used today (with some changes). The system is called the taxonomic hierarchy. Within the system, organisms are organised into groups according to morphological, behavioural, genetic and biochemical observations. Each level of classification is called a taxon. [11], [64]

Today, the taxonomic hierarchy consists of eight levels which are (from general to specific): domain, kingdom, phylum, class, order, family, genus, and species. The first and broadest level is domain. There are three domains – archaea, bacteria, and eukarya, and all living organisms belong to one of these categories. Within each domain, there are kingdoms, each kingdom contains phyla, followed by class, order, family, genus, and species. [11], [64]

The application proposed and described within this thesis focuses on the classification of organisms belonging to two of the mentioned domains – bacteria and archaea, since in eukarya, 30S rRNA, which contains the 16S rRNA sub-unit, is not present. Instead, there can be found 40S rRNA containing the 18S rRNA sub-unit, which is suitable for classification of eukarya. [32]

Chapter 3

Methods of Sequence Data Digital Processing

Sequence data are represented as strings composed of four nucleotides – A, C, G, U (or T for DNA sequences). In bioinformatics, a comparison of two gene sequences and measurement of their difference is a common task. One way of computing the difference between two strings is Hamming distance (the number of positions at which the corresponding characters differ). However, it is not typically used to compare RNA or protein sequences as it expects the compared sequences to have the same length. In these types of data, the i^{th} character in one sequence corresponds to a symbol at a different and unknown position in the other sequence due to DNA replication errors that lead to substitutions, insertions, and deletions of nucleotides. While strings $ACACACAC$ and $CACACACA$ are considered to be very different when using Hamming distance, their distance becomes significantly smaller if the sequences are aligned by moving one of them to the right over one place. Therefore, another approach of comparison of two sequences is used instead of the Hamming distance. It is called sequence alignment. [30]

The first part of this chapter, represented by section 3.1, describes the term sequence alignment, its types and well-known sequence alignment algorithms. The rest of this chapter consisting of section 3.2 is dedicated to the description of the k-mer spectrum feature, which can be also used for gene sequence comparison, and its characteristics that have the biggest impact on machine learning applications.

3.1 Sequence Alignment

Sequence alignment is a technique of measuring the similarity of two sequences. It is defined as the minimum number of editing operations needed to transform one sequence into another. There are three types of the editing operations – symbol insertion, symbol deletion, and substitution of one character for another. Another advantage of sequence alignment is that, unlike Hamming distance, it allows comparison of strings of different lengths. [30]

There are two types of sequence alignment – global and local sequence alignment. Global alignment accepts two sequences, v and w, and a scoring matrix as input and its objective is to find the best alignment between the given strings, i.e. to return the alignment with the maximal score among all possible alignments. The score is computed using values from the given scoring matrix, which prescribes awards for matching characters and penalizations for differing characters and gap character insertion. [30]

Global alignment searches for similarities between two entire sequences, which is useful when the similarity between the strings is expected to extend over their entire length. When biologically significant similarities can be found only in certain parts of the gene sequences, the objective is to maximize the alignment score over all substrings of v and w. This problem is called local alignment as the alignment does not need to extend over the entire length of the sequence. The input of local alignment is the same as for global alignment – two sequences, v and w, and a scoring matrix, and it returns substrings of v and w whose global alignment is maximal among all global alignments of all substrings of v and w according to the given scoring matrix. [30]

3.1.1 Needleman–Wunsch Algorithm

The Needleman–Wunsch algorithm was introduced in 1970 by Saul B. Needleman and Christian D. Wunsch. It is a commonly used approach to compute the optimal global alignment of two genetic sequences. [43]

In order to determine the maximum match of two sequences, A and B, the algorithm uses a two-dimensional array to represent all possible pair combinations that can be obtained from the input sequences by gap insertions. Let A_j be the j^{th} symbol of sequence A and B_i be the i^{th} character of sequence B, A_j represents the column and B_i the row of the matrix M. Then the cell $M_{i,j}$ represents a pair containing A_j and B_i. The initial created array has $l(A) + 1$ columns and $l(B) + 1$ rows, where $l(X)$ represents the length of sequence X. An additional row and column is added at the beginning of the matrix to align with the gap. [43], [66]

Every possible comparison of two gene sequences can now be represented by a path through the matrix with every character of the input sequences occurring in every path maximally once. Any pathway can be then represented by a sequence of cells, $M_{a,b}$ to $M_{y,z}$, where $a \geq 1$, $b \geq 1$, $y \leq l(A)$, and $y \leq l(B)$. A pathway can be then displayed as a route connecting cells of the matrix. [43], [66]

Two characters of the aligned sequences can match, mismatch or a gap can be applied meaning an insertion or a deletion. Multiple scoring systems can be applied. In the basic schema used by Needleman and Wunsch, the cell $M_{i,j}$ is assigned value 1 if the nucleotides A_j and B_i match and -1 if the two characters differ. The gap penalty is also given as -1. [43]

The algorithm of finding the optimal global alignment begins with creating the described matrix and then initialising its first row and column with values corresponding to the count of consequent inserted gaps. After that, the rest of the matrix is filled starting from the upper left corner. To find the maximum score of each cell, it is necessary to know the scores of neighbours of the current cell (diagonal, left and right). From these values, it is possible to obtain tree different scores – by adding the match or mismatch score to the diagonal value and adding the gap penalization to the remaining neighbouring values. The maximum among the three resulting values is then filled into the $M_{i,j}$ cell. The formula for computing the score of cell i, j of a matrix M can be written as follows: [66]

$$M_{i,j} = \max(M_{i-1,j-1} + S_{i,j}, M_{i,j-1} + W, M_{i-1,j} + W), \tag{3.1}$$

where $S_{i,j}$ represents the score or penalty of the characters A_j and B_i and W is the gap penalty. The equation 3.1 is applied to compute the values of the remaining rows and columns in the matrix M and (according to the improved version of the Needleman–Wunsch algorithm presented in article [66]) back pointers are added pointing to the cell from where the maximum score originated. [43], [66]

The aim of the final step of the algorithm is to trace back and find the best alignment. This phase starts in the bottom right corner of the matrix and follows the back pointers towards the beginning of the matrix. Every cell can have one or more back pointers so, generally, there can be two or more alignments possible between the two aligned sequences. By following the pointers to the upper left corner of the matrix, the alignment of the two input sequences can be found. The best alignment among all alignments can be determined with the use of the maximum alignment score. [66]

3.1.2 Smith–Waterman Algorithm

The Smith–Waterman algorithm is a well-known algorithm solving the problem of local sequence alignment, which means it finds similar subsequences between two gene sequences. The algorithm is a modification of the Needleman-Wunsch algorithm for global sequence alignment and it was proposed in 1981 by Temple F. Smith and Michael S. Waterman. Instead of aligning the entire sequences, the Smith-Waterman algorithm compares subsequences of all possible lengths. It aims to find the optimal local alignment according to the used scoring system. [2], [30]

This algorithm differs from the Needleman-Wunsch algorithm mostly in two aspects. Firstly, instead of assigning the matrix cell a negative value, it is set to zero in order to highlight the best local alignments. This step can be interpreted as search restarting. And secondly, the traceback phase starts from all cells with the highest score and continues until a cell with the score of zero is reached. [2]

In the first step of the Smith–Waterman algorithm for local alignment of two sequences, A and B, the matrix with $len(A) + 1$ columns and $len(B) + 1$ rows is formed. As this algorithm does not assign cells negative scores, the values in the first row and first column are set to zero. [2], [67]

The second step consists of filling the rest of the matrix with scores according to values of their neighbouring cells and scores and penalizations defined by the scoring schema. The back pointers to the cell from where the maximum score originated are stored for every cell of the matrix. The only difference in this step is in setting negative values to zero. The formula for computing the score of cell i, j of a matrix M is now in the following form: [67]

$$M_{i,j} = \max(M_{i-1,j-1} + S_{i,j}, M_{i,j-1} + W, M_{i-1,j} + W, 0), \tag{3.2}$$

where $S_{i,j}$ represents the score or penalty of the characters A_j and B_i and W is the gap penalty. [67]

The final step is to trace back to find the optimal alignment. It starts from the cell with the maximum score obtained in the entire matrix. There can possibly be more cells containing the maximum value which might lead to more than one alignment. By following the back pointers, it is possible to move to the predecessors of the cells until a cell with the score of zero is reached. Every cell can have more than one back pointer. In that case, both alignments can be taken into account and the best one is determined afterwards by summing up their scores for match and penalizations for mismatch and gaps. The alignment with the maximum overall score is the best local alignment of the two sequences. [67]

3.2 K-mer Spectrum

K-mer refers to a subsequence of length k found in an input sequence. The term k-mers then represents all length k subsequences of a given sequence. The multiplicity of each k-mer in an input sequence is shown in k-mer spectrum which represents the abundance histogram of individual k-mers. K-mer spectrum is also often used for k-mer visualisation. In the field of computational genomics, the sequences that are being processed are often composed of nucleotides. [34]

A sequence of length L contains $L - k + 1$ k-mers. The number of all possible k-mers of a sequence relies only on the k-mer size and the count of characters that can be found in the processed string. That means that the length of the input sequence does not affect the size of the extracted k-mer spectrum. The length of the k-mer spectrum (the number of all substrings that are being counted in the original string) can be computed using the formula [34]

$$n^k, \tag{3.3}$$

where n is the number of possible characters (size of the alphabet) and k is the k-mer size which represents the length of subsequences that are being found. [34]

To compare two k-mer spectra, k-mer similarity can be applied, which returns the higher the value the more alike the k-mer spectra are. For a query sequence q and reference database R, let $W(q)$ be the set of all k-mers of q. For each reference sequence $r \in R$, k-mer similarity between the sequences q and r is defined as: [17]

$$ks(r, q) = |W(r) \cap W(q)|, \tag{3.4}$$

which represents the number of k-mers the two sequences have in common.

One of the biggest advantages of using k-mer spectra is that the extracted k-mer spectrum is of the same size, regardless of the input sequence length. That is important when applying machine learning algorithms as they require their input vectors to be of the same dimensions. On the other hand, extracting k-mer spectrum from a sequence leads to loss of positional information of the subsequences in the original sequence, which could be also useful for classification. [34]

Chapter 4

Methods of Classification

Classification is the process of assigning a given object to a certain class based on its features (attributes). It can be also described as the task of approximating a mapping function (f) from input variables (x) to discrete output variables (y). Classification is a type of supervised learning, which means that labels of the training and validation data are known. Therefore, the input dataset is in the following form: [3]

$$S = \{\langle \vec{x}, y \rangle \mid f(\vec{x}) = y\}, \tag{4.1}$$

where $\vec{x}$ is one sample and y represents the corresponding label. [3]

The aim of this chapter is to give the basic introduction to machine learning terms and algorithms regarding classification, which are used in the presented bacteria classification method.

This chapter could be logically divided into three parts. The first part consisting of section 4.1 explains the term cross-validation and its importance in machine learning applications. The second part contains section 4.2 which introduces accuracy, a frequently used metric used to evaluate how well a classification model works and to compare the prediction performance of various classifier types. The last logical part of this chapter consists of sections 4.3 to 4.9, each of which is dedicated to the description of one classification algorithm and its features. In section 4.3, the SVM classifier is introduced along with its three kernel types. Section 4.4 contains information about the nearest centroid classifier and its distance metrics. Section 4.5 is devoted to the description of the k-NN classifier and its advantages and disadvantages. Section 4.6 focuses on the decision tree, which is a very popular and easy to understand classifier type. In section 4.7, the basic principle of a random forest model and the process of its generation are described. Section 4.8 is devoted to an explanation of multilayer perceptron, a simple neural network suitable for classification. Section 4.9 introduces the naive Bayes classifier and two ways of its extension to real-valued attributes.

4.1 Cross-Validation

Cross-validation is a method of statistical model validation which aims to estimate how well will the model perform in practice, on unseen data. It is also frequently used as a tool to compare various models and choose a model for a given predictive problem. During cross-validation, the model is tested already in the training phase (on validation dataset) in order to minimise problems such as overfitting and underfitting. [9], [16] There are various validation strategies which differ in the number of splits that are done in the dataset. One well-known validation strategy is called k-fold cross-validation. The process has one parameter, k, which represents the number of groups that the input dataset is to be divided into. The entire procedure starts with randomly shuffling the dataset and splitting it into k parts, each of which is used for validation in one iteration and the rest of the data is used for training. After fitting the model on the training data, the model is validated on the portion of data left for validation. After all k iterations, the evaluation scores from all loops are summarised to show the performance of the model. [9], [16], [44]

The schema of cross-validation is shown in figure 4.1. There is an example of 5-fold cross-validation which means that the input dataset is split into five parts and the training and validation phases are executed five times, every time with a different part of the original dataset left for validation.

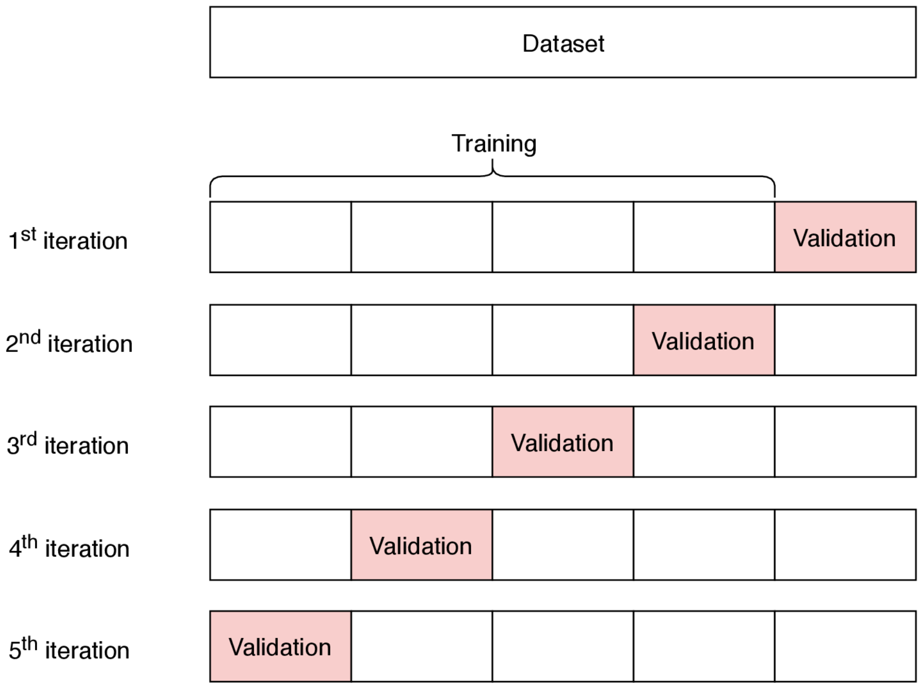


Figure 4.1: Schema of dataset division during cross-validation using 5 folds (created based on the information in the article by Karl Rosaen [52])

4.2 Model Evaluation

In order to evaluate how well a classification model works and to be able to compare the performance of multiple classifiers, several performance metrics are defined and used by data scientists on an everyday basis. In this work, the best-known metric called accuracy is used for this purpose.

The mentioned performance metric can be computed using values from a confusion matrix. The confusion matrix is an $N \times N$ table, where N symbolises the number of classes. On one axis is the predicted label and the other axis represents the actual label. The confusion matrix shows the classification predictions of the model and how successful they were. When dealing with a multi-class classification problem, the confusion matrix can be used to determine mistake patterns that occur more often than others. Confusion matrix for binary classification is given in table 4.1. [23]

Table 4.1: Confusion matrix for a binary classification problem

	Positive (predicted)	**Negative (predicted)**
Positive (actual)	True positive	False negative
Negative (actual)	False positive	True negative

Accuracy represents the fraction of correctly predicted labels out of all predictions. In a binary classification problem, it can be computed using the formula: [23]

$$accuracy = \frac{True\ positives + True\ negatives}{True\ positives + True\ negatives + False\ positives + False\ negatives}. \tag{4.2}$$

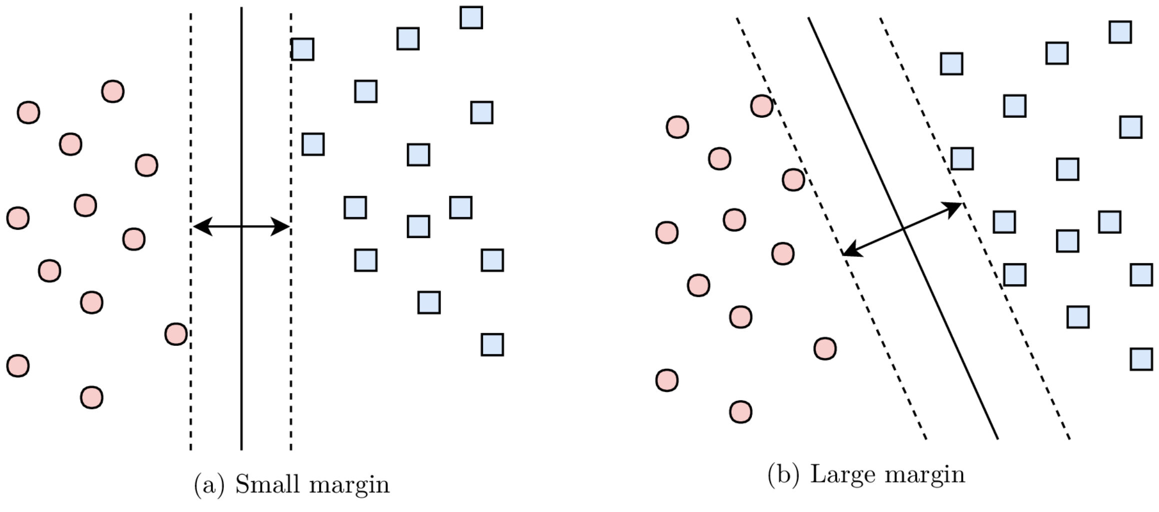
The generalised form of the previous formula, which can be used for computing accuracy in multi-class classification, is defined as: [23]

$$accuracy = \frac{Number\ of\ correct\ predictions}{Total\ number\ of\ predictions}. \tag{4.3}$$

4.3 SVM

SVM (Support Vector Machine) is one of the supervised algorithms used for machine learning problems such as classification and regression. In the basic version, it is linear (similarly to perceptron). SVM tries to find the line or hyper-plane that differentiates the two classes of objects the best, that means to find the line (or hyper-plane) which generates the largest margin between the two classes. The principle is shown in figure 4.2. [49]

Support Vectors are the data points which are the closest to the hyper-plane, and therefore influence its shape the most. [20]


(a) Small margin

(b) Large margin

Figure 4.2: Visualisation of SVM maximizing the distance margin (created based on information in the article by Rohith Gandhi [20])

SVM classifier has two major advantages – it is effective in high-dimensional spaces and it is also memory efficient as it uses a subset of training points in the decision function. Both of the advantages can be of great importance especially when working with larger k-mer sizes. On the other hand, SVM classifiers tend to be susceptible to over-fitting so it is important to tune the regularization parameter. [5]

4.3.1 Kernel Types

So far, only linear SVM has been introduced. However, the real world data are often not linearly separable. SVM is capable of converting any data into linearly separable by a method called the kernel trick. The method, which takes low-dimensional input space and maps it into a higher-dimensional space turning a non-separable problem into a separable one, is called a kernel, which is a mathematical function replacing the standard dot product operation. The three most commonly used kernel types are linear, RBF and sigmoid. [49]

The first mentioned kernel type, which has already been introduced, is the linear kernel. It is the basic kernel type and it is defined as: [59]

$$k(x, y) = x^T y, \tag{4.4}$$

where x and y are column vectors. [59]

The RBF (Radial Basis Function) kernel can be described using the following formula: [59]

$$k(x, y) = \exp(-\gamma \|x - y\|^2), \tag{4.5}$$

where x and y are the input vectors and γ is the “spread” of the kernel. [59]

The sigmoid kernel (also known as hyperbolic tangent or multilayer perceptron) is defined as: [59]

$$k(x, y) = \tanh(\gamma x^T y + c_0), \tag{4.6}$$

where x and y are the input vectors, γ represents slope and c_0 is known as intercept. [59]

4.4 Nearest Centroid Classifier

The nearest centroid classifier is built on a similar basis to the k-means algorithm. The main idea behind both mentioned algorithms is that each class is represented by its *centroid*, which is the centre of mass of its members or their vector average. Test samples are then assigned to the class which centroid is the nearest to the analysed sample. [47]

The visualisation of the classification process in two-dimensional space can be seen in figure 4.3. There are shown three classes of objects (represented by circles, diamonds and squares) with their corresponding centroids marked with an “x” symbol. The individual classes are separated by straight lines called decision boundaries. They are formed of points with the same distance from two nearest centroids. A new test input is shown as a filled circle in the centre of the image. It is connected to the centroids and the shortest line leads to the centroid of the final classification, in this case, it would be class 2.

