
# Comparing Artificial Neural Networks and Microbial Genome Representation Methods for Taxonomic Classification

Miriam Tamara Grødeland Aarag

Informatics: Programming and System Architecture

60 credits

Department of Informatics

Faculty of Mathematics and Natural Sciences

Autumn 2021

# Abstract

Taxonomic classification of microorganisms is useful as microorganisms play an intricate role in health. However, the microorganisms are difficult to classify both because of their diversity and unstable gene pools. Researchers are attempting to solve this issue using machine learning to handle the ever-growing amount of genomic data. While several tools are and have been developed for this purpose, there is little public research directly comparing the underlying methods used by these tools. The research discussing how to compare different ways of representing a genome numerically for example, is limited. Most of the research and tools are also developed for and tested on marker gene analysis, while other types of analysis, such as metagenomic and metatranscriptomic are less common.

This master thesis explores taxonomic classification on whole genome data by performing direct comparisons on different ways of representing a genome through k-mers and testing on different types of neural networks. A training, validation, and test set was made from the GTDB database which covers a wide range of bacterial and archaea whole genomes. These genomes were transformed into k-mer representation vectors using the following methods: MinHash sketching, frequencies of random k-mers, presence of random k-mers, and discriminative k-mers. Each of these methods were tested on a set of three different artificial neural networks, standard neural network, multilayer perceptron network, and convolutional neural network. All models were measured for accuracy and precision on a test set to determine the combination of representation method and model that would be the most suitable for taxonomic classification of microorganisms.

The findings indicated that a MinHash representation method on a multilayer perceptron network was the most promising. The findings also indicated k-mer counting will give better performance than k-mer presence, when the representation vectors are of equal length. For discriminative k-mers, the results were negative, but inconclusive as alterations in the implementation could potentially give a very different result. Overall, more research is necessary to form comprehensive guidelines for future classification tools.

# Acknowledgements

I would like to thank my supervisors, Torbjørn Rognes at the Department of Informatics at the University of Oslo, and Karin Lagesen at the Norwegian Veterinary Institute, for their guidance and useful feedback throughout the project. I would also like to thank UNINETT Sigma2 for the provided computational resources (project NN9383K).

# Contents

# List of Figures

# List of Tables

# Chapter 1: Introduction

Microorganisms are everywhere. Their presence, or lack thereof, play a role in the health of our bodies and ecosystems. For example, the composition of the microbiome in our gut can increase the risk of health problems such as cardiovascular disease, meaning the study of microbiomes may be an important venue for future health interventions (Shreiner, Kao and Young, 2015). Due to the great diversity of microorganisms however, differentiating between different categories can be challenging, especially at lower levels in the taxonomic tree. One of the ways in which this can be achieved, is through DNA analysis. This means classifying the microorganism by analysing its DNA and measuring the genetic similarity with others of its kind. However, DNA analysis is expensive and time consuming. Even with domain-expert knowledge, it is also not always obvious which parts of the DNA are useful for classification. Machine learning is a rapidly developing field of study that is pushing the limits of what can be accomplished with a computer. Properly trained machine learning models can discover complex relationships in data that humans are likely to miss. These models can especially be useful when the data size is large, and it is difficult to distinguish what parts will be useful for what one is trying to achieve. As such, machine learning is a suitable tool for classification of microbial data. Significant research has been conducted on using artificial neural networks for taxonomic classification. Artificial neural networks are a powerful machine learning technique that can discover complex relations between data samples. However, most of this research has been conducted on marker genes, such as 16S rRNA. A marker gene is a specific gene that is used for taxonomic classification due to being universally present in all organisms as well as being highly varied between species. The research on using the whole genome, an organism's entire DNA, to train models is far more limited. Another gap in the research is the lack of a known best practice for representing DNA when performing machine learning. Machine learning models typically do not work as well on sequence data but require the data to be re-formatted or transformed in some way. For example, a common restriction is that the model can only operate on numerals, meaning textual data must be transformed to a numerical representation. Another aspect of this is dimensionality reduction. In high dimensional data, reducing the dimensionality by extracting a subset of features from each data sample can improve performance. However, selecting features on high dimensional data is not a straightforward task. There are many different methods of dimensionality reduction for DNA input data that have been used in research. However, there is little research that directly compares methods on the same dataset and models.

In this thesis, we will attempt to classify whole genome microbiome DNA using artificial neural networks. This will include using a dataset of microbial genomes to train a neural network to taxonomically classify microorganisms. We will implement different ways of representing the dataset and use it to train different types of artificial neural networks. Finally, we will compare how the neural networks perform on each data representation, in terms of accuracy and precision. The goal is to make some observations on how data representation method and type of neural network affect classification accuracy on a large DNA dataset.

# Chapter 2: Theory

## 2. 1 Microbiomes and Taxonomic Classification

### 2.1.1 Microbiomes

The term microbiome refers to microorganisms, their genome, and environment. For example, the combined microorganisms residing in a human body. A microorganism, or microbe, is an organism of microscopic size that lives as a single-celled organism, or as a colony of cells. In the Three-domain system the domains Archaea and Bacteria are both made up entirely of microorganisms (see section 2.1.2). Microbiomes play a vital role in their environment and being able to identify and classify them can be very beneficial. For example, the microbiome in a human body can have a big impact on an individual's health and susceptibility to disease.

### 2.1.2 Taxonomy

The use of genomic sequencing led to a first division of life into either the cellular or the viral empires (Koonin, 2010). Modern taxonomy further divides the cellular empire into Archaea, Bacteria, and Eukaryota. Archaea and Bacteria are all single-celled organisms, while Eukaryota includes all multicellular organisms, as well as some single-celled organisms. Organisms belonging to the Archaea and Bacteria domains are often grouped together in the term prokaryotes as they share many features. An important difference between prokaryotes and eukaryotes is the ways in which they pass on their genes. While eukaryotes may reproduce sexually through meiosis, prokaryotes reproduce asexually through mitosis. However, prokaryotes also have the capability of horizontal gene transfer, where incoming genes may either replace an existing one or become added in. Such transfer allows them to exchange their hereditary material with each other without going through a replication process. This includes the transfer of both DNA and rRNA. Horizontal gene transfer mainly happens amongst archaea or amongst bacteria. However, gene transfers across domains can also happen. The horizontal exchange of DNA makes classification of prokaryotes based on DNA more difficult due to the lack of a stable gene pool.

In the world of biology, taxonomy is the structuring of organisms into groups based on shared characteristics. Taxonomic classification forms a hierarchical tree structure where the fewer levels

there are between two organisms, the more closely related those two organisms are. This tree structure is known as the Tree of Life. The tree starts off being divided into just a few branches. Each branch is then divided further, and the branch's branches are divided further etc. Each such divide is considered a level. The main levels define the biological domain, kingdom, phylum/division, class, order, family, and species. The path between different groups in the Tree of Life reflects the evolutionary relationship between them.

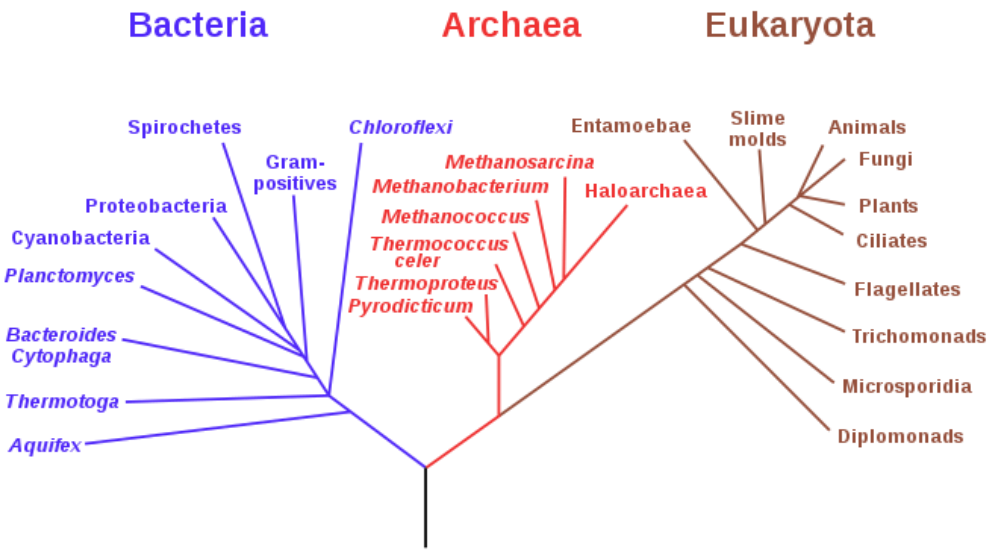


FIGURE 2.1: A STYLISED ILLUSTRATION OF THE TREE OF LIFE. THE TREE IS DIVIDED INTO THE THREE DOMAINS; EUKARYOTA, ARCHAEA, AND BACTERIA. THE LEAVES OF THE TREE ARE KINGDOMS. A COMPLETE TREE WOULD INCLUDE EACH BRANCH HAVING ITS OWN BRANCHES FOR PHYLUM/DEVISION ETC ALL THE WAY DOWN TO SPECIES. ('TREE OF LIFE (BIOLOGY)', 2021)