A class centroid is computed as an average of the members of the given class. Let C be the examined class, S be the set of all samples with their corresponding classifications, S_C be the subset of samples in S which belong to class C ($S_C = \{\langle \vec{x}, C \rangle \mid \langle \vec{x}, C \rangle \in S\}$). The centroid $\vec{\mu}_C$ can be then computed with the use of the following equation: [47]

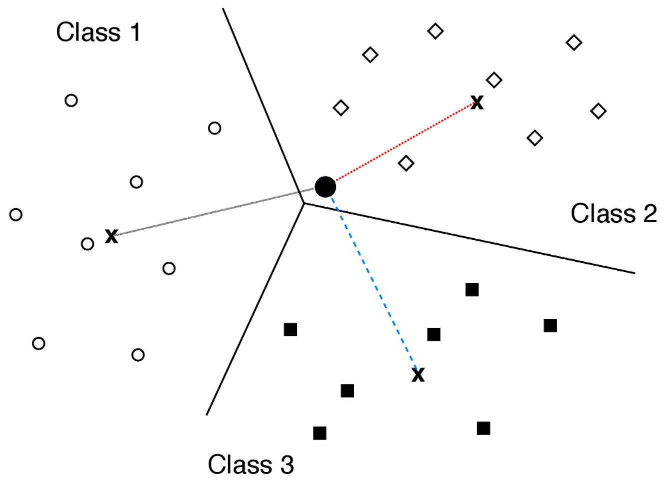


Figure 4.3: Visualisation of nearest centroid classification (created based on the information in the article on the Stanford NLP Group website [47])

$$\vec{\mu}_C = \frac{1}{|S_c|} \sum_{\langle \vec{x}, y \rangle \in S_c} \vec{x}. \tag{4.7}$$

When classifying a test sample, the distances from centroids of all classes are computed. The classified input is then assigned to the class with the lowest computed distance. [47]

Nearest centroid classifier is based on a simple to understand algorithm. Moreover, there are no parameters to tune which makes it very easy to start with. On the other hand, it has a problem when dealing with non-convex classes and classes with extremely different variances. [58]

4.4.1 Distance Metrics

When defining the nearest centroid classifier, the distance between a test sample and a centroid has been so far referred to as the Euclidean distance of two vectors. However, the use of various distance metrics is possible and its choice can have a huge impact on the prediction accuracy. In this section, some of the distance metrics will be introduced, specifically five metrics which are provided by the `sklearn.neighbors.DistanceMetric` library and a correlation coefficient transformed into a form of a distance metric.

Euclidean Distance

The first described is the widely used and well-known Euclidean distance. It is based on the Pythagorean theorem and can be described as the root of squared differences of coordinates of two sample objects. The Euclidean distance between two points in an n-dimensional space can be computed using the following formula: [4]

$$d(\vec{x}, \vec{y}) = \sqrt{\sum_{i=1}^{n} (x_i - y_i)^2}. \tag{4.8}$$

Manhattan Distance

The Manhattan distance represents the distance needed to be travelled from one point to the other while following a grid-like path.

The Manhattan distance between two points is computed as the sum of absolute differences of their corresponding coordinates. The distance between two points, x and y, represented by a vector of their coordinates can be obtained by using the following formula: [29]

$$d(\vec{x},\vec{y}) = \sum_{i=1}^{n} |x_i - y_i|. \tag{4.9}$$

Chebyshev Distance

Similarly to Manhattan distance, Chebyshev distance also uses the absolute values of coordinate differences. However, Chebyshev distance, instead of summing up the values, returns the maximum of all differences among the coordinates of two objects. [65]

Chebyshev distance is computed as the biggest value of absolute differences along an axis. It can be obtained using the formula: [65]

$$d(\vec{x},\vec{y}) = \max_i |x_i - y_i|. \tag{4.10}$$

Minkowski Distance

The Minkowski distance is a metric defining a distance between two points in a normed vector space. It can be considered as a generalised metric including other metrics as special cases of the generalised form. [56]

The generalisation in the Minkowski distance is created by comprising a parameter, p. The formula to compute Minkowski distance is: [10]

$$d(\vec{x},\vec{y}) = \left(\sum_{i=1}^{n} |x_i - y_i|^p\right)^{\frac{1}{p}}. \tag{4.11}$$

The value of the parameter p has a huge impact on the representation of the distance, for example: [56]

- for $p = 1$ it equals the Manhattan distance,
- for $p = 2$ it equals the Euclidean distance and
- in the limit that $p \to +\infty$, the distance equals the Chebyshev distance.

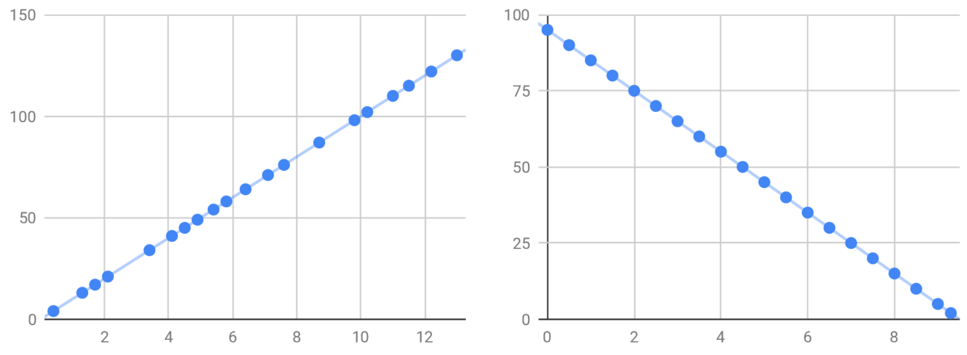
Standardized Euclidean Distance

When computing the Euclidean distance, the squared differences along all axes are summed up. In real life applications, the variables can be on completely different scales of measurement. An example of vast difference could be a database of people in a two-dimensional space with the number of children on one axis and annual salary on the other. In this case, the difference in the number of children would contribute minimally to the distance of two points in the plane.

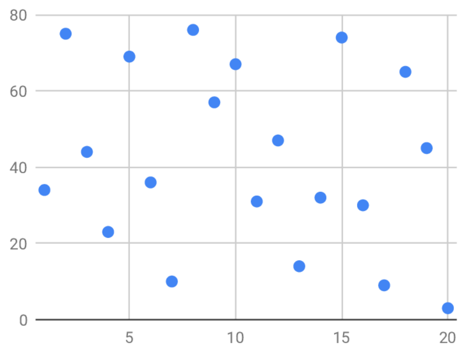
The aim of the standardized Euclidean (shortly Seuclidean) distance is to make all variables contribute to the overall distance equally. Therefore, it features data preprocessing in the form of normalization to standardize the variable variance to 1. The Seuclidean distance is then computed with the use of the same formula as for Euclidean distance. [24]

Correlation Coefficient

The last introduced metric is built on the basis of correlation of two objects, which can be obtained with the use of the Pearson correlation coefficient. The Pearson correlation coefficient is a measure of the linear relationship of two variables. Its values can range from -1 to 1, where 1 indicates a perfect positive linear correlation, -1 represents a perfect negative correlation and 0 indicates no linear relationship between the two variables. [33] Scatter plots visualising the mentioned significant values can be seen in figure 4.4.



(a) Correlation coefficient 1 (b) Correlation coefficient -1



(c) Correlation coefficient 0

Figure 4.4: Significant values of Pearson correlation coefficient (created based on the information in the article by David M. Lane [33])

In order to use the Pearson correlation coefficient as a distance metric, it is necessary to transform it into a true metric. The value of Pearson coefficient can range from -1 to 1, therefore, it is possible to transform the correlation coefficient into a distance metric by applying the following formula: [53]

$$Pearson\_distance(x, y) = 1 - correlation(x, y). \tag{4.12}$$

The defined Pearson distance falls between 0 and 2 and its value is the greater the less correlated the objects are.

4.5 k-NN

The k-Nearest Neighbors (shortly k-NN) algorithm can be used to solve both classification and regression problems. The principle of this method is similar to nearest centroid with the difference that in k-NN, the classes are not represented by their centroids yet by members of the classes themselves. When classifying an unknown sample, its k nearest neighbors are determined, which are the samples with the lowest vector distance from the unknown sample, and the object is assigned to the class which is the most common among its neighbors. [7]

The principle of k-NN classification is visualised in figure 4.5, which shows assigning a class to an unknown sample for k equal to 1 and 3.

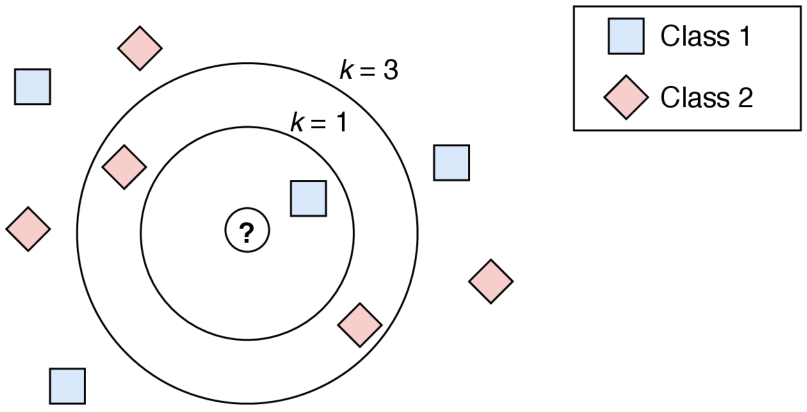


Figure 4.5: Classification with k-NN classifier for k equal to 1 and 3 (created on the basis of the article by Adi Bronshtein [7])

k-NN belongs to the category of lazy algorithms which means that it does not create any generalizations of the input data and instead it keeps all the training data for classification. Therefore, the training phase is very fast. [7]

The right value of the k parameter is very data-dependant. The best approach is to experiment with various values and find the one, which reaches the smallest error rate. In general, the smaller the value of k is, the more susceptible the classification is to noise and overfitting. On the contrary, increasing the value of k results in less distinct boundaries of the classification. [5], [26]

The biggest advantage of the k-NN classifier is that its algorithm is simple to understand and to implement. Another plus of this method is its lack of multiple parameters, which would need to be tuned. Its significant disadvantage is its high memory requirements as it stores most of the training data. Moreover, the prediction phase can be slow because the distances to all training samples are computed. [7], [26]

4.6 Decision Tree

Decision tree is another example of a very popular and easy to understand model. Similarly to k-NN, it can be also used for classification and regression tasks as well. The decision tree uses a tree-like model in which nodes represent attributes of the input data, branches are created from decision rules and leaf nodes mean categories (assigned labels). [25], [54]

During the training phase, the model is created by inferring simple decision rules from the attributes of input data. The creation of decision tree starts in its root node. All data attributes are taken into account and the training data is split into groups according to the chosen attribute. In the next step, the resulting accuracies of all possible splits are computed and the split with the best outcoming accuracy is chosen. Then, the algorithm repeats itself in the created sub-nodes recursively until the entire tree has been built. [5], [25]

An example of a decision tree for a simple problem of deciding whether to play given the input data on outlook, humidity and rain can be seen in figure 4.6.

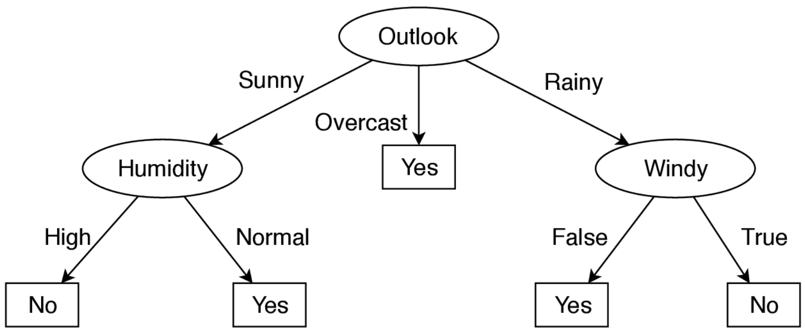


Figure 4.6: Decision tree solving a problem whether to play or not based on current weather conditions (created according to a model in the article by Madhu Sanjeevi [54])

Advantages of a decision tree model are its simplicity to understand and even visualize the model, its ability to handle both numerical and categorical data, and the fact that it requires little to no data preparation. The biggest disadvantages of this model are its susceptibility to overfitting, therefore it is important to involve pruning, instability as small variations in the data might result in creation of a completely different tree, and generation of biased trees if the classes are not balanced. [5], [25]

4.7 Random Forest

Random forest is another example of a very simple to understand and easy to use machine learning algorithm, which can also be used for both classification and regression tasks. As can be deduced from its name, it consists of multiple decision trees. The main idea behind random forest is to build various decision tree models and combine them in order to obtain more accurate prediction. [15]

The process of decision tree creation is deterministic, therefore, it is necessary to add some additional randomness to tree generation. Randomness is incorporated in two phases – a random subsample drawn from the training dataset is used to construct each tree in the ensemble and instead of finding the attribute which offers the best accuracy when splitting a node, the random forest searches for the most suitable attribute in a random subset of features. This approach results in creating a robust model with generally better accuracy. [6], [15]

The classification with the use of a random forest model can be seen in figure 4.7. There is an example of random forest consisting of three randomly created decision trees. The input data are presented to all the decision trees and classified by them. On the resulting labels, majority vote (for classification) or averaging (for regression) is then applied to get the final label for the input.

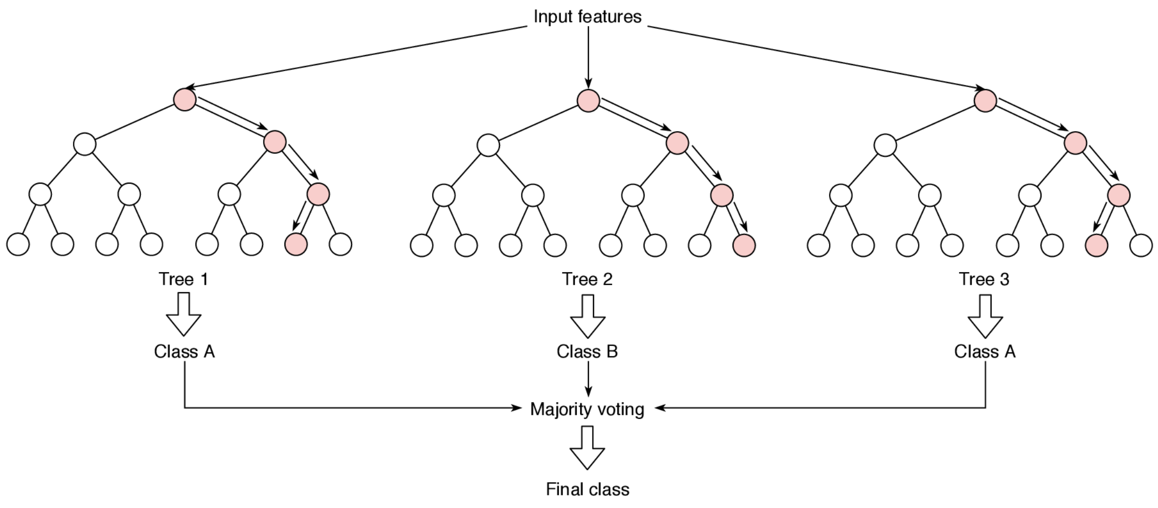


Figure 4.7: Process of classification with the use of a random forest classifier consisting of three random decision trees (created according to information in the article by Will Koehrsen [31])

An advantage of the random forest algorithm is that it is easy to use thanks to the small number of parameters to tune. Moreover, even default parameter values often lead to satisfactory results. Another pro of this method is its bigger resistance to overfitting in comparison to a simple decision tree algorithm. On the contrary, with increasing number of trees decreases the speed of training and prediction. Random forest can be quite slow especially during predictions, which can make them unusable for applications requiring real-time classification. [15]

4.8 Multilayer Perceptron

Multilayer perceptron (shortly MLP), also known as artificial neural network, is based on a joint net of perceptron layers. A single-layer perceptron is a well-known method capable of solving simple problems with data that is linearly separable into n dimensions for n being the number of attributes of the input data. The accuracy rapidly decreases, however, for data that are not linearly separable. The solution can be found in adding other layers and creating the multilayer perceptron. [51]

Multilayer perceptron consists of a net of multiple connected neurons. There is one input layer, which is represented by a set of neurons, each corresponding to one input feature. Then, there are one or more hidden layers. The neurons in the hidden layer apply a weighted linear summation on the outputs of the previous layer and then transform the results by a non-linear activation function. Lastly, there is one output layer, which converts the outputs of the last hidden layer into output variables. [5]

The schema of the structure of a single hidden layer MLP model can be seen in figure 4.8.

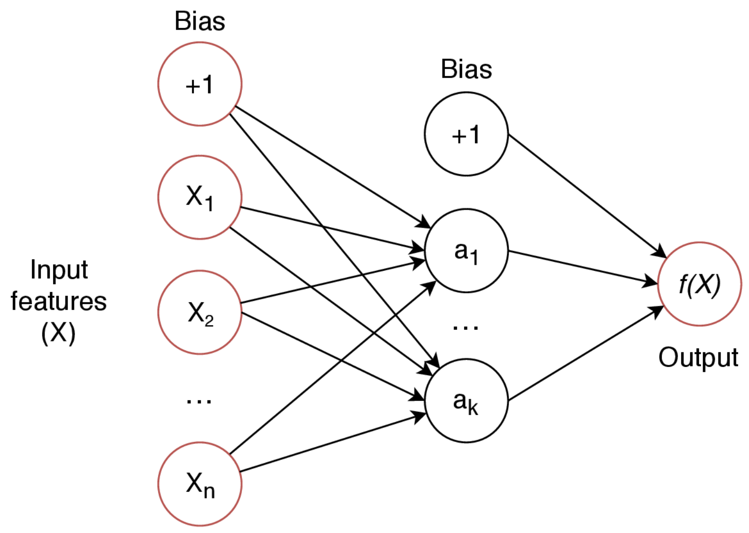


Figure 4.8: Schema of a simple MLP with one hidden layer (created on the basis of an image listed in the *scikit-learn* documentation on neural network models [57])

Neural networks are trained in cycles called *epochs*. One epoch has two phases – feed-forward and back propagation. During the feed-forward phase, a sample is presented to the input layer. The values received are passed onto connected neurons in the first hidden layer, multiplied with the weights and a bias is added to the result. On the obtained values, an activation function is applied, which can be a step function, sigmoid function or relu function. After that, the computation process is repeated until the output layer is reached. The values obtained in the last, output, layer are the outputs of the feed-forward stage. [51]

The resulting values rarely achieve satisfactory accuracy. In order to improve prediction performance, the second, back propagation, phase is involved. During back propagation, the weights of neurons are updated to make the difference between the predicted and expected output as small as possible. [51]

Back propagation consists of two steps. First, the *loss* is computed, which is the difference between the predicted and the desired output. The function used to calculate the loss is called the *loss function* and it can be a mean squared error or a cross entropy function. The aim of the second step is to minimize the calculated error. This is done by computing the gradient, which is a partial derivative of the error function. According to the derivatives, values of the individual weights are increased or decreased to reduce the overall error. The function, which aims to reduce this error, is called the optimization function. [51]

An advantage of the MLP classifier is its capability to solve non-linear problems. On the other hand, its notable disadvantage is the number of its hyperparameters (such as the number of hidden neurons and layers) which need to be tuned, and its sensitivity to feature scaling. Moreover, in MLP with hidden layers, there is a non-convex loss function which means that there exists more than one local minimum. That can lead to different validation accuracy for different random weight initializations. [5]

4.9 Naive Bayes

Naive Bayes is a set of simple yet powerful supervised learning algorithms based on the Bayes' theorem which provides a way to calculate the probability of a hypothesis given our prior knowledge. It is called naive because of the assumption of conditional independence between every pair of attributes. The Bayes' theorem can be written as: [8]

$$P(h|d) = \frac{P(d|h) \cdot P(h)}{P(d)}, \tag{4.13}$$

where $P(h|d)$ is the probability of hypothesis h under the condition of data d, $P(d|h)$ is the conditional probability of observing data d given that the hypothesis h is true, $P(h)$ and $P(d)$ are the prior probabilities of the hypothesis h and of the data respectively. [8]

Training of a naive Bayes model, which consists in deriving conditional probabilities from training data, is fast and there are no parameters to be fitted. Even though the assumptions of the Bayes model are simplified, it has proven to work well in many real-life applications. [8]

The naive Bayes classifier can be extended to data with real-valued attributes. This can be achieved by applying a function to estimate the data distribution. The easiest way to do so is to use the Gaussian distribution. The classifier is then called Gaussian naive Bayes. Another possibility is to use multinomial naive Bayes, which is the naive Bayes algorithm for data with the multinomial distribution. [8]

Chapter 5

Existing Tools

Some methods of bacteria classification, which have already been implemented, are described in more detail in this chapter. All of the methods introduced in this chapter extract k-mer spectra from the input sequences before applying the classification models.

Section 5.1 is devoted to the description of the RDP classifier which utilises the naive Bayesian classifier. Section 5.2 introduces a set of tools implemented within a bioinformatic pipeline called QIIME. Most of the techniques are built on the basis of k-NN classifier and apply various approaches to search for the nearest neighbours. In section 5.3, another solution named microclass, which is based on the naive Bayes classifier, is presented. Section 5.4 is focused on the 16S Classifier that creates a random forest classification model. The aim of section 5.5 is to describe the SINTAX algorithm, which also utilises the principle of nearest neighbours. The last section, 5.6, introduces the IDTAXA method which is based on the principle of tree classification.

Some of the introduced algorithms are used for comparison with the proposed method and the results can be seen in chapter 8.

5.1 RDP Classifier

One approach of solving bacteria classification has been introduced by Wang et al. in their article *Naive Bayesian Classifier for Rapid Assignment of rRNA Sequences into the New Bacterial Taxonomy* [68]. In this work, they described the RDP classifier which extracts k-mer spectra from the input sequences and then applies the naive Bayesian classifier to assign the unknown specimens their genera. Regarding k-mer size, they achieved the best accuracy with k set to 8 and 9 and decided to use k-mer size 8 to reduce memory requirements.

5.2 QIIME and QIIME 2 Pipelines

Some other existing solutions are implemented as a part of QIIME, a bioinformatic pipeline designed for analysing microbiome from raw DNA sequencing data. [48]

One of the methods implemented within QIIME is called USEARCH LCA. This method is based on the k-NN classifier. It uses a sequence database search algorithm that seeks high-scoring global alignments named USEARCH [19] for finding k sequences which are the most similar to the given sequence and whose taxonomy is known. Then, on their taxonomic classifications, the LCA [35] algorithm is applied to obtain the taxonomy of the unknown sequence.

Another approach is implemented in QIIME 2 and it is named BLAST LCA. The principle of this method is the same as in the previous algorithm with the only change in the search algorithm. In this method, the BLAST [42] search algorithm is used.

Last mentioned method is QIIME BLAST TOP HIT, which basically represents the k-NN algorithm with *k* set to 1. It uses the BLAST algorithm for finding the nearest neighbour and assigns the unknown sequence the taxonomy of the nearest sample. [12]

5.3 microclass

Another solution called microclass is available as an R package and while it has a standard R interface, its computational core is implemented in C++ (for example the extraction of k-mer spectra) to reduce time consumption. After experimenting with various k-mer spectra based classification methods, the authors decided to use the naive Bayes classifier. The same classifier type is used in the RDP method, however, while RDP only considers the presence of k-mers, in microclass, the abundance of k-mers is used. [37]

The package offers the possibility of creating a custom model by training a new model on an input dataset and classification of unknown samples by the previously created model. It also offers a ready-to-use pre-trained classification tool, which uses k-mer size 8 and has been trained on full-length 16S rRNA sequences. K-mer of size 8 was chosen since its increase to 9 or 10 results in a high cost in memory consumption and computation time while the gain in accuracy on the genus level is small. [37]

The authors have compared the presented method to classification based on the BLAST algorithm. The executed experiments proved that this method is both slower and less accurate than the proposed method. [37]

5.4 16S Classifier

16S Classifier is an example of more recent approaches. It is based on a random forest classification model and uses only the hypervariable regions of the 16S rRNA in order to increase speed and prediction accuracy. [13]

The authors decided to use random forest classifier for its quick and easy implementation, ability to deal with large datasets thanks to its robust classification algorithm, and high accuracy it can offer. Furthermore, it offers the possibility to be presented a large number of input variables and still prevent overfitting. The authors also applied bootstrapping to grow classification trees in the random forest with the use of the training data. [13]

During optimisations, the authors came to a conclusion that performances of 2-mer and 3-mer models offered the lowest accuracy. 5-mer and 6-mer models gave results with the lowest error, however, the 4-mer model needed significantly less time to prepare a model and smaller size of training data. Therefore, the authors decided to use 4 as the k-mer size. The authors also examined the impact of the number of decision trees on the resulting accuracy. To do so, they gradually increased the number of trees up to 1,000 and noticed a gradual increase in prediction accuracy, therefore, they decided to generate 1,000 trees when creating a random forest model. [13]

5.5 SINTAX

The SINTAX taxonomy classifier, which is introduced in article *SINTAX: a simple non-Bayesian taxonomy classifier for 16S and ITS sequences* by Robert C. Edgar, is focused on the classification of ribosomal 16S gene and fungal ITS (Internal Transcribed Space) regions. Similarly to other existing methods, it also extracts k-mer spectra from the input sequences and uses k set to 8 by default. The basic principle of this algorithm is simple, after the k-mer spectrum has been obtained, it uses k-mer similarity to find the top hit in a reference database. Along with the predicted taxonomy, it provides also bootstrap confidence for all taxonomic ranks in the predicted classification, which can be represented as a list of reference taxonomies with their k-mer similarities to the classified sequence. [17]

During the comparison of the SINTAX algorithm to the RDP classifier, it achieved better accuracy on full-length 16S and ITS sequences and comparable results on the V4 region of 16S rRNA. Moreover, SINTAX features a simpler algorithm than the RDP naive Bayesian classifier and does not require training. [17]

5.6 IDTAXA

The IDTAXA algorithm was presented by Murali et al. in their article *IDTAXA: a novel approach for accurate taxonomic classification of microbiome sequences.* It aims to propose a different approach to the taxonomic classification which utilises machine learning principles and manages to reduce overclassification errors. The IDTAXA algorithm consists of tho phases – training and new input sequences classification. [41]

Training takes reference sequences and their corresponding taxonomic classifications as input and starts with extracting k-mer spectra from them. By default, the value of k is chosen according to the length of input sequences. For example for an input dataset of full-length 16S rRNA gene sequences, the k would be set to 8. During k-mer spectra generation, the non-nucleotide characters are omitted. After that, the top 10 % of k-mer spectra that distinguish among the subgroups at each taxonomic rank the best are determined according to the differences of the frequency of each k-mer relative to other k-mers in a subgroup and the frequency of the same k-mer relative to other k-mers in its parent group which is computed based on the cross-entropy. [41]

The output object of training is taken as the input of input sequence classification together with a set of unknown sequences to be classified. From these sequences, the k-mer spectra are extracted and then classifier by descending through the taxonomic ranks. After finishing the classification process, the classification process returns the predicted classification for each input sequence in the form of the taxonomic assignment together with confidences for the corresponding taxonomic ranks. [41]

The authors managed to obtain higher accuracy with the IDTAXA algorithm than other popular classifiers, such as BLAST, QIIME, SINTAX, and the RDP Classifier.

Chapter 6

Proposed Bacteria Classification Method Specification

This chapter contains a detailed description of the proposed bacteria classification method. In each section, a part of the whole algorithm is specified, starting with k-mer spectra extraction from the input 16S rRNA sequences and continuing to classification tree creation, training and evaluation.

Section 6.1 focuses on k-mer spectra extraction and the importance of this step for further processing. It also addresses the impact of the value of k on classification. In section 6.2, the process of generation of the tree of classifiers is introduced together with the reason for using this structure of classifiers. Section 6.3 is devoted to the specification of a new type of classifier, the NMDK classifier, which is proposed and implemented as a part of this work. Section 6.4 features algorithm of the presented NMDK classifier.

6.1 K-mer Spectra Extraction

This method is designed to classify bacteria according to the sequence of their 16S rRNA gene. The 16S rRNA sequence can be found in the genome of all bacteria species and it differs enough among various genera so that it could be a good candidate for determining the type of bacteria. The 16S rRNA sequence can contain multiple mutations, insertions and deletions, therefore, various 16S rRNA sequences can be of different length. This could cause inconvenience as machine learning algorithms require their input vectors to be of equal dimensions. To overcome these difficulties, a k-mer spectrum is extracted from the input sequence and used for classification afterwards. With the use of k-mer spectra, it is possible to transform every 16S rRNA sequence, which is in the form of a string, into a numeric vector, where each value represents the number of occurrences of the corresponding k-mer in the original sequence.

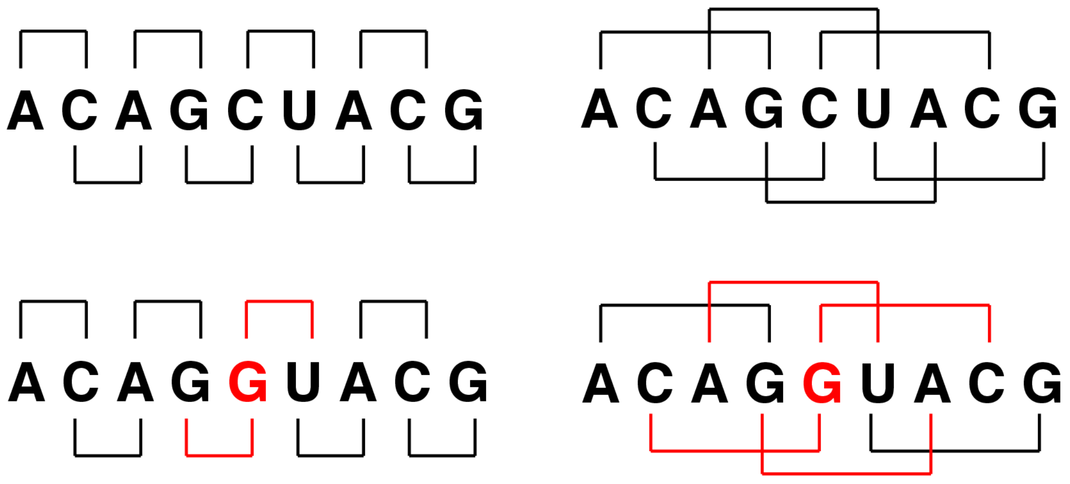
K-mer spectra extracted from the 16S rRNA sequences of all bacteria species have the same predetermined length, which only relies on the k-mer size and number of nucleotide characters used (in the equation 3.3 in section 3.2, the count of nucleotides would represent the variable n). There are four nucleotides, which are being found in an RNA of a bacteria – adenine (A), cytosine (C), guanine (G), and uracil (U). However, other characters are also frequently present in the rRNA sequences in various databases. These characters (similarly to regular expressions) represent two or more bases, e.g. Y, which can mean either cytosine or uracil [63]. To include also these characters and avoid unnecessary loss of information, the proposed classifier deals with substrings which contain one or more non-nucleotide characters by transforming them into all possible real nucleotide sequences (created according to substitution table defined by the IUPAC federation [63]) and incrementing the count of occurrences of the corresponding k-mers.

Moreover, transforming the 16S rRNA gene sequences into k-mer spectra offers the possibility of experimenting with various k-mer sizes. The smaller size of k-mer is capable of extracting less information than larger k-mer size, on the contrary, the extracted k-mer spectrum is less affected by the mutations in genes (explanation of this statement can be found in section 6.1.1).

6.1.1 Impact of K-mer Size

As mentioned in the previous section, smaller k-mer size leads to extracting less information from the sequence in comparison to bigger k-mer sizes, meaning there are fewer distinct subsequences being counted in the gene sequence when using a smaller k-mer size and it is not possible to capture the occurrence of that many subsequent characters. For example, consider two subsequences of “CAGC” and “GCCA”. When using a k-mer spectrum with size 4, one sequence motif is extracted from each subsequence and the two subsequences are easily distinguishable. However, for a k-mer spectrum of size 2, there are two sequence motifs found in each subsequence (“CA” and “GC”) and the obtained k-mer spectra are the same for both subsequences.

The second idea mentioned was that the bigger the k-mer size is, the more the extracted k-mer spectrum is affected by a single gene mutation. The idea behind this statement is explained in figure 6.1, which shows a comparison of k-mer sizes 2 and 4. The example contains a part of a genome sequence in which the fifth base has been mutated from cytosine to guanine. For the k-mer size of 2, the mutated gene changes only two values (for subsequences “GC” and “CU”) by one. When using the k-mer size of 4, however, there is a total of four affected values (for subsequences “CAGC”, “AGCU”, “GCUA”, and “CUAC”) by this single mutation.



(a) K-mer size 2 (b) K-mer size 4

Figure 6.1: Impact of mutations using various k-mer sizes

6.2 Classification Tree

This classification tool is designed to assign unknown bacteria to their most probable genera. In order to minimise the classification error, the presented method is based on a tree structure of classifiers corresponding to the taxonomic tree. The whole classifier is decomposed into multiple classifiers (referred to as partial classifiers) on all levels of taxonomy from domain down to genus. Thanks to the use of the taxonomic tree structure and the concept of successive classification it should be possible to decrease the overall classification error in comparison to direct classification on the lowest taxonomic level since every classifier distinguishes only among a few classes on the lower level of the taxonomy. This may contribute to increased accuracy since with an increasing number of predicted classes increases the probability of overlapping of the individual classes. The comparison of accuracy obtained by the tree structure of classifiers and direct genus assignment with the use of only one classifier can be seen in section 8.4. The introduced approach also offers the possibility of presenting the whole taxonomic classification from domain down to genus with values of reliability for predicted labels on every taxonomic level.

The tree structure of partial classifiers is shown in figure 6.2. Every rectangle represents a single classifier of one of the well-known classifier types, such as SVM, nearest centroid, decision tree and random forest. The labels in them show the classification of the input gene sequence on the corresponding level of the taxonomy. An example classification of *Escherichia coli* is given. The partial classifiers are connected with arrows indicating the order of sequence classification through the whole tree of classifiers.

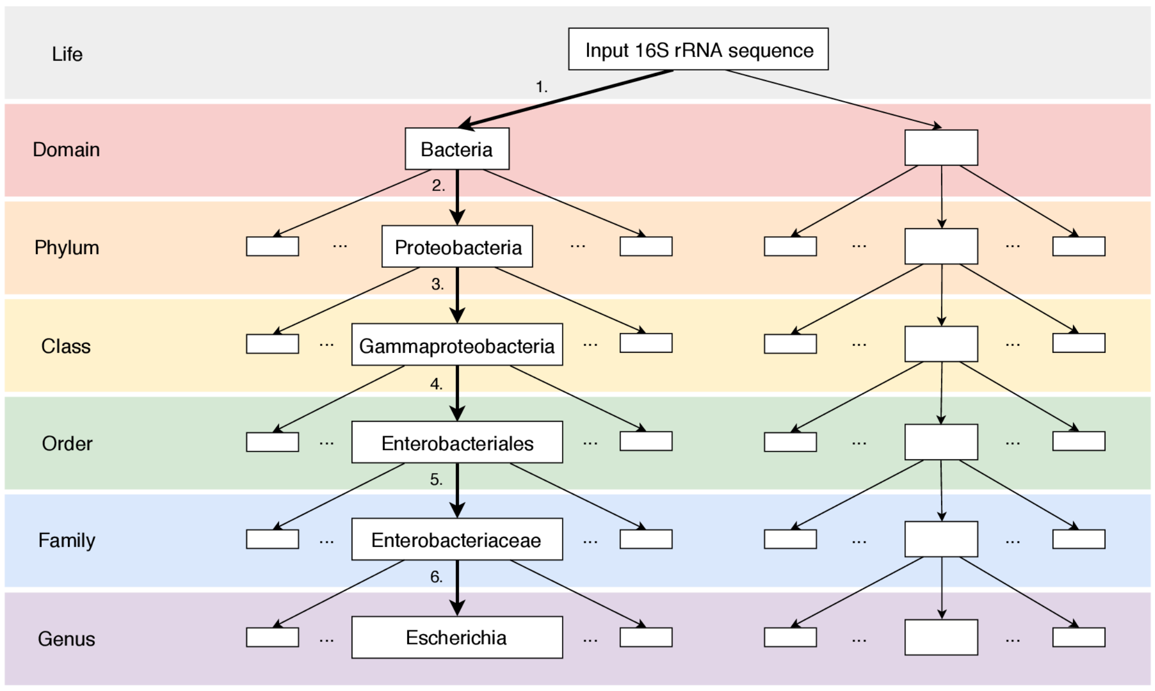


Figure 6.2: Tree structure of partial classifiers created according to the taxonomic tree with emphasised classification order

With this method, it is possible to obtain a complete taxonomic classification from domain to genus and, in case of an unsuccessful classification, determine the exact partial classifier which caused the error.

Diagram of training of the presented tree structure of classifiers can be seen in figure 6.3. The training method consists of two phases. The aim of the first phase is to determine the best performing classifier setting among all evaluated settings. In order to compare the accuracies of the individual settings, all types of classifiers with various settings compete against each other. To execute the competition, the entire process of tree creation, training and validation is wrapped inside cross-validation, which represents the outer loop of the entire training method. In the diagram, the outer loop is represented by the five divisions of the input dataset, each of which is the input for one outer loop iteration. The inner loop, which takes place inside every outer loop iteration and is visualised as the rectangle at the top of the diagram, represents the grid search. It is executed iteratively for every classifier setting in all available settings and consists of three steps – building a tree of classifiers of the given type, training and validation. The outcome of validation in every inner loop iteration is a list of accuracies achieved by the current classifier setting on every taxonomic level. These values are then used for comparison of multiple classifier settings.

After the entire cross-validation process, the best performing classifier is determined, which is the output of the first training phase and input of the second training phase. In the second training phase, which can be seen at the bottom of the diagram, a new classifier tree composed of the determined classifier type is generated, trained on all available data and stored as the final model for further use. With this approach, it is possible to obtain the best performing classifier for every dataset used.

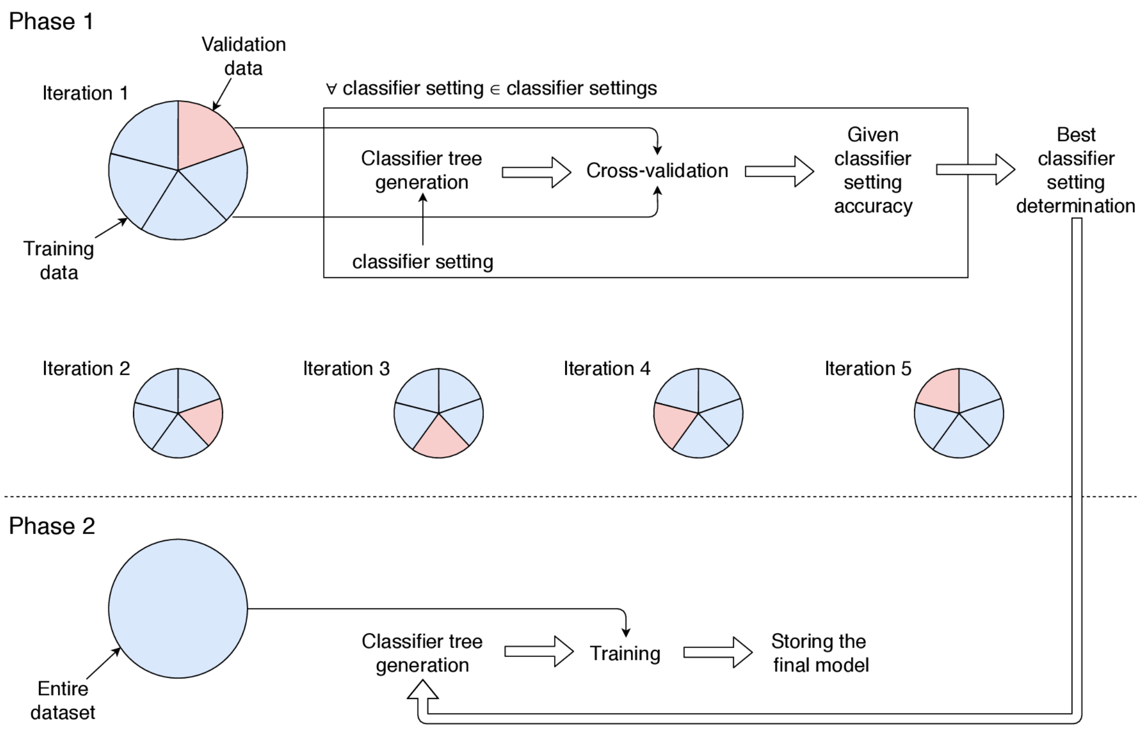


Figure 6.3: Diagram of of the two-phase training method of the presented tree classifier

Training of each partial classifier is executed separately and independently on other partial classifiers and it should be executed on a subset of training data, only on those rRNA sequences, which belong to the taxon of the classifier.

To train the model as well as to validate the accuracy of specimen taxonomy prediction, a reference database containing sequences with taxonomy annotations is required. The databases used to train models and validate their accuracies are described in section 8.2.

The process of rRNA gene sequence classification can be seen in figure 6.4, in which arrows show the order of processing and dashed arrows represent alternative routes.

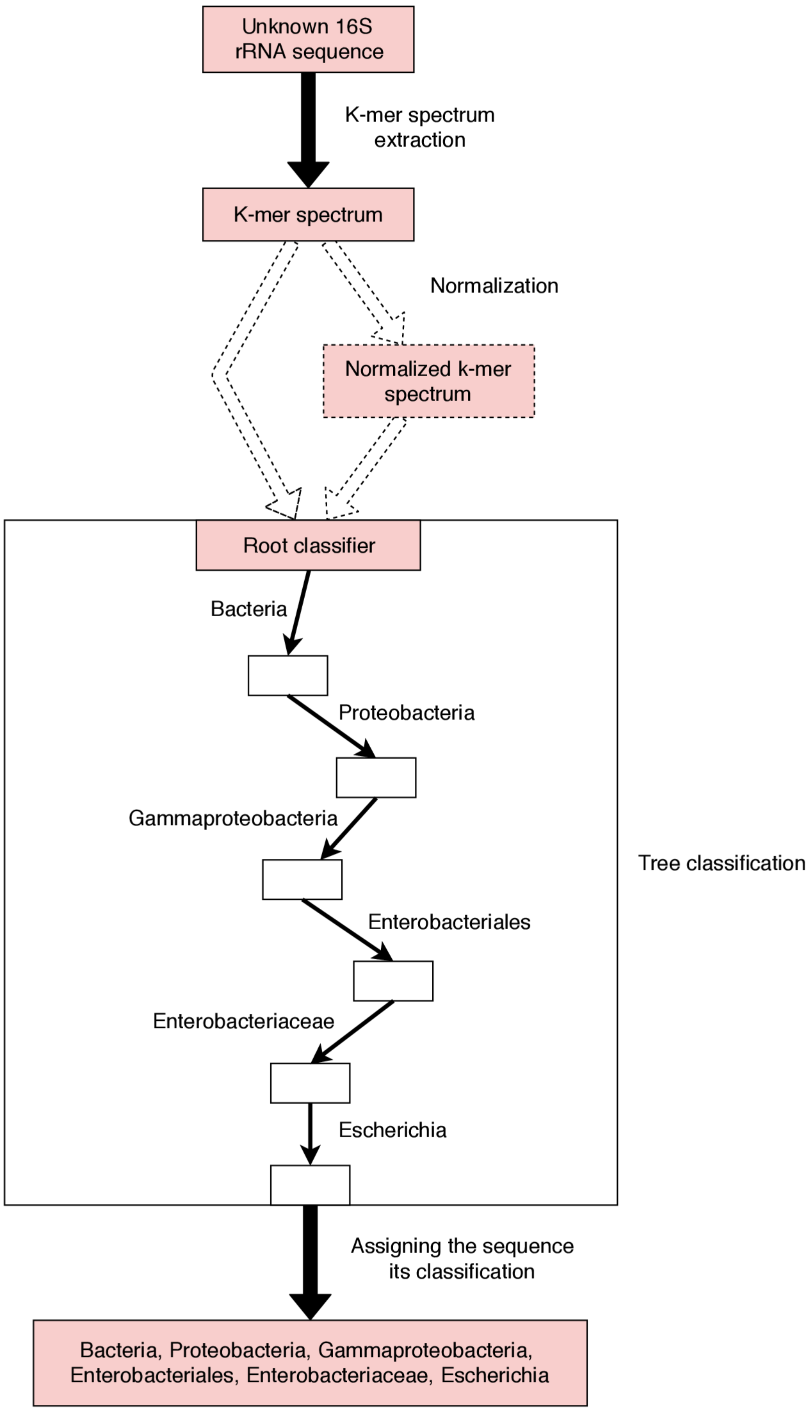

Unknown 16S rRNA sequence
K-mer spectrum extraction
K-mer spectrum
Normalization
Normalized k-mer spectrum
Root classifier
Bacteria
Proteobacteria
Gammaproteobacteria
Enterobacteriales
Enterobacteriaceae
Escherichia
Tree classification
Assigning the sequence its classification
Bacteria, Proteobacteria, Gammaproteobacteria, Enterobacteriales, Enterobacteriaceae, Escherichia

Figure 6.4: Diagram of unknown input sequence classification with the use of the presented KTC classifier

The procedure of classification is initiated by obtaining a k-mer spectrum from the input sequence. After that, the k-mer spectrum is optionally normalized and presented to the top classifier, which categorises it as either a bacteria or an archaea. Afterwards, the chosen one of the two classifiers on the domain level is used to assign the input spectrum its phylum. Then, the classifier belonging to the assigned phylum is used to classify the input into one of the connected classes and this process repeats itself until the final classification, genus, is assigned, which consists of the path from the root classifier to the bottom classifier.