Knowing the classification of an organism is useful as it gives information about what features and functionalities the organism is likely to have. For example, if a microbe belongs to a family known to be harmful to humans, and the sample containing that microbe came from a human, then we know that there is a greater risk of that human being subjected to the negative effects associated with that family of microbes.

Taxonomic classification is not a process that is 100% accurate. The lines between different classes are often somewhat blurry as the diversity within a class can be greater than what separates the class overall from other classes. Individual specimens can display one or more features that are more common within other classes as compared to their own. Because of this, one criterion is often not enough for defining a class, especially at lower levels in the hierarchy where the classes are more closely related. For example, a common definition of a species is that it is as a group where two

individuals from that group can produce fertile offspring, provided they are of suitable sexes/mating types. However, this definition is not adequate, particularly when it comes to microorganisms reproducing asexually through mitosis. Defining microorganisms into classes based on their ability to sexually reproduce is therefore not viable. To make the term species more applicable to also the prokaryotic domains, other definitions are used. These other definitions include looking at an organism's karyotype, DNA, morphology, behaviour, and ecological niche. Due to the inherent fluidity both between and within biological groups, using classification as a means of discovering an organism's effect on its surroundings cannot be considered an exact science, but rather an approximation or educated guess.

### 2.1.3 Marker Genes

The genetic material of an organism will contain stretches of nucleotides that can have a function, as such constituting a gene. Most genes encode proteins, some of which have "household" functions in a cell, whereas other genes encode proteins with a more specialized function. Genes that are highly variable between groups or individuals can be used to identify individuals, populations, or species. Such genes are called marker genes, and contain polymorphisms (genetic variations) that may be used to divide individuals into distinct groups (The Editors of Encyclopedia Britannica, 2020). Such groups can for example be species, or sub-groups within a species.

Ribosomal RNA (rRNA) is often used for marker genes when studying microbiomes. rRNA is central in the process of expressing DNA into proteins, a vital process which exists in all forms of life. Using rRNA for classification is therefore widely applicable. rRNA changes over generations, but very slowly. Thus RNA-based marker genes remain relatively stable within a species and allow for detection of distant evolutionary relationships between species (Woese and Fox, 1977). However, while gene redundancy is uncommon in prokaryotic genomes, rRNA genes often have multiple copies within the same genome. While this makes them easier to detect, divergent evolution of the ribosomal RNA in the same organism can complicate classification as there is the possibility of conflicting genes putting the same organism into more than one species (Pei *et al.*, 2009).

A very widely used ribosomal RNA marker gene is 16S rRNA. 16S is currently one of the most widely used markers used for taxonomic classification and there are entire quality-controlled databases dedicated to this specific gene (Yarza *et al.*, 2014). 16S rRNA is a small subunit of a prokaryotic

ribosome and owes its popularity in part due to how slowly it evolves. The slow rate of evolution makes it possible to detect and reconstruct evolutionary relationships between species by studying the amount of divergences in the 16S rRNA, even for species which are relatively far apart in the tree of life (Woese and Fox, 1977). The slow evolution also ensures a high level of consistency within a species. As 16S is a component of prokaryotic ribosomes, this marker gene is used in classifying organisms belonging to the Archaea and Bacteria domains. 18S rRNA is a different marker gene which is homologue of 16S rRNA but found in eukaryotes rather than prokaryotes. Just like 16S, 18S evolves slowly and is therefore well-suited for reconstructing evolutionary relationships. 16S and 18S are both mainly used for high resolution taxonomic studies due to their stability and low intraspecific variability (Administrator of CD Genomics Blog, 2018).