6.3 N Most Distinguishing K-mers (NMDK) Classifier

The N Most Distinguishing K-mers (NMDK) classifier is the proposed type of classifier which aims to offer a solution with good accuracy and to achieve a significant decrease in memory requirements and time consumption thanks to dimensionality reduction. The basic principle of the introduced classifier is very similar to the nearest centroid algorithm. Every taxonomic classification in the classifier tree is represented by the *N most distinguishing* positions of an average k-mer spectrum of the given taxon.

In machine learning, the *curse of dimensionality* is a commonly known issue resulting in slow training and evaluation phases as well as extremely large memory requirements. It can also make the final classifier too complex without a notable increase in accuracy as shows the Hughes Phenomenon. It states that with an increasing number of attributes, the classifier performance also increases, however, only until the optimal number of attributes is reached. Including additional features then only decreases the classifier's performance. [60]

The NMDK classifier tries to combat this problem by selecting only a given number of the input features and in order to increase prediction accuracy, it omits attributes that are not of great importance for distinguishing among the given classes.

Training an NMDK node classifier starts with computing the average k-mer spectra of the node and of all classes on the lower level of the taxonomy. These k-mer spectra are computed from those samples in the training data which belong to the given taxon. The average k-mer spectrum of a class is computed using one of the two methods. It is either represented by the mean k-mer spectrum of the given class or by its median, which is not as susceptible to extreme values as mean. Then, the average k-mer spectra are reduced to only the N most distinguishing positions. To do so, two principles are applied – joint and separate.

In the first (joint) approach, only one set of the N most distinguishing positions of the k-mer spectra is determined for the current node. That means that the absolute differences among the k-mer spectrum of the node and all k-mer spectra of the lower taxonomic classifications are summed for each feature and the N attributes, which have the highest overall difference, are chosen as the most distinguishing ones. The differences for each attribute can be computed using formula 6.1, where L is the length of a k-mer spectrum (number of all features), n is the count of classes at the lower level of taxonomy, $\vec{x}_1$ to $\vec{x}_n$ are the average k-mer spectra of classes at the lower taxonomic level and $\vec{y}$ is the average k-mer spectrum of the current node.

The second (separate) approach is to determine the N most distinguishing positions separately for every category at the lower taxonomic level. The differences among the k-mer spectrum of the node and all k-mer spectra of lower taxonomic classifications are computed separately and for every lower taxonomic class, the N attributes which have the highest difference from the average k-mer spectrum of the upper node are chosen as the most distinguishing ones for the given taxonomic category. The difference between the average k-mer spectrum of the current node and of one of its subclasses is computed using the formula 6.2, where L is the length of a k-mer spectrum, and $\vec{x}$ and $\vec{y}$ are the examined average k-mer spectra.

$$\forall i \in \langle 1, L\rangle : differences_i(\vec{x}_1, \vec{x}_2, \ldots, \vec{x}_n, \vec{y}) = \sum_{j=1}^{n} |x_{j_i} - y_i| \tag{6.1}$$

$$\forall i \in \langle 1, L\rangle : differences_i(\vec{x}, \vec{y}) = |x_i - y_i| \tag{6.2}$$

Classification of an unknown sample is then done by computing the absolute difference of values in the N most distinguishing positions and the NMDK spectra of all lower taxonomic classifications, and the classified sample is assigned to the class of the lower taxonomic label with which it obtained the lowest summed difference. The index of the class which will be assigned to the unknown specimen can be computed using the following formula:

$$k\text{–}mersDifference(\vec{x}, \vec{y}) = \arg \min_{\vec{x} \in \{\vec{x}_1, \ldots, \vec{x}_n\}} \Big(\sum_{i=1}^{|\vec{x}|} |x_i - y_i| \Big), \tag{6.3}$$

where $\vec{x}_1$ to $\vec{x}_n$ are the NMDK spectra of classes at the lower taxonomic level and $\vec{y}$ is the NMDK spectrum of the classified sample.

Evaluation of all variations of the NMDK classifier prediction accuracy can be found in section 8.5.

6.4 The NMDK Classifier Algorithm

In order to implement and be able to utilise the proposed NMDK classifier, two functions need to be specified – one for training, which is called `train`, and another one for classification of unknown input sequences named `classify`.

Algorithm of the first mentioned function of the NMDK classifier, `train`, can be seen in algorithm 1. It takes two parameters – X representing the input data for training, and y in which are passed labels corresponding to the training data. First, the shapes of data and labels are checked whether they are valid in function `check_X_y`. In the next step, the average k-mer spectrum of the current node is computed using the chosen metric – mean or median. After that, unique labels are stored as names of classes at the lower taxonomic level. Next, in function `ComputeAverageSpectra`, the average k-mer spectra for nodes at the lower taxonomic level are computed. In function `DetermineDivergentIndices`, the N most divergent indices in k-mer spectra are obtained and the corresponding most distinguishing k-mer spectra for every subclass are acquired in function `GetDivergentKmers`. Finally, the created classifier instance is returned.

The second mentioned function of the NMDK classifier is called `classify` and its algorithm is listed in algorithm 2. This function accepts only one parameter – X, which contains the unknown 16S rRNA sequences. Firstly, it is checked whether this classifier object has been previously fitted. After that, the shape of input samples is validated in function `check_array`. In the next step, the input k-mer spectra are reduced to the most divergent k-mer spectra. Subsequently, for every input specimen, the differences from classes on the lower taxonomic level are computed and the class which is the closest to the currently

Algorithm 1 Training of the NMDK classifier

```
procedure $train(X, y)$
    X, y = check_X_y(X, y)
    averageKmer = averageMetric(X, axis=0)
    classes = unique_labels(y)
    averageSpectra = ComputeAverageSpectra(X, y)
    divergentIndices = DetermineDivergentIndices(averageKmer, averageSpectra)
    divergentKmers = GetDivergentKmers(averageSpectra)
    return classes, divergentIndices, divergentKmers
```

examined sample is added to the list of predicted labels. When all input samples have been classified, the list of predicted labels for all classified specimens is returned.

Algorithm 2 Unknown specimens classification using the NMDK classifier

```
procedure $classify(X)$
    check_is_fitted(['classes', 'divergentIndices', 'divergentKmers'])
    X = check_array(X)
    divergentKmers = ExtractDivergentKmers(X, divergentIndices)
    predictedLabels = []
    for specimen in divergentKmers do
        differences = ComputeDifferencesFromSubclasses(specimen)
        closest = GetLowestDifferenceIdx(differences)
        predictedLabels.append(classes[closest])
    return predictedLabels
```

Chapter 7

Implementation of the K-mer Tree Classifier (KTC)

This chapter focuses in detail on the description of the final implementation of the KTC (K-mer Tree Classifier) application introduced in chapter 6. The application is implemented with the use of object-oriented programming. The entire application consists of multiple classes, each of them is implemented in a dedicated module. There are also some additional modules which are used for Cython compilation. An overview of the modules, their purpose and noteworthy methods implemented in them can be seen in appendix B.

The NMDK classifier, which was introduced and described in section 6.3, is implemented with the same interface as other classifiers imported from the *scikit-learn* library so that it is possible to handle all classifier types the same way. Therefore, it was necessary to implement the `fit` method for training and the `predict` method for unknown samples classification and to preserve the methods' declarations given by *scikit-learn* classifiers. The `fit` method is implemented according to algorithm of the `train` function and the `predict` method is written based on the `classify` function, both of them listed in section 6.4.

In section 7.1, there are mentioned requirements and dependencies, which includes primarily the used tools and libraries. The next section 7.2 is devoted to the way the input parameters of the application are solved and to the specification of the format of input files containing the 16S rRNA sequences. The schema of communication of the classes implemented within these modules is shown in section 7.3. Finally, two extensions of the application, which have been implemented beyond specification and aim to increase prediction accuracy, are introduced and described in section 7.4.

7.1 Requirements and Dependencies

The described KTC application is implemented in Python 3 scripting language and compiled using *Cython* static compiler [14] to increase its performance.

Another necessity is the *numpy* library providing an efficient implementation of a multidimensional array object and advanced broadcasted operations. [45]

The last required component is *scikit-learn*, which is an open source library implementing simple and efficient tools for data analysis [46]. From this library, classes of the mentioned classifier types are imported.

7.2 Application Inputs

On the basis of the application design, controlling of the KTC application is implemented through the values of the parameters with which the application is executed. There are three main parameters, each representing one of the tree phases – preprocessing (`--preprocess`), training (`--train`) and evaluation (`--evaluate`). Parameter `--dataset` sets the location of the file with 16S rRNA sequences to train on or to classify, depending on which one of the previous parameters is set. The name of the model to be stored after training or loaded for classification is given with the `--model` parameter. K-mer size used for training a model is specified using the `--kmer-size` parameter. If the last parameter, `--subsample`, is set, only a given number of samples chosen randomly from the dataset is used for preprocessing or training.

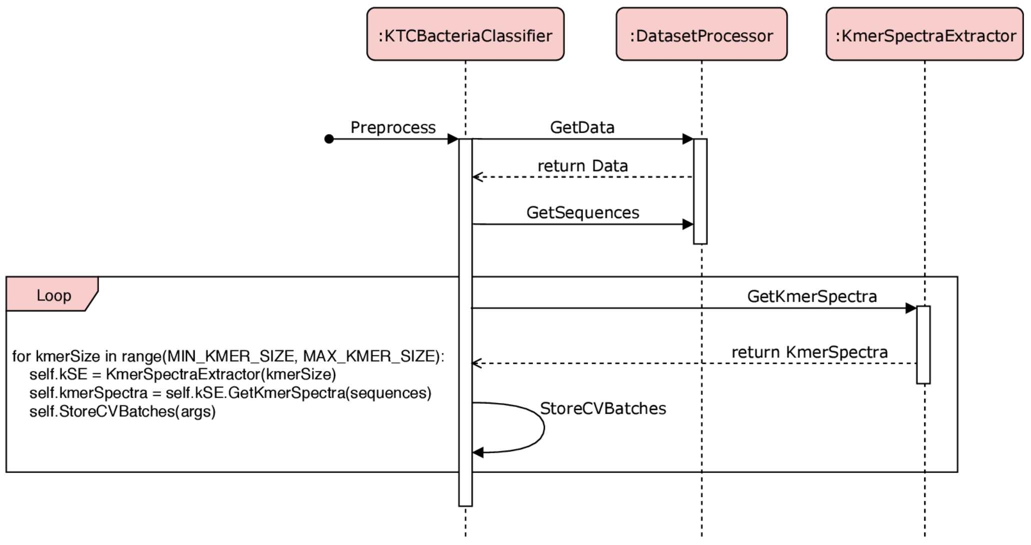
Default values of the mentioned parameters, valid parameter options and examples of the application's usage are listed in README file which can be found on the attached storage media.

The input datasets containing 16S rRNA samples for training can be in one of the two formats – JSON and FASTA, and the input for evaluation is expected in plain text format with every unknown sequence on a new line.

7.3 Schema of Class Communication

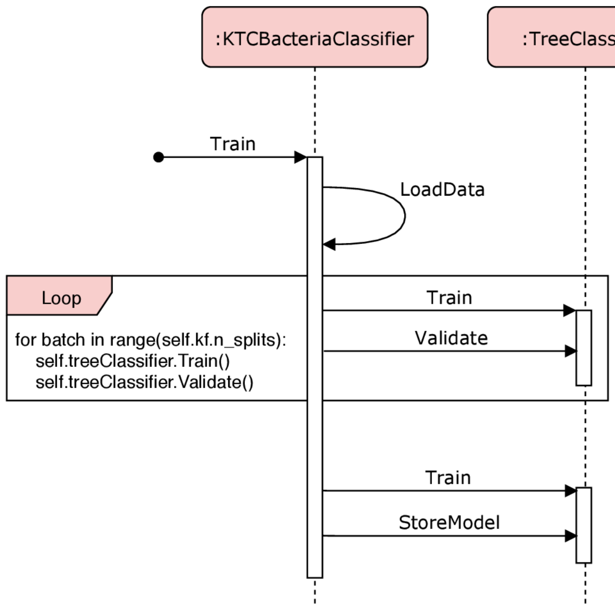
Schema of communication of the implemented classes can be seen in figure 7.1. There are three diagrams, each representing communication during one of the phases: preprocessing, training and evaluation.

During preprocessing, the `KTCBacteriaClassifier` class instance asks the `Dataset-Processor` to open file with input dataset (`GetData`) and process its content (`GetSequences`). After the dataset has been loaded into its inner representation, the `GetKmerSpectra` method of `KmerSpectraExtractor` is evoked in order to extract k-mer spectra from the input 16S rRNA gene sequences. Then, the created k-mer spectra and their corresponding labels are stored to persistent files in `StoreCVBatches` method. The last three steps are executed in a for loop iterating over all used k-mer sizes.

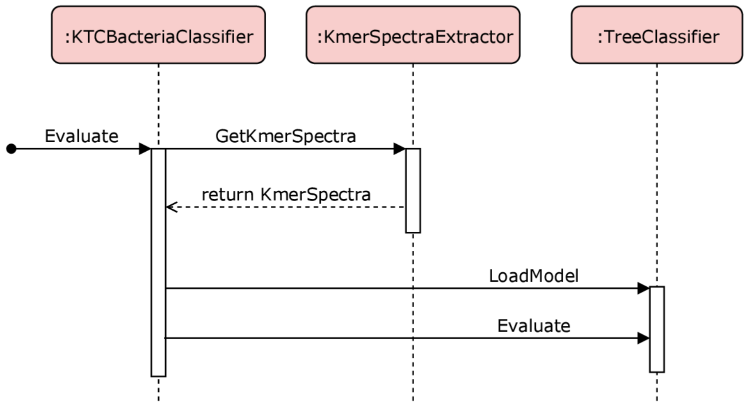


```
for kmerSize in range(MIN_KMER_SIZE, MAX_KMER_SIZE):
    self.kSE = KmerSpectraExtractor(kmerSize)
    self.kmerSpectra = self.kSE.GetKmerSpectra(sequences)
    self.StoreCVBatches(args)
```

(a) Preprocessing


:KTCBacteriaClassifier
:TreeClassifier
Train
LoadData
Loop
for batch in range(self.kf.n_splits):
self.treeClassifier.Train()
self.treeClassifier.Validate()
Train
Validate
Train
StoreModel

(b) Training



(c) Evaluation

Figure 7.1: Sequence diagrams of communication of implemented classes

During the training phase, the entire training procedure is executed. First, the k-fold cross-validation takes place, every iteration of which consists of training (`Train`) and validation (`Validate`) evoked from the `TreeClassifier` class instance. After that, the results from validation are combined to determine the most accurate classifier type and setting, which is then used for final training and validation only of the winning classifier. The final step is then storing the model into a persistent file in the `StoreModel` method.

Evaluation phase starts with evoking the `GetKmerSpectra` method of `KmerSpectraExtractor` class which extracts k-mer spectra from the unknown sequences. In the next step, the previously stored model is loaded in the `LoadModel` method evoked from `TreeClassifier` class instance. When the k-mer spectra to be classified have been obtained and the model to classify the k-mer spectra with has been loaded, the `Evaluate` method is evoked, where the unknown samples are classified by descending through the tree of classifiers.

7.4 Extensions Implemented Beyond Specification

During the implementation phase, additional extensions have been proposed which are now part of the application. Both of them are realised as added parameters and the application can be run both with their use and the standard way. In this section, the parameters will be introduced and their importance and principle of work will be explained.

7.4.1 Parameter `normalize`

The first introduced extension is the `normalize` parameter which was designed to improve the accuracy of the classification. When it is set, the values of all k-mer spectra are normalized to the range $\langle 0, 1 \rangle$.

Normalization is most important when dealing with samples whose features are on different scales. In this case, the values among features could vary since some subsequences might be more frequent than others. Normalization could, therefore, increase the accuracy of methods which are susceptible to different ranges of variables, such as SVM.

The examination of the impact of k-mer spectra normalization can be seen in section 8.7.

7.4.2 Parameter `regions`

The second described extension is the `regions` parameter. The entire 16S rRNA consists of two types of regions, some are highly variable and others are conserved and change very slowly. Therefore, creating k-mer spectra only from one or more extracted variable regions without the impact of the conserved portions might lead to an increase in prediction accuracy.

The proposed application can be presented an input dataset containing only one or more variable regions, then the specification of no additional parameter is needed, or it can have entire 16S rRNA sequences as input and when the `regions` parameter is given, the application extracts the defined variable regions with the use of the V-Xtractor tool, which is an open-source library for identifying and extracting hypervariable regions with the use of Hidden Markov Models to detect the conserved boundaries [27].

Since there are nine variable regions in a 16S rRNA sequence [1], it is possible to use only one at a time and then determine the variable region, which offers the best distinguishing capabilities or experiment with various combinations of the regions.

The impact of the regions extraction and examination of the best performing regions is described in detail in section 8.8.

Chapter 8

Evaluation of the KTC Application

The aim of this chapter is to evaluate the implemented K-mer Tree Classifier (KTC) application and to examine the impact of the values of its parameters on the prediction accuracy. This chapter also features a comparison of the proposed classifier with existing tools and its final expected accuracy evaluated on a validation dataset.

Section 8.1 presents an overview of all classifier settings which take part in the competition of classifiers during the first training phase of the proposed KTC application. In section 8.2, there are briefly introduced the reference 16S rRNA databases that were used to train and evaluate the KTC classifier and to obtain the results listed in this chapter.

In each of sections 8.3 to 8.8, the impact of one aspect of the KTC application on prediction accuracy is examined. Section 8.3 is dedicated to the examination of approaches for solving the problem of non-nucleotide characters. Section 8.4 is devoted to the comparison of prediction accuracy of the tree structure of classifiers presented in section 6.2 and direct genus assignment using only one classifier. Section 8.5 focuses on the NMDK classifier introduced in section 6.3 and on the evaluation of its accuracy using various averaging metrics, count of most divergent attributes extracted from k-mer spectra and of joint and separate approaches. Section 8.6 presents the impact of the used k-mer size on prediction accuracy. In section 8.7, the impact of applying normalization is examined. Section 8.8 features an examination of accuracies obtained with the use of individual variable regions of the 16S rRNA gene.

Section 8.9 presents the comparison of accuracies of the classifier types used within the KTC application. In section 8.10, there is a comparison of the presented application with other existing tools. Lastly, section 8.11 describes the resulting accuracy of the implemented classifier for multiple reference databases.

The box plots listed in this chapter were created (if not explicitly stated otherwise) with values that were obtained as the resulting accuracies of validation during 5-fold cross-validation. The output of every iteration of the cross-validation was a list of accuracies – six values representing accuracies on corresponding taxonomic levels for every classifier setting. For every classifier setting, the six values representing accuracies on individual taxonomic levels were used to compute the Area Under a Curve (AUC) value using the following formula:

$$AUC(classifier\_setting) = \frac{1}{k} \cdot \sum_{i=1}^{k} accuracy(classifier\_setting)_i, \qquad (8.1)$$

where *classifier_setting* represents the classifier setting whose AUC value is being computed, k is the number of predicted taxonomic levels (in this work, k is set to six as labels from domain to genus are being predicted) and *accuracy* is an array of accuracies of the given classifier in one iteration of the cross-validation.

With the use of the described process, it was possible to obtain five AUC values for every classifier setting – one for every iteration of the cross-validation. Then, for every iteration of the cross-validation separately, the mean of the AUC values of all classifiers was computed. This resulted in five AUC values, each corresponding to one iteration of the cross-validation, which were used to create one box plot.

The AUC metric is used since it is a suitable mean of comparison of accuracies of various classifiers. In the ideal case, accuracies on all levels would be equal to 1 resulting in the AUC value of 1. Therefore, the higher the final AUC value of a classifier is, the more accurate predictions were obtained by the model.

8.1 Used Classifier Settings

During the first phase of the KTC application training, which uses cross-validation to determine the best performing classifier settings, ten types of classifiers altogether in one hundred and three configurations, which have been determined using grid search, are trained and validated in every iteration. The overview of all used classifier types and their settings can be seen in table 8.1.

Table 8.1: Classifier types and their settings used in the classifier competition

Classifier type	**Classifier settings**
SVM	kernel = {linear, rbf, sigmoid}
Nearest centroid	metric = {euclidean, manhattan, chebyshev, minkowski, seuclidean, correlation}
k-NN	n_neighbors = {1, 2, ... , 12}
Decision tree	max_depth = {2, 3, ... , 20}

Random forest	max_depth = {7, 10, 15}
	estimator_count = {10, 15}
	max_features = {auto, sqrt, log2}

MLP	penalty = {0.01, 0.1, 0.15}
	max_iterations = {200, 300, 500}

GaussianNB	-
MultinomialNB	-

NMDKJoint	position_count = {100, 200, ... , 1000}
	average_metric = {mean, median}

NMDKSeparate	position_count = {100, 200, ... , 1000}
	average_metric = {mean, median}

8.2 Description of Used Datasets

A reference database containing sequences with taxonomy annotations is required to train a model as well as to validate the accuracy of taxonomy prediction. For evaluation of the implemented method, three datasets have been used. The first and the largest one is SILVA rRNA database[1]. It contains more than 530,000 samples of entire 16S rRNA of bacteria and archaea together with their taxonomic classifications from domain to genus. The SILVA dataset is so large since the majority of its taxonomy annotations are predicted using computational and manual analyses that are based on trees predicted from multiple alignments [17]. The second used dataset is the BLAST 16S dataset[2] consisting of more than 7,500 16S rRNA sequences of well-known and examined bacteria in FASTA format. This dataset contains only sequences with authoritative names and, therefore, is much smaller tan SILVA. [18] The last dataset used for evaluation of the implemented algorithm is the BLAST V4[2] dataset, which contains region V4 sequences extracted from the 16S rRNA genes in the BLAST 16S dataset. Another dataset has been used for validation of the final application, which is described in detail in section 8.11. It is the ITS dataset[2] of the fungal internal transcribed spacer (ITS) region that contains more than 16,000 fungal specimens. This dataset is used to show the possible reusability of the presented classifier for other databases for which the model was not originally designed.

Every gene sequence is a string composed of four nucleotides – A, C, G, U, and for every sequence, there is its complete taxonomic classification. The entire 16S rRNA sequence consists of approximately 1,500 nucleotides.

8.3 Examination of Weight of Possible Real Sequences in K-mer Spectra Extraction

The first examined aspect of the proposed KTC classifier is the weight of generated sequences during k-mer spectra extraction. As was mentioned in section 6.1, the non-nucleotide characters in 16S rRNA sequences are being replaced by all possible real nucleotide characters. A single k-mer, which contains a non-nucleotide symbol, then leads to the generation of multiple possible real k-mers, whose count of occurrences is incremented by some value. This section inspects the impact of the value which is added to the count of occurrences of all possible real gene sequences after replacing the non-nucleotide characters with the nucleotides they can represent.

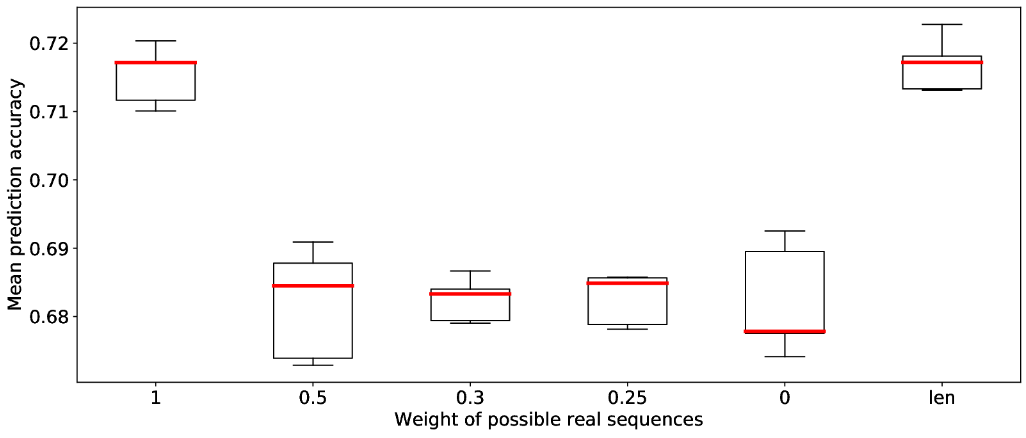
The comparison of various weights of the generated real sequences can be seen in box plots presented in figure 8.1. Six values have been experimented with – 1, 0.5, 0.3, 0.25, $\frac{1}{Number\ of\ generated\ sequences}$, which is labelled as "$len$" in the graphs, and 0, which represents omitting sequences containing non-nucleotide characters.

The graphs in figure 8.1 were created using two datasets – the results obtained with the use of BLAST V4 dataset can be seen in figure 8.1a and the results acquired on SILVA dataset are displayed in figure 8.1b. The presented values were obtained using k-mer size 5 as it is the average of used sizes of k-mer and it proved to be time and memory efficient and offer a quite good accuracy at the same time. It has been experimented also with

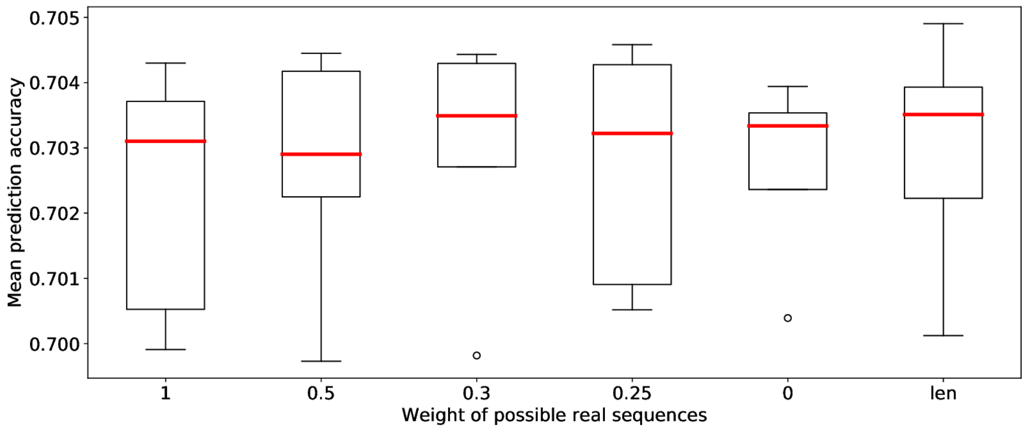
[1] The SILVA database is available at www.arb-silva.de.

[2] The BLAST 16S and V4 datasets and the ITS dataset are available on site https://drive5.com/taxxi/doc/fasta_index.html

the border k-mer sizes, 2 and 8, and the differences among the individual weights were almost the same.



(a) Using BLAST V4 dataset



(b) Using SILVA dataset

Figure 8.1: Comparison of prediction accuracy using various weights of possible real gene sequences

According to the results in figures 8.1a and 8.1b, the best results seem to be the ones obtained using the "*len*" option. The results acquired with the BLAST V4 dataset show it as clearly the best option. The value 1 also gave satisfactory results. All the remaining values performed significantly worse and the least successful value seems to be 0, which is used in some other existing solutions, such as IDTAXA [41].

When using SILVA dataset, the differences among the performances of the given weights are rather small, the differences might not be significant as the individual boxes overlap. This can be caused by the size of the dataset. Here, the values 1 and 0.5 seem like the worst performing options. The best accuracy has been reached when using values 0.3, 0.25 and "*len*".

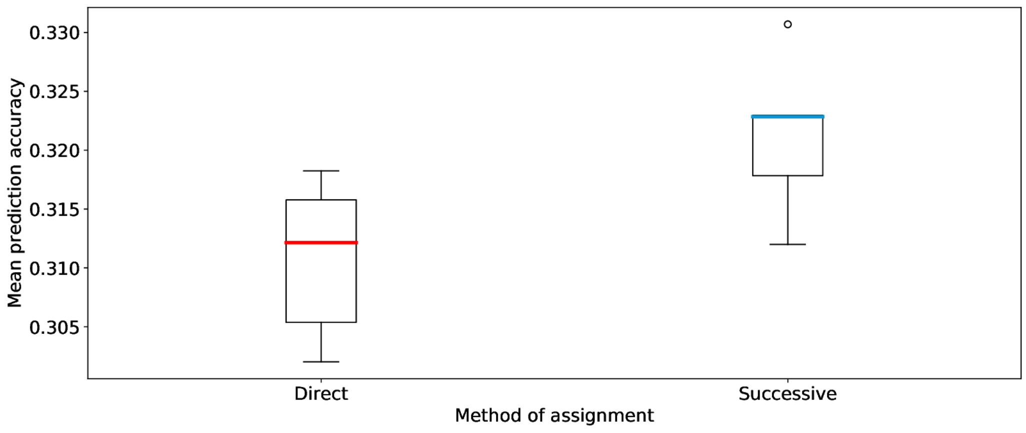
The weight "*len*" reached the best overall performance and therefore is currently used in the KTC application. All other experiments presented in this chapter are executed with this weight of the generated sequences as well.

8.4 Comparison of Tree Structure and Direct Assignment

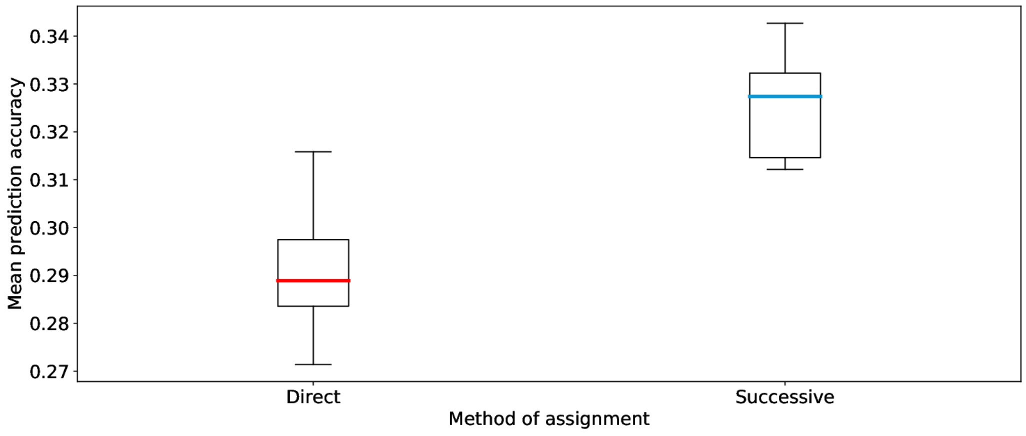
This section aims to prove the benefits of the tree structure of classifiers in comparison to direct assignment on the genus level. The hypothesis, that the successive assignment of taxonomic categories leads to an increase in prediction accuracy thanks to the fact that every classifier distinguishes only among a few categories on the lower taxonomic level and therefore can learn different patterns for every partial classification, is the core of the presented algorithm and thus is very important to prove to be true.

The box plots presented in figure 8.2 compare the accuracy of the described KTC algorithm, which is labelled in the graphs as "*successive*", and direct genus assignment marked in the figure as "*direct*". This approach uses only one classifier distinguishing among all genera present in the dataset.

The comparison was evaluated on two datasets – the results acquired on BLAST V4 dataset can be seen in image 8.2a and the results obtained with the use of SILVA dataset are displayed in figure 8.2b. The displayed results were obtained using k-mer size 5 and weight of generated gene sequences of the type "*len*".



(a) Using BLAST V4 dataset



(b) Using SILVA dataset

Figure 8.2: Comparison of accuracy of tree structured and direct assignment

In both graphs, the prediction accuracy on the genus level is better when using the tree structure of classifiers (by circa 1 % on BLAST V4 dataset and by approximately 4 % better on SILVA dataset). On the two used datasets, the hypothesis of the accuracy increase achieved by utilizing the tree structure of classifiers seems to have been confirmed.

The difference between the two approaches, however, may be smaller than expected. This might be influenced by the fact that during the successive assignment, every unknown sequence is classified six times, by six classifiers, and therefore there are more classifications that can possibly be wrong. For example, if we had one classifier with 90 % accuracy and six subsequent classifiers with an accuracy of 98 %, the overall accuracy of the successive classification is 0.98^6 which equals approximately 0.89, which is 89 %. Thus the one classifier with 90 % accuracy reaches better overall accuracy than the listed successive approach. This effect applies to the tree structure of predictions in the KTC algorithm as well, therefore, even if the accuracy of every single partial classifier can be notably higher than the accuracy of the one classifier present in the direct assignment approach, the overall accuracy of the entire successive classification is not so high.

8.5 Examination of Performance for Various NMDK Classifier Settings

The NMDK classifier proposed and described in section 6.3 contains three variable parameters – metric used to compute the average k-mer spectrum, count of attributes that are extracted from every k-mer spectrum and joint or separate approach. The aim of this section is to inspect all of the mentioned aspects and their impact on prediction accuracy.

8.5.1 Examination of Averaging Metric Impact

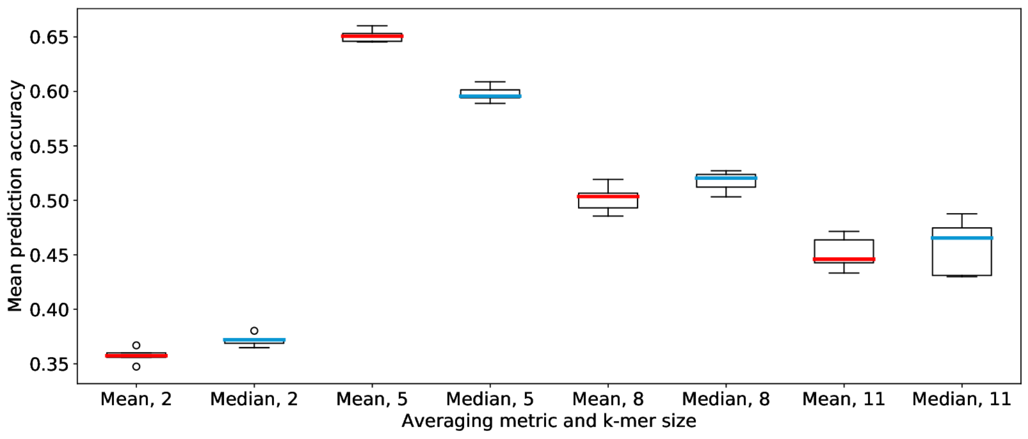
The first examined variable of the NMDK classifier is the averaging metric which is used to compute the average k-mer spectrum of a class. In the application, two averaging metrics have been implemented and experimented with – mean and median. Mean is the best-known and simple way to compute an average and median has the advantage of not being as susceptible to extreme values as mean.

Box plots visualising the comparison of the averaging metrics can be seen in figure 8.3. It has been experimented with k-mer sizes from 2 to 11. According to the results obtained using the individual k-mer sizes, the better performing metric was different for different k-mer sizes. Therefore, the presented graphs show the comparison of the two metrics for various k-mer sizes ranging from 2 to 11.

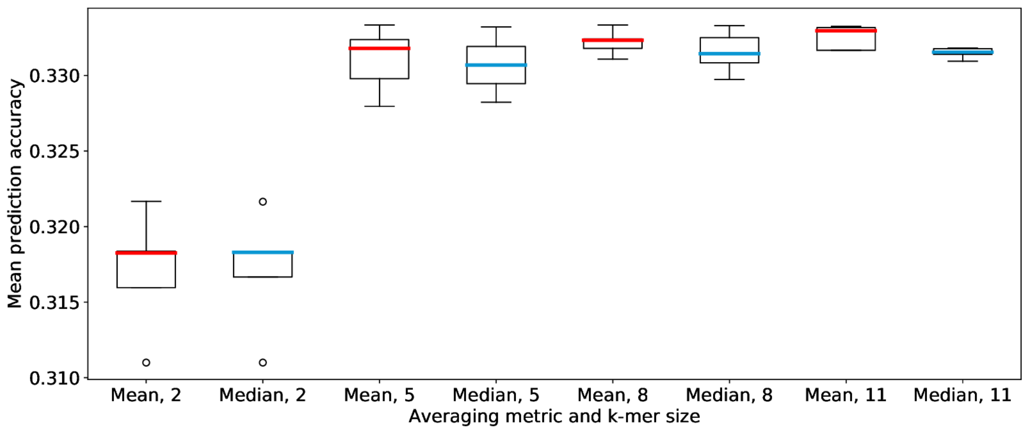
The comparison of the two mentioned averaging metrics has been evaluated on two datasets – the results obtained using BLAST V4 dataset are displayed in figure 8.3a and the results acquired on SILVA dataset can be seen in image 8.3b.

From both the presented graphs can be deduced that neither of the metrics performed generally better than the other one. In many cases, the results differ also for a given k-mer size. For example, when we consider k-mer size 11, median performs better than mean on BLAST V4 dataset, however, significantly worse than mean when using SILVA dataset.

Overall, it seems that the better performing metric for BLAST V4 dataset is median since it achieved higher accuracies for all k-mer sizes except 5. On the other hand, when using SILVA dataset, mean can be considered as the better performing metric as its accuracies are surely higher for k-mer sizes 5, 8 and 11. This leads to the hypothesis that some of the classes in BLAST V4 dataset contain extreme values, while the specimens of a class



(a) Using BLAST V4 dataset



(b) Using SILVA dataset

Figure 8.3: Comparison of prediction accuracy on genus level using mean and median as the averaging metric for various k-mer sizes

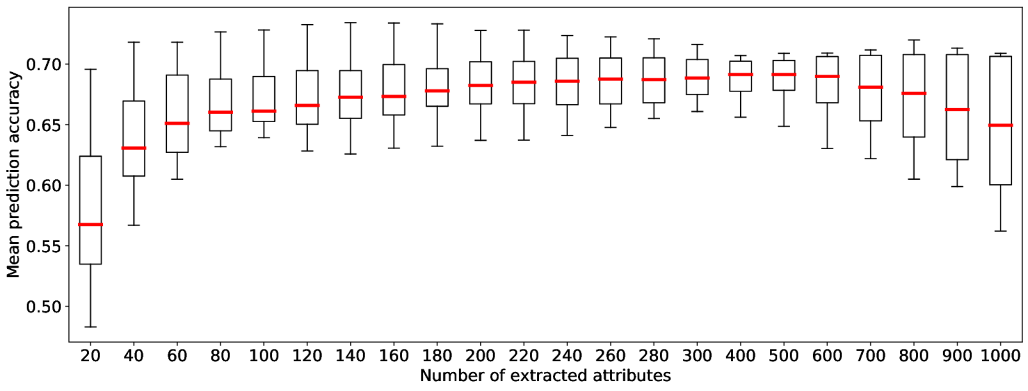
in SILVA dataset are rather close to each other, which can be caused by the smaller size of BLAST dataset.

8.5.2 Examination of Impact of the Size of *N*

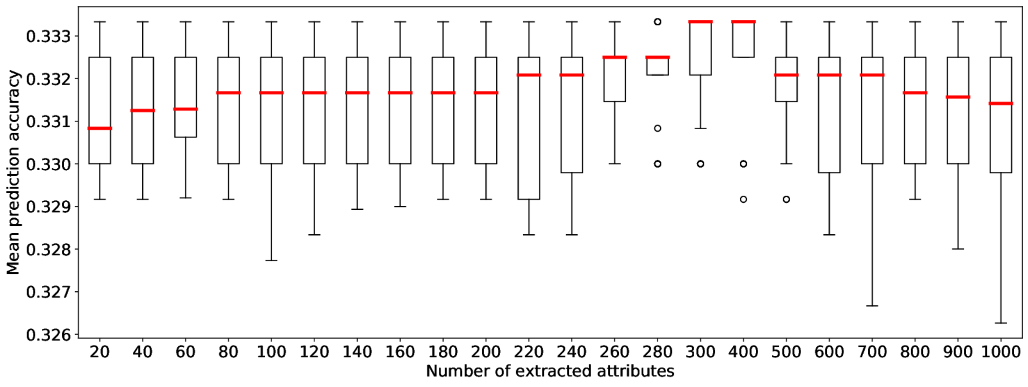
The second evaluated aspect of the NMDK classifier is the impact of the value N, that means the number of the most divergent attributes which are extracted from every average k-mer spectrum. This step aims to dramatically reduce memory requirements and increase the speed of classification as instead of using an input consisting of a large number of attributes (1,024 when using k-mer size 5 and 65,536 for k-mer size 8), only a fraction of the number of attributes is applied. Moreover, these features are determined with an approach which aims to omit attributes that contribute minimally to the differences of the given distinguished subclasses.

Results of the evaluation can be seen in box plots presented in figure 8.4. It has been experimented with the values of N ranging from 10 to 1,000 and with k-mer sizes from 2 to 8. For every k-mer size, the accuracy increased with an increasing number of extracted attributes, however, only until a certain threshold was reached. After reaching the threshold, the accuracy began to drop again. The threshold value was the greater, the larger the k-mer size. The results displayed in this section have been obtained with the use of k-mer size 5.

The experiments have been evaluated using two datasets – the comparison created using BLAST V4 dataset can be seen in figure 8.4a and the results acquired on SILVA dataset are presented in image 8.4b.



(a) Using BLAST V4 dataset



(b) Using SILVA dataset

Figure 8.4: Comparison of various numbers of the extracted most distinguishing features

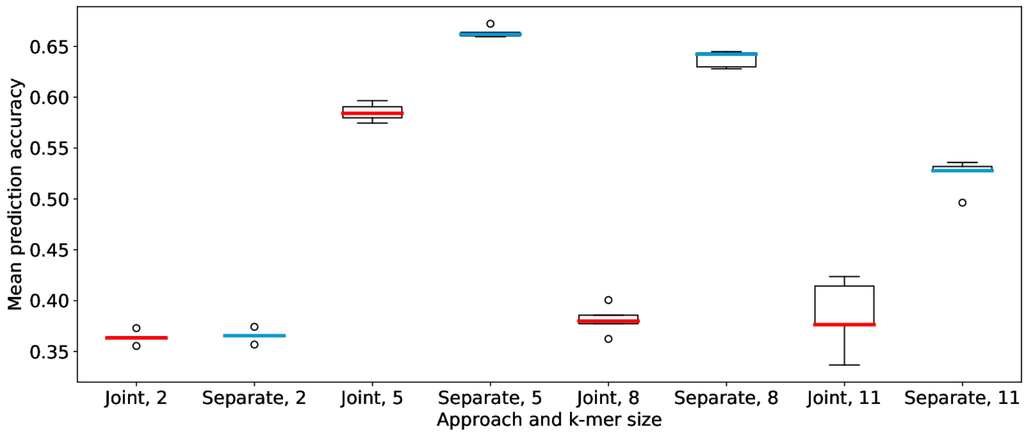
In both graphs, there is an increase in accuracy with an increasing number of extracted features. The threshold, which reached the highest accuracy for k-mer size 5, was around 300 or 400 features on both evaluated datasets, which results in the decrease in input vector size of approximately 70 % or 60 % .