### 2.1.4 Databases

When performing taxonomic classification of samples, it is paramount to have a reference database available with which to compare samples. However, when comparing a sample to the reference database, a 100% match is not always achievable, either due to the diversity within a species or due to the database being incomplete. Thus, when comparing data, the goal is not necessarily to find a perfect match, but rather a great similarity. Query rank is used as a measure of how similar a query is to the database (Edgar, 2016). For example, a genomic query returning a match with rank 82% could indicate that 82% of the order of nucleotides is identical to the query genome. In the world of taxonomy, any query is likely to have at least a partial match, as all life is related and thus share common traits. For instance, the genome of the fruit fly (*Drosophila melanogaster*) has an over 60% conservation as compared to the human genome (*Homo sapiens*), while the house mouse (*Mus musculus*) has a conservation of over 90% (Brandt and Vilcinskas, 2013). To avoid wrongful classification, a threshold is defined as the lowest common rank (LCR). The query needs to have rank equal to or greater than the LCR to be considered a match. This approach allows for holes in the database and variation within a species to be accounted for, while also limiting over-classification of unknown classes. The LCR is also a way to determine if a given query presents a new, previously unseen class. A query with a rank below the LCR is defined as novel data (Edgar, 2016).

In this section we will review some of the databases that currently are available.

- Ribosomal database project (RDP) is a taxonomy database that provides quality-controlled bacterial and archaeal 16S rRNA sequences, as well as fungal 28 rRNA sequences. As such this

database is best suited for marker gene analysis. The RDP database is smaller than SILVA, Greengenes, and UNITE as it only contains authoritive names, while the others also include environmental names (Edgar, 2016). Authorative names are formal scientific names, while environmental names are temporary names for species that cannot be identified or have not yet been given a scientific name.

- SILVA is a comprehensive database with ribosomal data from bacteria, archaea, and eukaryota. The SILVA contains small ribosomal RNA subunits, such as 16S, 18S, and SSU, as well as large units, such as 23S, 28S, and LSU (Official SILVA Website, 2020).
- Greengenes is a 16S rRNA gene database. The publicly available version of Greengenes uses taxonomic terms which has not been updated since 2013, and thus may be outdated (Official Greengenes database website, 2020).
- UNITE is a database as well as a sequence management tool which is mainly focused on the eukaryotic nuclear ribosomal ITS region. UNITE started out being dedicated only to the study of fungi, and therefore a fungi-only version of the database is available for download (Edgar, 2016).
- The Genome Taxonomy Database (GTDB) was created as an attempt to establish a standardised microbial taxonomy based on genome phylogeny. Phylogeny is the history of the evolution of a species, essentially, its position in the Tree of Life (Gittleman, 2016). GDTB provides a domain-to-species framework for bacterial and archaeal organisms, meaning it can be used to classify new samples at a higher level than species when species-level classification is not possible (Parks *et al.*, 2020). This database also contains a growing amount of genome samples from uncultivated microorganisms, thus giving a greater representation of the diversity of microbiomes. Currently, almost 40% of the samples in the GTDB do not have an authorative name (Parks *et al.*, 2020).

**TABLE 2.1: OVERVIEW OF DIFFERENT DATABASES.**

| Database | Domain | Important Features |
|---|---|---|
| RDP | Archaea, Bacteria, and Eukaryota | Has 16S and fungal 28S rRNA sequences. |
| SILVA | Archaea, Bacteria, and Eukaryota | Has large and small ribosomal units. Ex. 16S, 18S, SSU, 23S, 28S etc. |
| Greengenes | Archaea and Bacteria | Is a 16S rRNA database. Outdated. |
| UNITE | Mainly Eukaryota | Attempts to identify new classes through clustering. Mainly focuses on ITS region. |
| GTDB | Archaea and Bacteria | Contains a growing number of uncultivated microbiomes giving a more accurate presentation of the diversity of microbiomes. |

### 2.1.5 Designing a Taxonomic Classification Experiment

This section is about the different steps needed for a taxonomic classification experiment on microbiomes. The experiment design process has been divided into four steps to be performed in order. For each step I will present some of the alternative approaches, and some criteria to consider when selecting an approach.