8.5.3 Examination of Impact of Approach

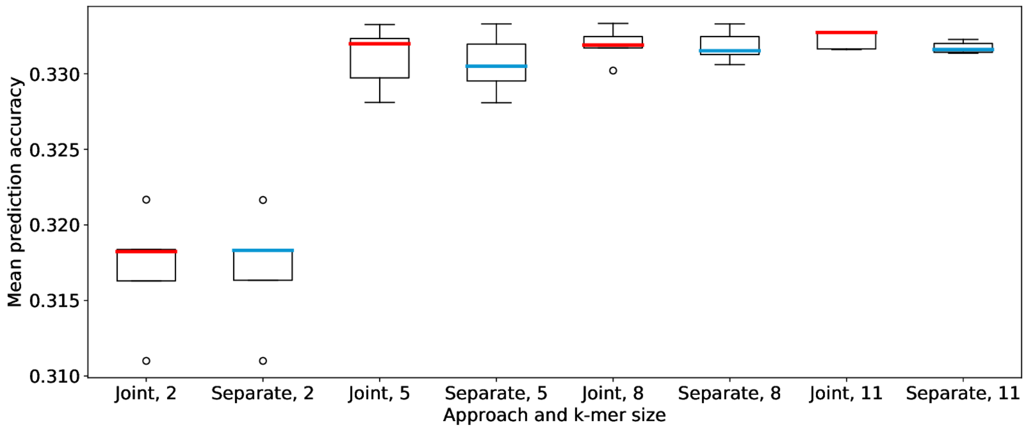
The third inspected element of the NMDK classifier is the approach used during determining the *N most distinguishing* k-mer spectra, that means whether only one set of the *N most distinguishing* positions of k-mer spectra is determined for every partial classifier or if a separate set of positions is chosen for every class on the lower taxonomic level. The first and the second mentioned approaches are called "*joint*" and "*separate*" respectively.

The comparison of the two approaches is visualised in box plots in figure 8.5. It has been experimented with k-mer sizes ranging from 2 to 11. The results obtained using the individual k-mer sizes were not consistent in determining the approach with higher prediction accuracy. Therefore, the presented graphs show the comparison of the two approaches for various k-mer sizes from 2 to 11.

The experiments have been executed on two datasets – the figure 8.5a presents the results obtained using BLAST V4 dataset and in the image 8.5b, there can be seen the results acquired on SILVA dataset.



(a) Using BLAST V4 dataset



(b) Using SILVA dataset

Figure 8.5: Comparison of the joint and separate approaches

Considering the results presented in the graphs in figure 8.5, a generally better performing approach cannot be clearly determined. An interesting outcome is that even for every k-mer size, the better approach differs. For example, when we take a look at k-mer size 5, the separate approach obtains significantly higher accuracy than joint on BLAST V4 dataset. On the contrary, when using SILVA dataset, the joint approach reaches higher accuracy.

The two compared principles reach seemingly equal accuracy when using k-mer 2. This is caused by the fact that the k-mer spectra are composed only of 16 attributes and, therefore, for most values of N remain the k-mer spectra unchanged, in their full length.

Apart from k-mer size 2, the approach with the higher prediction accuracy on BLAST V4 dataset was separate. Moreover, the differences between joint and separate approaches are significant. On the other hand, the differences between the two approaches are rather small when using SILVA dataset and the joint approach managed to reach the higher accuracy for the remaining k-mer sizes.

Considering the results presented in this section, it can be deduced that the approach reaching the higher prediction accuracy depends on both k-mer size and dataset used. Similarly, the results listed in section 8.5.1 point out that the better averaging metric depends on the k-mer size as well as the dataset used. This shows how different these datasets are and also proves the benefits of the two-phase training method of the KTC application described in section 6.2, which chooses the highest accuracy obtaining classifier specifically for the input training dataset.

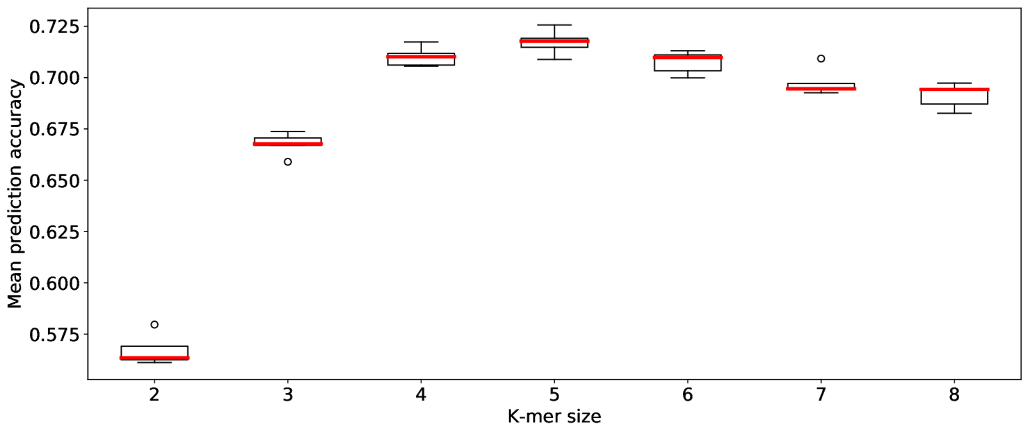
8.6 Examination of Impact of K-mer Size

Another evaluated element of the KTC application is the impact of the k-mer size used when extracting the k-mer spectra from the input gene sequences. The impact of the size of k-mer and its meaning were described in detail in section 6.1, which says that smaller size of k-mer is capable of extracting less information than larger k-mer size, however, the extracted k-mer spectrum is less affected by the mutations in them.

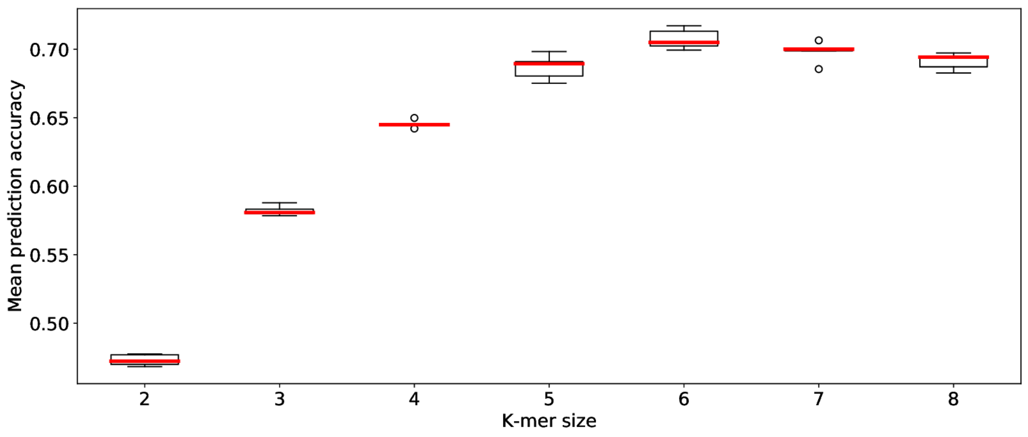
K-mer sizes ranging from 2 to 8 have been experimented with and evaluated. Larger k-mer size leads to a significant increase in both memory and time requirements. The comparison of various k-mer sizes used for k-mer spectra extraction can be seen in box plots presented in figure 8.6. All the k-mer sizes, which have been experimented with, are included in the graphs.

The graphs in image 8.6 were created using two datasets – the results acquired on BLAST V4 dataset can be seen in figure 8.6a and the results obtained with the use of SILVA dataset are displayed in figure 8.6b.

Comparisons evaluated on both datasets show that with increasing k-mer size, the accuracy increases as well, however, only until a certain threshold is reached and then decreases again. This threshold, i.e. the k-mer size reaching the highest accuracy, differs on both datasets – when using BLAST V4 dataset, the best performing k-mer size was 5 and on SILVA dataset was the highest accuracy obtained when using k-mer size 6.



(a) Using BLAST V4 dataset



(b) Using SILVA dataset

Figure 8.6: Comparison of various k-mer sizes

On both datasets, the worst performing k-mer size was 2, which was slightly predictable as the extracted k-mer spectrum contains only 16 attributes. The second worst accuracy was reached by k-mer size 3, also most likely due to the small number of attributes. When using BLAST V4 dataset, the best performing k-mer sizes were 4, 5 and 6, for SILVA dataset worked k-mer sizes 6, 7 and 8 the best.

8.7 Examination of the Impact of Normalization

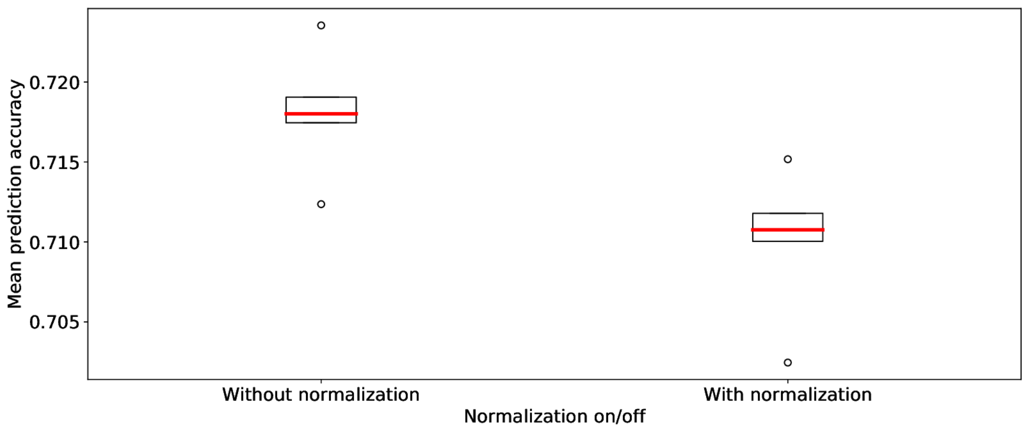
Next inspected aspect of the KTC application is the impact of normalization. The individual elements of a k-mer spectrum, which is extracted from an input gene sequence, can be seen as separate attributes that can generally be on different scales as some k-mers are present in the gene sequences more often than others. This can be problematic for some classifiers which are susceptible to different ranges of various parameters. To overcome the problem of the major influence of variables with bigger ranges of values, normalization is applied to the extracted k-mer spectra to unify the influence of the individual attributes. The focus of this section is on experimenting with normalization and exploring its impact on the prediction accuracy of the KTC application.

The comparison of prediction accuracy with and without the use of normalization can be seen in figure 8.7. It has been experimented with k-mer sizes ranging from 2 to 8 and the differences between the two approaches were very similar for all used k-mer sizes. To create the box plots presented in the image 8.7, the results obtained by using the k-mer size 5 were used.

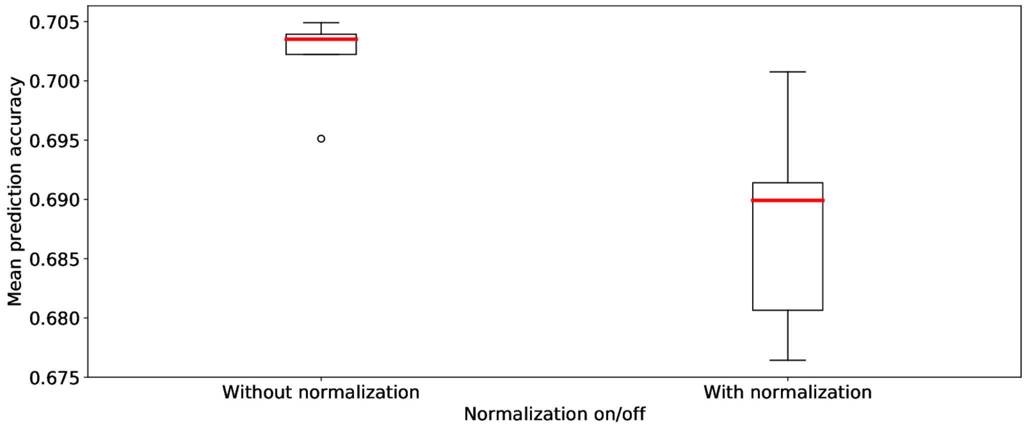
The experiments have been executed on two datasets – the image 8.7a presents the results obtained using BLAST V4 dataset and in the figure 8.7b, there can be seen the results acquired on SILVA dataset.

On both used datasets, the prediction accuracy was significantly higher when using k-mer spectra without normalization. This may be caused by the fact that while normalization increases the prediction accuracy of some classifier types, it notably decreases the accuracy obtained by other types of classifiers.

Therefore, another exploration of the impact of normalization was done, which aims to determine the influence of normalization for every classifier type separately. These ex-



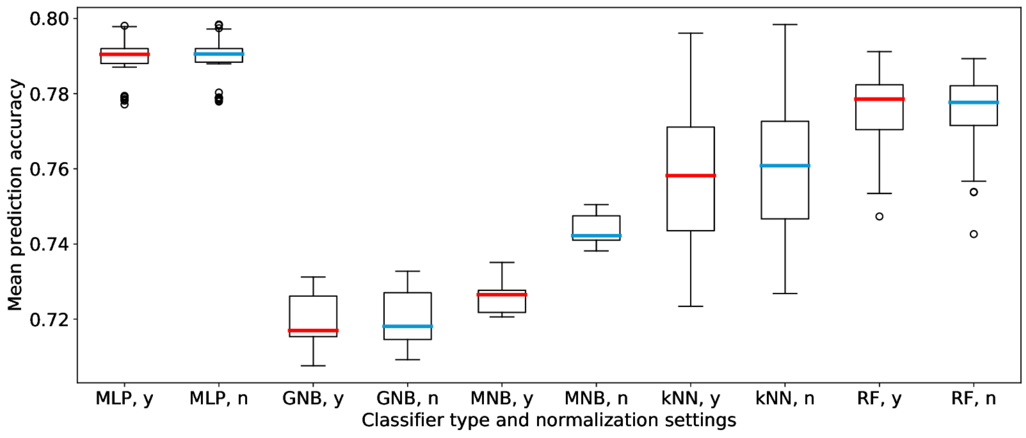
(a) Using BLAST V4 dataset



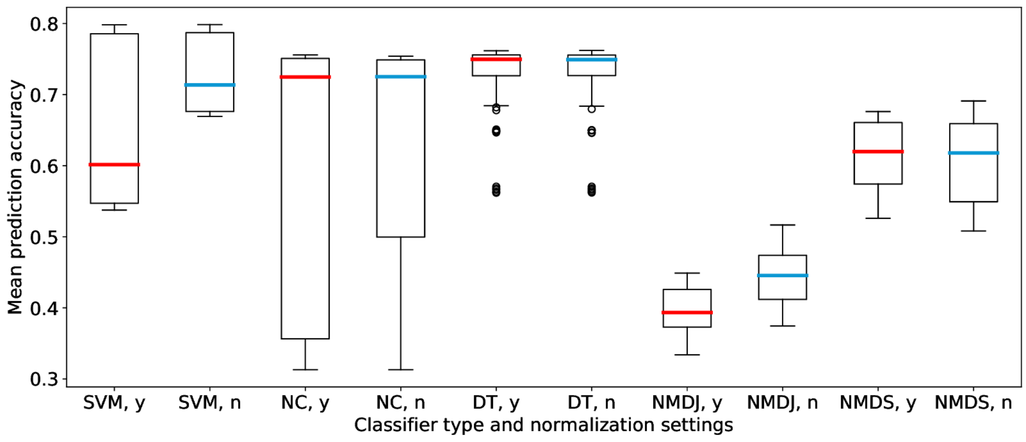
(b) Using SILVA dataset

Figure 8.7: Comparison of prediction accuracy using normalization

periments were executed only on BLAST V4 dataset and also used k-mer size 5. Results of the comparison are presented in box plots in figure 8.8, which contains comparison of accuracy obtained with (labelled as "y" in the graphs) and without (marked with "n") the use of normalization for all used classifier types.



(a) Part 1



(b) Part 2

Figure 8.8: Comparison of prediction accuracy using normalization for all used classifier types separately

Notably better prediction accuracies without normalization were achieved by multinomial naive Bayes, SVM, nearest centroid and NMDK classifier with the joint approach. Only slightly better results without normalization were obtained by Gaussian naive Bayes and k-NN. On the other hand, applying normalization led to better results when using the random forest classifier. Almost the same results with and without normalization were acquired by multilayer perceptron, decision tree and NMDK classifier with the separate approach.

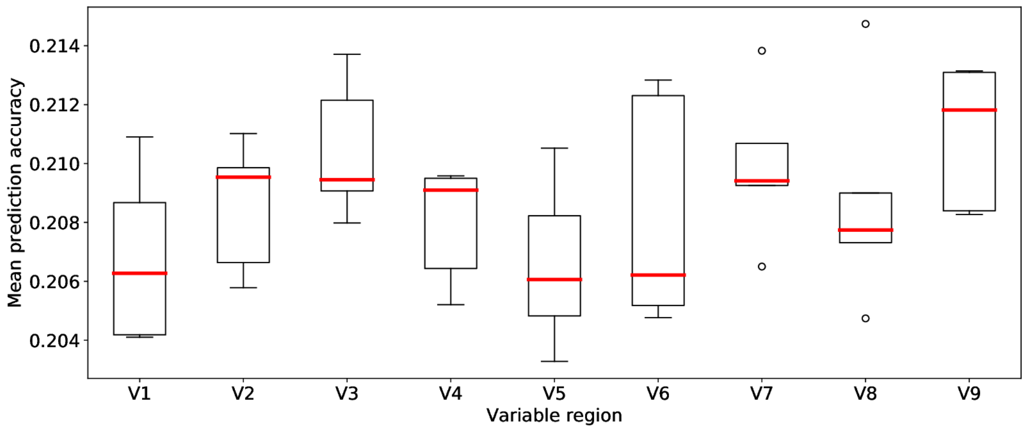
8.8 Examination of the Impact of Variable Regions

As was mentioned in section 2.2, the 16S rRNA gene sequence consists of two types of segments – conserved and variable regions. The conserved regions do not change during evolution and variable regions are the ones suitable for distinguishing among various genera. Many existing solutions described in chapter 5 work on the basis of only one variable region to increase prediction speed as the input gene sequence of a variable region contains significantly fewer characters than entire 16S rRNA. This section aims to experiment with all nine variable regions and try to determine the variable region which achieves the highest accuracy.

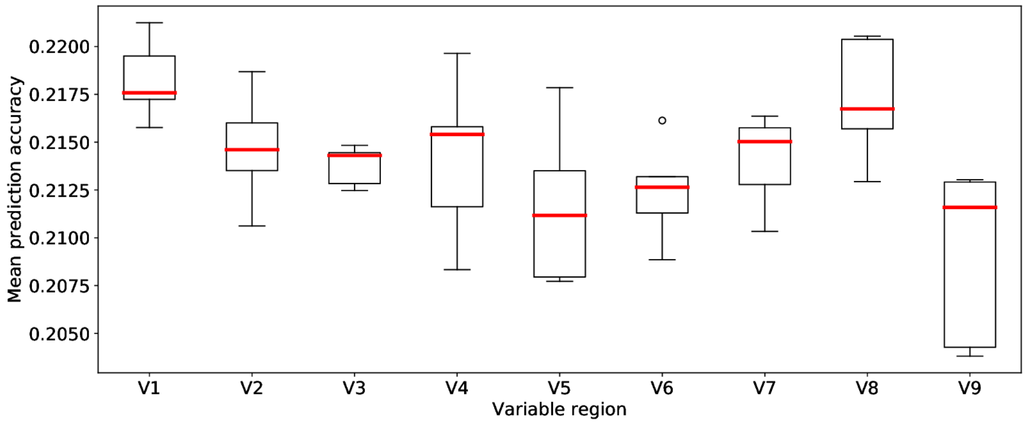
The comparison of accuracies of the individual variable regions is visualised in box plots in figure 8.9. The image shows the comparison of all nine variable regions the 16S rRNA gene sequence consists of. The displayed values were obtained with the use of k-mer size 5.

Two datasets were used to create the presented graphs – BLAST 16S dataset was used to create the box plot in figure 8.9a and the results acquired when using SILVA dataset can be seen in image 8.9b. Both of these datasets contain entire 16S rRNA sequences and the variable regions were extracted from them using the V-Xtractor tool described in section 7.4.2.

The results are extremely different on both datasets. While the best accuracy for BLAST 16S dataset was reached when using region V9, the results from SILVA dataset display this region as the least successful one. Moreover, the best performing region for



(a) Using BLAST 16S dataset



(b) Using SILVA dataset

Figure 8.9: Comparison of prediction accuracy of the individual variable regions

SILVA dataset, V1, was also one of the worst when using BLAST 16 dataset. Overall, good results considering both datasets were achieved by regions V1, V3 and V8.

These discrepancies in the results may be caused by the fact that even though both of these datasets are large, the variable region extraction using the V-Xtractor tool is not always successful and therefore the resulting dataset contains approximately 60 % of the input specimens. Other aspect affecting the results could be the error of the V-Xtractor tool itself. This might have a significant impact since even the best obtained accuracies are bellow 25 %.

Nowadays, the most commonly used hypervariable regions for microbial community profiling are V4, V3–V4 and V4–V5 thanks to the high accuracy that can be reached with their use [61]. On the contrary, in these experiments, neither of regions V4 and V5 offered the highest accuracy.

8.9 Best Performing Classifier Setting

The goal of this section is to determine the classifier setting which reaches the highest prediction accuracy separately for every examined classifier type and to compare the individual classifier types by comparing prediction accuracies of their best performing settings.

The results presented in this section were acquired using k-mer size 5 and the comparison was done for two datasets – BLAST V4 and SILVA. To obtain the values which were used to create the graphs in this section, the 5-fold cross-validation was executed. From the resulting accuracies, five AUC values were computed for each classifier setting (one for every iteration) and then mean of the obtained values was calculated to acquire one average AUC value for every classifier setting. After that, these values were compared to determine the setting with the highest overall prediction score for every classifier type separately. For every type of classifier, the setting which reached the highest prediction accuracy on individual datasets can be seen in table 8.2.

The comparison of the chosen classifier settings is presented in figure 8.10. The graphs for the individual classifier settings were created using the mean of accuracies (of five cross-validation iterations) on every taxonomic level separately.

Table 8.2: Classifier setting reaching the highest accuracy for every classifier type

Classifier type	**Best classifier setting**	
	For BLAST V4 dataset	**For SILVA dataset**
SVM	kernel = linear	kernel = linear
Nearest centroid	metric = correlation	metric = euclidean
k-NN	n_neighbors = 1	n_neighbors = 1
Decision tree	max_depth = 20	max_depth = 17

Random forest	max_depth = 10
	estimator_count = 15
	max_features = sqrt
		max_depth = 10
		estimator_count = 15
		max_features = auto

MLP	penalty = 0.15
	max_iterations = 300
		penalty = 0.15
		max_iterations = 300

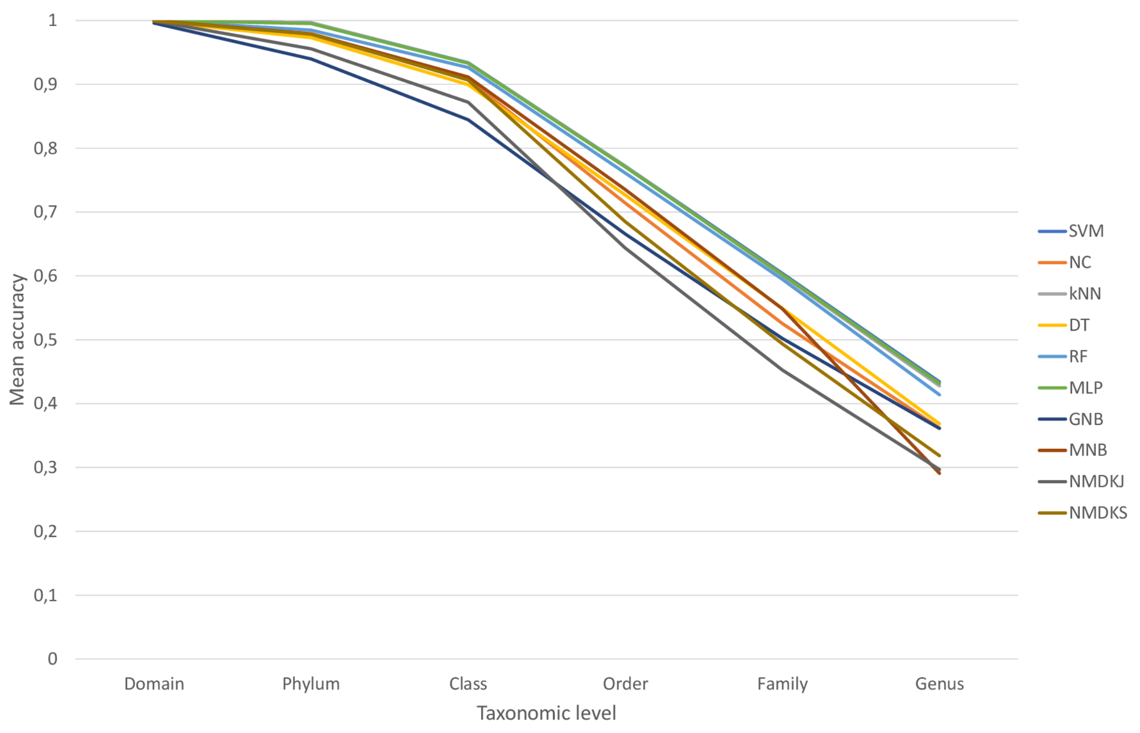
GaussianNB	-	-
MultinomialNB	-	-

NMDKJoint	position_count = 700
	average_metric = median
		position_count = 400
		average_metric = mean

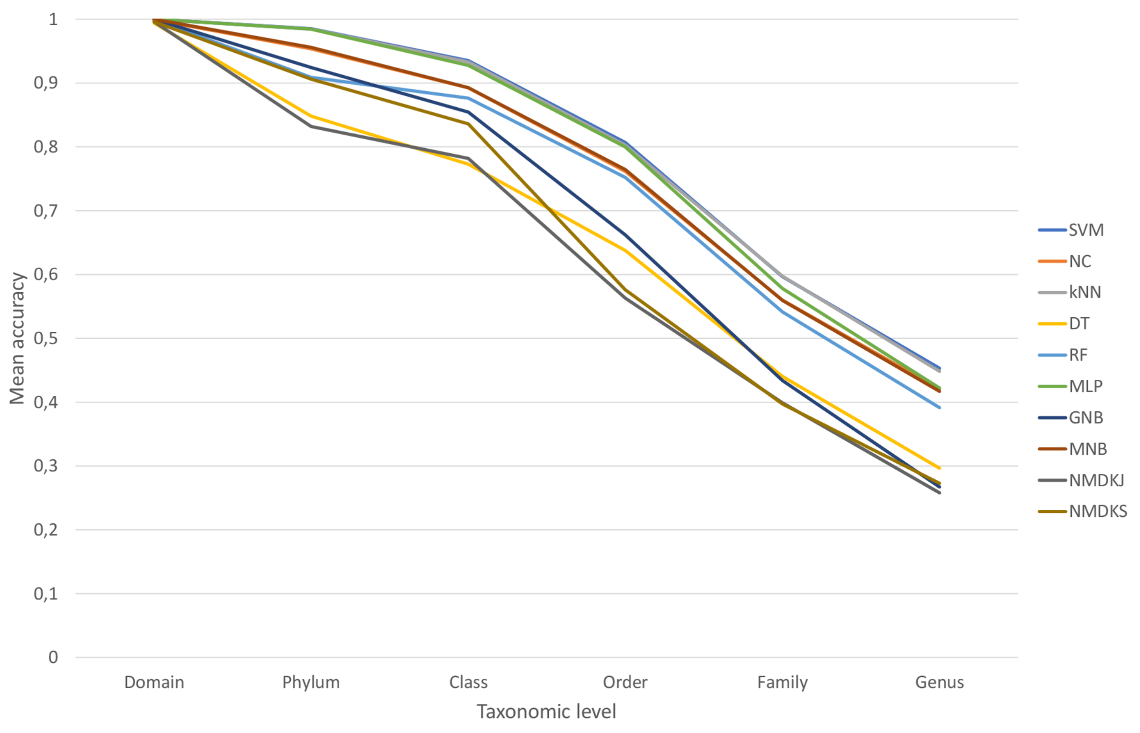
NMDKSeparate	position_count = 120
	average_metric = mean
		position_count = 140
		average_metric = mean

The comparison of the best results every classifier type has reached can be seen in figure 8.10. The results obtained on BLAST V4 dataset can be seen in image 8.10a and the results acquired with the use of SILVA dataset are displayed in figure 8.10b. Both graphs show the dependence of mean prediction accuracy, represented by mean AUC value, of the individual classifier types on predicted taxonomic level.

From graph 8.10a, the Gaussian naive Bayes and NMDK Joint classifiers reached the worst accuracies. Slightly better accuracies were reached by NMDK Separate classifier and multinomial naive Bayes, whose accuracy was good from domain to family, however, significantly decreased on genus level. Decision tree and nearest centroid performed quite good, although worse than the four most successful classifier types – random forest, k-NN,



(a) Using BLAST V4 dataset



(b) Using SILVA dataset

Figure 8.10: Comparison of mean prediction accuracies of the best results obtained by individual classifier types

MLP and SVM. Three of them, k-NN, MLP, and SVM, gave the best results, almost indistinguishable on all levels except genus, where SVM slightly stands out.

Graph 8.10b notionally divides the classifiers into two groups. The worse predicting classifiers are NMDK Joint and NMDK Separate classifiers, Gaussian naive Bayes and decision tree. The group of better-performing classifiers consists of the remaining six types – SVM, nearest centroid, k-NN, random forest, MLP, and multinomial naive Bayes. Multinomial naive Bayes obtained a good accuracy on most levels except for family and genus. The best performing two classifier types on this dataset are SVM and k-NN, which reached almost the same score on most levels, only on order and genus SVM performs better.

For every classifier, there is a notable decrease in accuracy with increasing taxonomic level, as expected. While the domains bacteria and archaea separated millions of years ago (and their 16S rRNA sequences differ significantly), two genera could have been split no longer than a decade ago and, therefore, their 16S rRNA sequences are very similar. Additionally, any misclassification at any taxonomic level is passed further to the lower levels, which means that the classification accuracy at a given taxonomic level can be at most as good as the accuracy obtained at the previous level.

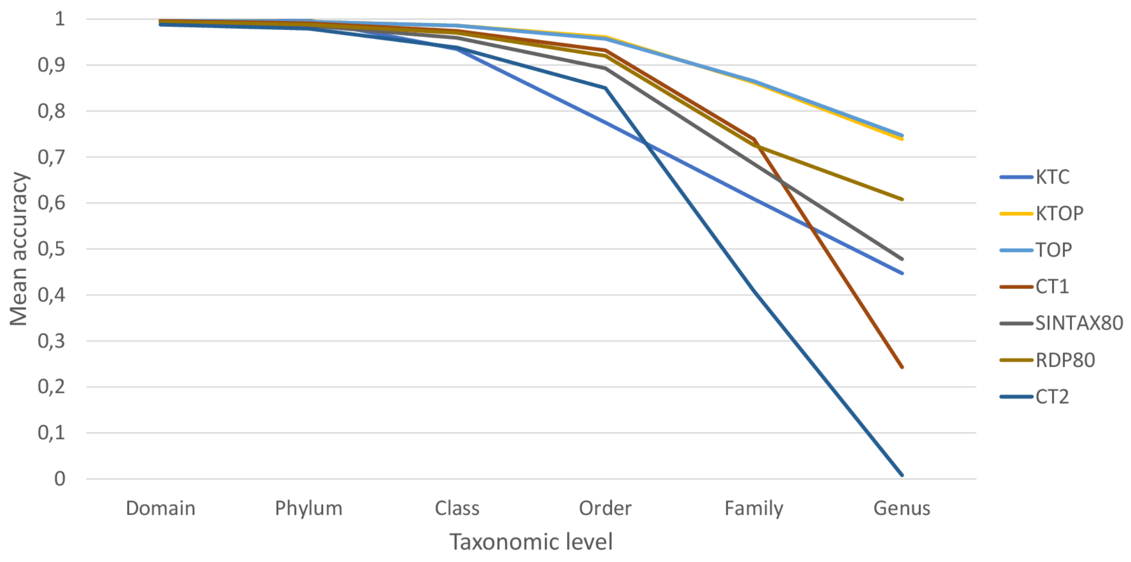
8.10 Comparison with Existing Tools

The goal of this section is to compare the prediction accuracy of the proposed KTC application to other existing solutions. Accuracy of the KTC algorithm was evaluated using the leave-one-out cross-validation. During every run, entire dataset except one specimen was used for training and the left out sample was then used for validation. By utilizing this principle, it was possible to obtain the overall mean accuracy of the presented algorithm on a reference dataset.

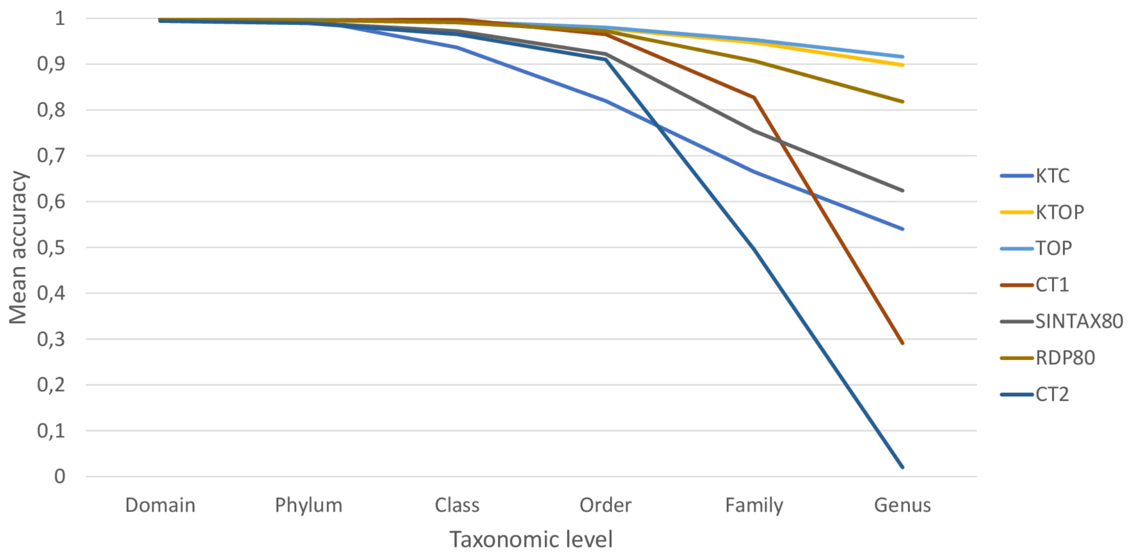
In graph 8.11, there is a comparison of the results obtained by the proposed method to other existing solutions, namely SINTAX, RDP, TOP, KTOP, CT1, and CT2. The first three methods were described in chapter 5, the remaining three algorithms are described in the article *Accuracy of taxonomy prediction for 16S rRNA and fungal ITS sequences* [18]. Both graphs display the dependence of mean prediction accuracy of a classification method on predicted taxonomic level. The presented results of the KTC method for BLAST V4 dataset have been obtained with the use of the MLP classifier with a penalty of 0.1 and 500 iterations and for BLAST 16S dataset with the use of SVM with a linear kernel, which are the classifier settings that have reached the best overall accuracies on the individual datasets. The displayed results of the KTC application were acquired with the use of k-mer size 6 and the accuracies of other solutions were taken from results of the leave-one-out cross-validation executed on the same datasets that are presented in the mentioned article [18]. The article contains the evaluation of the SINTAX and RDP classifiers for three bootstrap thresholds – 80 %, 50 % and 0 %. The results obtained with the threshold of 80 % are presented in this section as it is recommended by authors according to the article [18].

The comparison has been evaluated on two datasets – BLAST V4 dataset was used to create the graph in figure 8.11a and the results acquired when using BLAST 16S dataset can be seen in image 8.11b.

The outcome of both graphs is very similar. The worst accuracy was achieved by the CT2 and CT1 methods. The KTC classifier reached significantly better accuracy than the previous two methods, however, SINTAX, TOP and KTOP classifiers managed to obtain ever higher accuracies. On both datasets, the TOP and KTOP methods proved



(a) Using BLAST V4 dataset



(b) Using BLAST 16S dataset

Figure 8.11: Comparison of results of the proposed KTC application with other existing solutions

to be the most accurate with almost indistinguishable accuracies on all taxonomic levels except genus, which shows that the highest accuracy was obtained by the TOP classifier.

8.11 Validation

The last section of this chapter is dedicated to the validation of the implemented KTC application. Its aim is to estimate the application accuracy in practice, on unseen data. For this purpose, 10 % of an evaluated dataset was stored for validation and excluded from the entire cross-validation, training and evaluation process. First of all, the best performing classifier was determined by cross-validation, trained on the remaining 90 % of the dataset and stored as the final model. After that, the 10 % of the dataset previously left aside was used to evaluate the final expected prediction accuracy of the created model.

The obtained accuracies can be seen in table 8.3. For every dataset, the resulting values for two k-mer sizes, 5 and 6, are presented since according to results presented in section 8.6, these two k-mer sizes were able to reach the highest overall prediction accuracies.

The validation has been executed on three datasets – table 8.3a presents the expected accuracy of the model trained on BLAST 16S dataset and 8.3b shows results obtained with the use of BLAST V4 dataset. Table 8.3c contains accuracies acquired on fungal internal transcribed spacer (ITS) dataset. Its aim is to show the possible reusability of the presented classifier for other databases for which the model was not originally designed.

Table 8.3: Prediction accuracy of the KTC application on validation datasets

(a) BLAST 16S dataset

Taxonomic level	**K-mer size**	
	5	**6**
Domain	1.0	1.0
Phylum	0.99871	0.99871
Class	0.94057	0.94057
Order	0.77003	0.77132
Family	0.59819	0.60207
Genus	0.46899	0.47287

(b) BLAST V4 dataset

Taxonomic level	**K-mer size**	
	5	**6**
Domain	1.0	1.0
Phylum	0.99738	0.99476
Class	0.94102	0.93971
Order	0.77588	0.77195
Family	0.60157	0.60026
Genus	0.45478	0.44954

(c) ITS dataset

Taxonomic level	**K-mer size**	
	5	**6**
Domain	1.0	1.0
Phylum	1.0	1.0
Class	0.98629	0.98567
Order	0.91340	0.91340
Family	0.82243	0.82368
Genus	0.75265	0.75265

The model trained on BLAST 16S dataset managed to reach approximately 47.3 % accuracy on genus prediction when using k-mer size 6. On BLAST V4, the best obtained accuracy when predicting genus was approximately 45.5 % and it was acquired with the use of k-mer size 5. Surprisingly, on the ITS dataset, both used k-mer sizes reached the same accuracy of approximately 75.3 % on genus level, which is significantly higher than on the two previous datasets. This may be caused by the fact that while the BLAST datasets contain a large range of organisms, the ITS dataset consists only of fungi and, therefore, does not contain that many categories on every level.

Chapter 9

Conclusion

At the beginning of this thesis, the basic principles in the field of bioinformatics, metagenomics and bacteria classification were introduced. Moreover, other methods of digital processing of sequence data were presented. Several machine learning algorithms used for classification were described together with their parameters influencing the prediction accuracy and some other existing methods solving the bacteria classification problem were presented. The practical part of this work started with a detailed description of the design of the proposed bacteria classification method. Afterwards, the implementation of the created application was previewed and the implemented modules and their notable methods were listed. The last chapter was focused on application testing and presenting the results of experiments executed with the implemented application. The evaluations were aimed at examination of prediction accuracy when using different k-mer sizes, various types of classifiers and region selection.

The classification method presented within this work was built on the basis of the tree structure of taxonomic categories. The whole classifier consisted of a tree of partial classifiers with topology respecting the taxonomic tree. Classification of an input specimen started in the top classifier distinguishing between bacteria and archaea and the input sequence descended through the tree according to the predicted labels. The partial classifiers were well-known machine learning methods (such as SVM, decision tree, and nearest centroid) and their aim was to classify the given bacteria and assign it a label at the lower taxonomic level.

Furthermore, a new type of classifier was introduced and described called NMDK classifier. Its aim is to offer a solution with good accuracy while achieving a significant decrease in memory requirements and time consumption thanks to dimensionality reduction, which consists of selecting only a given number of the input features and (in order to increase prediction accuracy) omitting attributes that are not of great importance for distinguishing among the given classes.

During the comparison of various k-mer sizes, the highest prediction accuracy was obtained when using k-mer of sizes 5 and 6. Examination of prediction accuracy with the use of individual hypervariable regions showed that the best performing regions differ significantly for the two used datasets. Overall, the best performance was obtained when using regions V1, V3 and V8. During validation, the implemented KTC application reached approximately 47.3 % accuracy on genus prediction on BLAST 16S dataset and 45.5 % accuracy on BLAST V4 dataset. The application was tested also on fungal ITS database, where the acquired accuracy was approximately 75.3 % on genus level.

During the implementation of this application, a few of its constraints, and several ways in which the application could be extended in the future, have emerged.

The first suggestion is to use different classifier types and their settings for different levels. As was shown in section 8.9, the best performing classifier instance differs for various taxonomic levels. Therefore, one approach to increase the classifier accuracy is to thoroughly explore the relations between classifier accuracy and taxonomic level and pick the best classifier instance for every level, or even for every node in the tree of classifiers. A similar principle could be applied to k-mer size as well. Prediction accuracy could be also increased by inspecting the influence of k-mer size on each level, determining the best working k-mer size on every taxonomic level and utilising it.

Time consumption and memory requirements could be further improved by some additional profiling, which would lead to reimplementing some methods in order to decrease the time complexity. Another way to improve the speed of the application would be to provide variables with their static types from the Cython library. That way it would not be necessary to deduct their types while executing the application. According to experiments presented in the article *Speeding Up Python With Cython* by Gigi Sayfan [55], adding static types to code can speed it up approximately 190 times.

Lastly, new types of classifiers or new settings of already available classifiers could be added to the classifier competition, which takes place in the first training phase. The classifier types and settings, which are part of the implementation, were the most accurate ones for the datasets that have been tested. For a different dataset, however, other classifier settings could theoretically reach better accuracy.

Bibliography

[1] Alberts, B.; Johnson, A. D.; Lewis, J.; et al.: *Molecular Biology of the Cell (Sixth Edition)*. W. W. Norton & Company. 2014. ISBN 9780815344322.