#### 2.1.5.1 Step 1: Defining Scope

According to Knight et al. (2018), the first step when setting up an experiment for analysing microbiomes, is defining the scope of the experiment and selecting an appropriate experimental design (Knight *et al.*, 2018). Examples of such designs are cross-sectional, or longitudinal studies. In microbiome studies there are typically many confounding factors that, if not controlled, can hide important patterns and information in the obtained data. Limiting the scope of the experiment to only those variables that are of interest is therefore essential for deriving meaningful results. To limit confounding factors, a set of inclusion and exclusion criteria must be defined for the experiment. For example, samples that may alter the results due to influences which go beyond the scope of the project are excluded from the experiment. Other exclusion criteria are defined based on the selected experimental design. In case-control experimental design, for example, controls are matched using factors such as age, sex, medicinal use etc, depending on the specific experiment.

#### 2.1.5.2 Step 2: Determining the Type of Sequencing

When analysing the microbiome, three main methods are used to sequence the genome: marker gene, metagenome, and metatranscriptome analysis. The three methods target different parts of the genome and can produce different results (Knight *et al.*, 2018). These methods are described below.

- Marker gene analysis targets a specific region of a gene (see section 2.1.3). The type of gene selected for such experiments is typically one which is known to be highly variable between different types of microbes and is therefore highly informative about the nature of the microbe it belongs to. Marker gene analysis is less costly than some of the other methods, but produces a low-resolution overview of the microbial community being studied.
- Metagenomic analysis is a method where taxonomic classification is performed on a sample containing DNA from multiple species. The goal is then to map the genomes in the sample to their respective species. Analysing the whole genome provides more detailed information and accurate taxonomic predictions as compared to marker gene analysis. However, this process is significantly more costly and time-consuming.
- Unlike the previous two methods, metatranscriptome evaluations examine messenger RNA (mRNA) instead of DNA. The metatranscriptive method is therefore the only method that gives information on which genes are expressed, and as such the functional output of the microbe. Metatranscriptome analysis offers unique insights as compared to the other methods, and can reveal greater variation between samples from the same taxonomic class since their transcriptome is likely more different than the genetic background of different microbes.

#### 2.1.5.3 Step 3: Selecting a Method of Analysis

High-throughput sequencing (HTS) refers to cost-effective technologies for sequencing DNA and RNA (Pradhan *et al.*, 2019). HTS methods are mainly used for three types of analysis: microbe-level, DNA-level, and mRNA-level analysis. The type of analysis used during a project needs to be selected based on the type of samples available, and goal of the project.

#### 2.1.5.4 Step 4: Choosing a Tool for Taxonomic Classification

Once data has been collected, there are multiple tools available for the taxonomic classification step. This section goes through a few of the tools which are used for this purpose.

#### 2.1.5.4.1 TOOLS FOR TAXONOMIC CLASSIFICATION

Centrifuge is a tool for metagenomic analysis that uses a k-mer-based indexing schemes (Kim *et al.*, 2016). The k-mer selection process works as follows: they start with the two most similar genomes in the dataset, based on shared k-mers. These two genomes are then combined leaving out those k-mers from genome two that have ≥ 99% similarity to a k-mer in the first genome. This combined genome is then merged in the same way with the genome that has the greatest similarity to the combined genome. This process continues until it covers all genomes in the dataset. This method allows for a drastic decrease in input size.

Kraken2's k-mer algorithms achieve a fast and accurate result. However, the high memory requirements can limit its use depending on the available computational resources. In their article, Lu and Salzberg (2020) performed a series of simulations which showed that when compared to some already existing well-known tools, Kraken2 required less computational resources, executed faster, and gave a more accurate results (Lu and Salzberg, 2020).

KAIJU is a metagenomic analysis tool that finds matches by comparing sequences of amino acids, rather than nucleotide sequences (Menzel, Ng and Krogh, 2016). As such it examines the proteins present in a particular microbe. An advantage to this approach is that protein sequences tolerate several changes to the coding DNA without being altered, and are thus more stable between different individuals from a given species. Using amino acids also make it easier to classify organisms that are underrepresented in the reference database, or for which the genome is largely unknown. A disadvantage of this approach is that KAIJU will not be able to classify queries not based on non-coding regions of the genome. However, this disadvantage is somewhat lessened by the high density of protein encoding genes in microbial genomes. KAIJU has been shown to have greater sensitivity and comparable precision to k-mer based classifiers. This is particularly true for classes that are underrepresented in the reference database.