[2] Ansari, A.: Smith-Waterman Algorithm. January 2018. [Online; visited 6. 5. 2019].
Retrieved from: https://bioinfoguide.com/index.php/algorithms-and-methods/11-smith-waterman-algorithm

[3] Asiri, S.: Machine Learning Classifiers – Towards Data Science. June 2018. [Online; visited 3. 1. 2019].
Retrieved from: https://towardsdatascience.com/machine-learning-classifiers-a5cc4e1b0623

[4] Barrett, P.: Euclidean Distance. September 2005. [Online; visited 7. 1. 2019].
Retrieved from: https://www.pbarrett.net/techpapers/euclid.pdf

[5] Bishop, C. M.: *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Springer. 2011. ISBN 0387310738.

[6] Breiman, L.: Random Forests. *Machine Learning*. vol. 45, no. 1. October 2001: pp. 5–32. ISSN 1573-0565. doi:10.1023/A:1010933404324.
Retrieved from: https://doi.org/10.1023/A:1010933404324

[7] Bronshtein, A.: A Quick Introduction to K-Nearest Neighbors Algorithm. April 2017. [Online; visited 17. 3. 2019].
Retrieved from: https://medium.com/@adi.bronshtein/a-quick-introduction-to-k-nearest-neighbors-algorithm-62214cea29c7

[8] Brownlee, J.: Naive Bayes for Machine Learning. April 2016. [Online; visited 31. 3. 2019].
Retrieved from:
https://machinelearningmastery.com/naive-bayes-for-machine-learning/

[9] Brownlee, J.: A Gentle Introduction to k-fold Cross-Validation. May 2018. [Online; visited 2. 4. 2019].
Retrieved from:
https://machinelearningmastery.com/k-fold-cross-validation/

[10] Burckhardt, P.: compute-minkowski-distance – npm. May 2015. [Online; visited 11. 1. 2019].
Retrieved from: https://www.npmjs.com/package/compute-minkowski-distance

[11] Cain, A. J.: taxonomy | Definition, Examples, Levels, & Classification | Britannica.com. April 2018. [Online; visited 5. 4. 2019].
Retrieved from: https://www.britannica.com/science/taxonomy

[12] Caporaso, J. G.; Kuczynski, J.; Stombaugh, J.; et al.: QIIME allows analysis of high-throughput community sequencing data. *Nature Methods*. vol. 7, no. 5. April 2010: pp. 335–336. doi:10.1038/nmeth.f.303.
Retrieved from: https://doi.org/10.1038/nmeth.f.303

[13] Chaudhary, N.; Sharma, A. K.; Agarwal, P.; et al.: 16S Classifier: A Tool for Fast and Accurate Taxonomic Classification of 16S rRNA Hypervariable Regions in Metagenomic Datasets. *PLOS ONE*. vol. 10, no. 2. February 2015: page e0116106. doi:10.1371/journal.pone.0116106.
Retrieved from: https://doi.org/10.1371/journal.pone.0116106

[14] Cython developers and contributors: Cython: C-Extensions for Python. February 2019. [Online; visited 5. 4. 2019].
Retrieved from: https://cython.org/

[15] Donges, N.: The Random Forest Algorithm – Towards Data Science. February 2018. [Online; visited 20. 3. 2019].
Retrieved from:
https://towardsdatascience.com/the-random-forest-algorithm-d457d499ffcd

[16] Drakos, G.: Cross-Validation – Towards Data Science. August 2018. [Online; visited 2. 4. 2019].
Retrieved from:
https://towardsdatascience.com/cross-validation-70289113a072

[17] Edgar, R. C.: SINTAX: a simple non-Bayesian taxonomy classifier for 16S and ITS sequences. *bioRxiv*. September 2016. doi:10.1101/074161.
Retrieved from: https://doi.org/10.1101/074161

[18] Edgar, R. C.: Accuracy of taxonomy prediction for 16S rRNA and fungal ITS sequences. *PeerJ*. vol. 6. April 2018: page e4652. ISSN 2167-8359. doi:10.7717/peerj.4652.
Retrieved from: https://doi.org/10.7717/peerj.4652

[19] Edgar, R. C.: USEARCH. March 2019. [Online; visited 21. 3. 2019].
Retrieved from: https://www.drive5.com/usearch/

[20] Gandhi, R.: Support Vector Machine — Introduction to Machine Learning Algorithms. June 2018. [Online; visited 3. 1. 2019].
Retrieved from: https://towardsdatascience.com/support-vector-machine-introduction-to-machine-learning-algorithms-934a444fca47

[21] Genetics Home Reference: What is a genome? – Genetics Home Reference – NIH. January 2019. [Online; visited 16. 1. 2019].
Retrieved from: https://ghr.nlm.nih.gov/primer/hgp/genome

[22] Genetics Home Reference: What is DNA? – Genetics Home Reference – NIH. January 2019. [Online; visited 16. 1. 2019].
Retrieved from: https://ghr.nlm.nih.gov/primer/basics/dna

[23] Google Developers: Machine Learning Glossary | Google Developers. January 2019. [Online; visited 2. 4. 2019].
Retrieved from: https://developers.google.com/machine-learning/glossary

[24] Greenacre, M.: Measures of distance between samples: Euclidean. September 2008. [Online; visited 12. 1. 2019].
Retrieved from: http://www.econ.upf.edu/~michael/stanford/maeb4.pdf

[25] Gupta, P.: Decision Trees in Machine Learning. May 2017. [Online; visited 17. 3. 2019].
Retrieved from: https://towardsdatascience.com/decision-trees-in-machine-learning-641b9c4e8052

[26] Harrison, O.: Machine Learning Basics with the K-Nearest Neighbors Algorithm. September 2018. [Online; visited 17. 3. 2019].
Retrieved from: https://towardsdatascience.com/machine-learning-basics-with-the-k-nearest-neighbors-algorithm-6a6e71d01761

[27] Hartmann, M.; Howes, C. G.; Abarenkov, K.; et al.: V-Xtractor: An open-source, high-throughput software tool to identify and extract hypervariable regions of small subunit (16S/18S) ribosomal RNA gene sequences. *Journal of Microbiological Methods*. vol. 83, no. 2. November 2010: pp. 250–253. doi:10.1016/j.mimet.2010.08.008.
Retrieved from: https://doi.org/10.1016/j.mimet.2010.08.008

[28] Hiergeist, A.; Reischl, U.; Gessner, A.: Multicenter quality assessment of 16S ribosomal DNA-sequencing for microbiome analyses reveals high inter-center variability. *International Journal of Medical Microbiology*. vol. 306, no. 5. August 2016: pp. 334–342. doi:10.1016/j.ijmm.2016.03.005.
Retrieved from: https://doi.org/10.1016/j.ijmm.2016.03.005

[29] Improved Outcomes Software: Manhattan Distance Metric. December 2004. [Online; visited 9. 1. 2019].
Retrieved from: http://www.improvedoutcomes.com/docs/WebSiteDocs/Clustering/Clustering_Parameters/Manhattan_Distance_Metric.htm

[30] Jones, N. C.: *An Introduction to Bioinformatics Algorithms (Computational Molecular Biology)*. The MIT Press. August 2004. ISBN 0262101068.

[31] Koehrsen, W.: Random Forest Simple Explanation. December 2017. [Online; visited 31. 3. 2019].
Retrieved from: https://medium.com/@williamkoehrsen/random-forest-simple-explanation-377895a60d2d

[32] Lafontaine, D. L.; Tollervey, D.: The function and synthesis of ribosomes. *Nature Reviews Molecular Cell Biology*. vol. 2, no. 7. July 2001: pp. 514–520. doi:10.1038/35080045.
Retrieved from: https://doi.org/10.1038/35080045

[33] Lane, D. M.: Values of the Pearson Correlation. August 2013. [Online; visited 13. 1. 2019].

Retrieved from:
http://onlinestatbook.com/2/describing_bivariate_data/pearson.html

[34] Lee, B.: K-mer – PLoSWiki. August 2018. [Online; visited 3. 1. 2019].
Retrieved from: http://compbiolwiki.plos.org/wiki/K-mer

[35] Lewis, C.: Microsoft PowerPoint – Lowest Common Ancestor (LCA) techniques.ppt – lowest_common_ancestor.pdf. September 2004. [Online; visited 19. 3. 2019].
Retrieved from:
http://homepage.usask.ca/~ctl271/810/lowest_common_ancestor.pdf

[36] Li, L.: Ribosomal RNA (rRNA) – Definition and Functions | Biology Dictionary. April 2017. [Online; visited 16. 1. 2019].
Retrieved from: https://biologydictionary.net/ribosomal-rna/

[37] Liland, K. H.; Vinje, H.; Snipen, L.: microclass: an R-package for 16S taxonomy classification. *BMC Bioinformatics*. vol. 18, no. 1. March 2017. doi:10.1186/s12859-017-1583-2.
Retrieved from: https://doi.org/10.1186/s12859-017-1583-2

[38] Mandal, A.: What is RNA? August 2018. [Online; visited 16. 1. 2019].
Retrieved from:
https://www.news-medical.net/life-sciences/What-is-RNA.aspx

[39] Marco, D.: *Metagenomics: Theory, Methods and Applications*. Caister Academic Press. 2010. ISBN 978-1-904455-54-7.

[40] Morgan, X.; Huttenhower, C.: Chapter 12: Human Microbiome Analysis. *PLoS computational biology*. vol. 8. 12 2012: page e1002808. doi:10.1371/journal.pcbi.1002808.

[41] Murali, A.; Bhargava, A.; Wright., E. S.: IDTAXA: a novel approach for accurate taxonomic classification of microbiome sequences. *Microbiome*. vol. 6, no. 1. August 2018: page 140. ISSN 2049-2618. doi:10.1186/s40168-018-0521-5.
Retrieved from: https://doi.org/10.1186/s40168-018-0521-5

[42] National Center for Biotechnology Information: BLAST: Basic Local Alignment Search Tool. March 2019. [Online; visited 21. 3. 2019].
Retrieved from: https://blast.ncbi.nlm.nih.gov/Blast.cgi

[43] Needleman, S. B.; Wunsch, C. D.: A general method applicable to the search for similarities in the amino acid sequence of two proteins. *Journal of Molecular Biology*. vol. 48, no. 3. March 1970: pp. 443–453. doi:10.1016/0022-2836(70)90057-4.
Retrieved from: https://doi.org/10.1016/0022-2836(70)90057-4

[44] Norena, S.: Python Model Tuning Methods Using Cross Validation and Grid Search. May 2018. [Online; visited 3. 1. 2019].
Retrieved from: https://medium.com/@sebastiannorena/some-model-tuning-methods-bfef3e6544f0

[45] NumPy developers: NumPy – NumPy. October 2018. [Online; visited 6. 3. 2019].
Retrieved from: http://www.numpy.org

[46] Pedregosa, F.; Varoquaux, G.; Gramfort, A.; et al.: scikit-learn: machine learning in Python – scikit-learn 0.20.3 documentation. March 2019. [Online; visited 6. 3. 2019].
Retrieved from: https://scikit-learn.org

[47] Press, C. U.: Rocchio classification. April 2009. [Online; visited 7. 1. 2019].
Retrieved from: https://nlp.stanford.edu/IR-book/html/htmledition/rocchio-classification-1.html

[48] QIIME development team: QIIME. January 2018. [Online; visited 19. 3. 2019].
Retrieved from: http://qiime.org

[49] Ray, S.: Understanding Support Vector Machine algorithm from examples (along with code). September 2017. [Online; visited 3. 1. 2019].
Retrieved from: https://www.analyticsvidhya.com/blog/2017/09/understaing-support-vector-machine-example-code/

[50] Rettner, R.: DNA: Definition, Structure & Discovery. December 2017. [Online; visited 5. 4. 2019].
Retrieved from: https://www.livescience.com/37247-dna.html

[51] Robinson, S.: Introduction to Neural Networks with Scikit-Learn. January 2018. [Online; visited 20. 3. 2019].
Retrieved from: https://stackabuse.com/introduction-to-neural-networks-with-scikit-learn

[52] Rosaen, K.: 2016-06-20 | scikit-learn Pipeline gotchas, k-fold cross-validation, hyperparameter tuning and improving my score on Kaggle's Forest Cover Type Competition | ML Learning Log. June 2016. [Online; visited 3. 1. 2019].
Retrieved from: http://karlrosaen.com/ml/learning-log/2016-06-20

[53] Ruegg, C.; Cuda, M.; Gael, J. V.: Distance Metrics. May 2017. [Online; visited 13. 1. 2019].
Retrieved from: https://numerics.mathdotnet.com/distance.html

[54] Sanjeevi, M.: Chapter 4: Decision Trees Algorithms. October 2017. [Online; visited 17. 3. 2019].
Retrieved from: https://medium.com/deep-math-machine-learning-ai/chapter-4-decision-trees-algorithms-b93975f7a1f1

[55] Sayfan, G.: Speeding Up Python With Cython. October 2017. [Online; visited 8. 4. 2019].
Retrieved from: https://code.tutsplus.com/tutorials/speeding-python-with-cython--cms-29557

[56] Schulz, J.: Minkowski distance. May 2008. [Online; visited 11. 1. 2019].
Retrieved from: http://www.code10.info/index.php?option=com_content&view=article&id=61:article

[57] scikit-learn developers: 1.17. Neural network models (supervised) – scikit-learn 0.20.3 documentation. 2007 – 2018. [Online; visited 20. 3. 2019].
Retrieved from:
https://scikit-learn.org/stable/modules/neural_networks_supervised.html

[58] scikit-learn developers: 1.6. Nearest Neighbors — scikit-learn 0.20.2 documentation. 2007 – 2018. [Online; visited 7. 1. 2019].
Retrieved from: https://scikit-learn.org/stable/modules/neighbors.html#nearest-centroid-classifier

[59] scikit-learn developers: 4.8. Pairwise metrics, Affinities and Kernels — scikit-learn 0.20.2 documentation. 2007 – 2018. [Online; visited 3. 1. 2019].
Retrieved from: https://scikit-learn.org/stable/modules/metrics.html

[60] Shetty, B.: Curse of Dimensionality – Towards Data Science. January 2019. [Online; visited 5. 4. 2019].
Retrieved from:
https://towardsdatascience.com/curse-of-dimensionality-2092410f3d27

[61] Singer, E.; Bushnell, B.; Coleman-Derr, D.; et al.: High-resolution phylogenetic microbial community profiling. *The ISME journal*. vol. 10, no. 8. 02 2016: pp. 2020–2032. doi:10.1038/ismej.2015.249.
Retrieved from: https://doi.org/10.1038/ismej.2015.249

[62] Smith, Y.: What is Metagenomics? August 2018. [Online; visited 16. 1. 2019].
Retrieved from:
https://www.news-medical.net/life-sciences/What-is-Metagenomics.aspx

[63] Stothard, P.: IUPAC Codes. https://www.bioinformatics.org/sms/iupac.html. May 2000. [Online; visited 14. 3. 2019].

[64] Swafford, A. L.: Carolus Linnaeus: Classification, Taxonomy & Contributions to Biology – Video & Lesson Transcript | Study.com. April 2018. [Online; visited 5. 4. 2019].
Retrieved from: https://study.com/academy/lesson/carolus-linnaeus-classification-taxonomy-contributions-to-biology.html

[65] Teknomo, K.: Chebyshev Distance. 2017. [Online; visited 9. 1. 2019].
Retrieved from: https://people.revoledu.com/kardi/tutorial/Similarity/ChebyshevDistance.html

[66] vlab.amrita.edu: Global alignment of two sequences – Needleman-Wunsch Algorithm (Theory) : Bioinformatics Virtual Lab II : Biotechnology and Biomedical Engineering : Amrita Vishwa Vidyapeetham Virtual Lab. 2012. [Online; visited 6. 5. 2019].
Retrieved from: https://vlab.amrita.edu/?sub=3&brch=274&sim=1431&cnt=1

[67] vlab.amrita.edu: Smith-Waterman Algorithm – Local Alignment of Sequences (Theory) : Bioinformatics Virtual Lab II : Biotechnology and Biomedical Engineering : Amrita Vishwa Vidyapeetham Virtual Lab. 2012. [Online; visited 6. 5. 2019].
Retrieved from: https://vlab.amrita.edu/?sub=3&brch=274&sim=1433&cnt=1

[68] Wang, Q.; Garrity, G. M.; Tiedje, J. M.; et al.: Naive Bayesian Classifier for Rapid Assignment of rRNA Sequences into the New Bacterial Taxonomy. *Applied and Environmental Microbiology*. vol. 73, no. 16. June 2007: pp. 5261–5267.

doi:10.1128/aem.00062-07.
Retrieved from: https://doi.org/10.1128/aem.00062-07

[69] yourgenome: What is DNA sequencing? | Stories | yourgenome.org. June 2016. [Online; visited 16. 1. 2019].
Retrieved from: https://www.yourgenome.org/stories/what-is-dna-sequencing

Appendix A

Contents of the Attached Storage Media

On the attached storage media, the following files can be found:

- this thesis in digital form,
- source codes of this thesis in LaTeX,
- source codes of the KTC application together with additional files necessary for Cython compilation,
- README file with detailed description of the application usage and some example commands for testing the application,
- script `install.sh` for installing all required libraries and dependencies,
- subfolder Datasets containing the four used datasets:
  - ten_16s.100.txt – the BLAST 16S dataset,
  - ten_16s_v4.100.txt – the BLAST V4 dataset,
  - silva.json – the SILVA dataset,
  - rdp_its.100.txt – the fungal ITS dataset,
- subfolder Models containing six pre-trained models:
  - model_16s_ks_5.sav and model_16s_ks_6.sav trained on BLAST 16S dataset,
  - model_v4_ks_5.sav and model_v4_ks_6.sav trained on BLAST V4 dataset,
  - model_its_ks_5.sav and model_its_ks_6.sav trained on ITS dataset,
- file `evaluationInput` containing sample 16S sequences for classification,
- subfolders KmerSpectra and Labels containing already preprocessed k-mer spectra and their corresponding labels.

Appendix B

Description of Implemented Modules and Classes

`Makefile` The first important file for Cython compilation is `Makefile`. The main aim of this file is to allow the user to compile the code regardless of their Cython skills. Typing `make` to the command line triggers the copy of files with ".py" extension to files with ".pyx" extension which is required by Cython. The process of file duplication is implemented in the `copy_files.py` script. After that, the Cython compilation is executed for which the details and parameters are implemented in the `compile.py` module. The compilation creates multiple additional ".c" and ".so" files that contain the compiled code. By typing `make clean`, all of these extra files are deleted.

`main.py` Another notable auxiliary file is `main.py` which is a very simple file added as an entry point file for the application. This file is required as Cython does not generate executable binaries by default. The `ktc` module is imported in this file and an instance of the `KTCBacteriaClassifier` is launched from it.

`ktc.py` The `ktc.py` module contains implementation of the main `KTCBacteriaClassifier` class which has three significant methods corresponding to the tree phases: preprocessing, training and evaluation. The `Preprocess` method creates subfolders for storing persistent files and uses the `DatasetProcessor` class to load and process the input dataset. After that, it executes k-mer extraction from the loaded data and storing the k-mers to persistent files. This is done iteratively for every k-mer size from 2 to 8. Within the `Train` method, the k-fold cross-validation takes place. In every iteration, the training and validation data are loaded and all classifier settings, which are involved in the competition, are trained and validated using the `Train` and `Validate` methods from `tree_classifier.py` module. After the entire cross-validation process, the `StoreBestPerformingClassifier` method is called in which the accuracies of all classifiers in all iterations are combined to determine the best performing classifier settings which is then trained on all available data and persistently stored. The `Evaluate` method starts with loading the unknown sequences and using the `kmer_spectra_extractor.py` module to extract k-mers from them. Then, the classification model is loaded using the `LoadModel` method of `TreeClassifier` class and used to classify the input specimen in the `Evaluate` method of the same class.

dataset_processor.py This module contains the `DatasetProcessor` class, the goal of which is to load the content of an input database in JSON or FASTA format and process it to obtain two separate arrays, one with the loaded sequences and the second one with extracted labels of the specimens. This format is suitable for training since the machine learning algorithms used accept the sequences and their corresponding labels as two separate parameters.

kmer_spectra_extractor.py Within this module, the `KmerSpectraExtractor` class is impelemented which takes care of k-mer extraction from the input 16S rRNA sequences. Specifically, the `GetKmerSpectra` method accepts the sequences as input and returns an array of k-mers extracted from them. Another notable method is called `StoreKmerSpectra` and its purpose is to store the created k-mers to a persistent file using *joblib* from the *scikit-learn* library. Previously, the k-mer files and models were stored using *cPickle*, however, problems with too large data were soon encountered.

tree_classifier.py The `tree_classifier.py` module implements the core functionality of tree classification. The first important method is `BuildTree` in which the tree of partial classifiers is generated on the basis of labels of the given training data. The tree is created using the depth first search algorithm. Another notable method is `Train` in which the classifier tree is traversed and all partial classifiers are trained using the `fit` method. In the method called `Validate`, labels of all validation data are predicted and their accuracy is computed. The classification of an input sample is done in `PredictLabel` method, which descends through the tree according to assigned labels. Last noteworthy method is named `Evaluate` and takes care of unknown data classification. It also uses the `PredictLabel` method to assign labels to the data and then outputs the obtained classifications.

nmd_classifier.py This module implements the proposed classifier type – the NMDK classifier. There are three classes implemented – `BaseNMDKClassifier`, which is the base class implementing the core functionality that is the same for both types of this classifier, and `NMDKClassifierJoint` and `NMDKClassifierSeparate` classes inheriting from the base class.

classifier_instances.py The aim of the `classifier_instances.py` module is to implement the classes representing every type of classifier. Every class then contains the attributes which are specific for the given classifier type. For example, the `kNNClassifier` class has two attributes – the classifier type and the neighbour count.

common.py The module `common.py` contains methods and attributes which are used by more than one of the mentioned modules and do not logically belong to any other module. There are statically defined the classifier settings that are evaluated within the classifier competition. Regarding methods, there is the `TransposeMatrix` method implementing matrix transposition, `PrintError` method for printing the given error message to standard error output, and `CreateFilename` method, which returns a filename created on the basis of passed argument values.