Most tools for taxonomic classification of microorganisms are based on k-mers. In their article, Vinje et al compared five different k-mer based methods for marker gene analysis (Vinje *et al.*, 2015). The methods in question were RDP, multinomial, Markov, nearest-neighbour, and pre-processed nearest neighbour. All five methods were tested on sequences of varying length of the 16S marker gene. RDP (Ribosomal Database Projct) uses the naïve Bayes classifier. The RDP method looks for the

presence/absence of words in a sequence. Multinomial also relies on the naïve Bayes principle and considers the frequency of words in the sequence. Markov considers word frequencies, just like multinomials. However, Markov does not use the naïve Bayes principle. Nearest neighbour performs classification based on multinomial probabilities. Pre-processed nearest neighbour is nearest neighbour extended with partial least squares (PLS). The results of the comparisons were as follows: Adding some extension to the RDP method, such as counting frequencies in addition to presence/absence gave slightly better results than standard RDP. On full-length 16S sequences pre-processed nearest neighbour proved to be the most accurate. For short sequences the multinomial method had the lowest error rate, implying that this method would be the most suited when taxonomic classification must happen quickly. However, no method was universally best for both full-length sequences and smaller fragments.

The classification tool CLARK discovers significant k-mers by first finding all possible k-mers for each element in the reference database, and then removing any common k-mers (Ounit *et al.*, 2015). The results are sets of discriminative k-mers that uniquely characterize the elements in the reference database. This method can be used to classify objects at a higher taxonomic level than genus by grouping together elements and creating a common pool of discriminative k-mers for the whole group. The CLARK method can be classified as a feature selection filter method.

Statistical tests can be used to select a subset of all k-mers in the database to use as input. In their article LaPierre et al (LaPierre *et al.*, 2019a) identified significant k-mers through statistical tests based on the abundance of each k-mer. They first pooled the k-mer counts, and then for each k-mer calculated the p-value using Student's t-test, followed by the Benjamini-Hochberg procedure to control the false discovery rate from doing multiple hypothesis tests. The k-mers were then sorted based on their p-values. The top 1000 k-mers were selected as features while the rest were discarded. This method can be classified as a feature selection filter method.

**TABLE 2.2: OVERVIEW OF TOOLS FOR GENOMIC CLASSIFICATION.**

| Tool | Used for | Important Features |
|---|---|---|
| Centrifuge | Metagenome | Based on careful k-mer selction through similarity. |
| Kraken 2 | Metagenome | Testing shown it can perform better than well-known tools. |
| KAIJU | Metagenome | Matches amino acids rather than nucleotides. Better for classifying underrepresented organisms. Cannot classify queries not based on protein section of genome. Shown to have greater sensitivity and comparable precision compared to K-mer based classifiers. |
| CLARK | Metagenome | Discovers discriminative k-mers through removing common k-mers. |
| Statistical k-mer selection | Metagenome | Use statistics such as Student's t-test to select the most useful k-mers. |
| RDP | Marker gene | Can be improved by extending the algorithm. Beat out by multinomial and pre-processed nearest neighbour. |
| Multinomial | Marker gene | Lower error rate on short sequences compared to RDP, Markov, nearest neighbour, and pre-processed nearest neighbour. |
| Markov | Marker gene | Beat out by multinomial and pre-processed nearest neighbour. |
| Nearest neighbour | Marker gene | Beat out by multinomial and pre-processed nearest neighbour. |
| Pre-processed nearest neighbour | Marker gene | More accurate on full-length sequences compared to RDP, multinomial, Markov, and nearest neighbour. |

## 2. 2 Machine Learning

### 2.2.1 Introduction to Machine Learning

Improvements of the available technology (e.g. genetic sequencing) has led to a significant increase in the amount of available biological data. Much of this data has a high dimensionality and the acquisition rates for new data is only increasing. The growing amount of data to sift through means that traditional analysis strategies are being challenged (Angermueller *et al.*, 2016). To handle the growing volume of data, the use of computational methods such as machine learning has become more prevalent, as they allow the extraction of information from very large datasets. Machine

learning is about developing programs that can learn and improve through data (Wang, Ma and Zhou, 2009). Machine learning is used for pattern recognition, classification, and predictions on unseen data samples (Tarca *et al.*, 2007). This is essentially a form of black box programming where an algorithm is used to optimize a performance criterion over a given set of training data (Larrañaga *et al.*, 2006). A common criterion is prediction accuracy. Machine learning makes it possible to discover functional relationships between data objects without the need for defining them a priori, making it possible to discover connections the researcher was unaware of (Knight *et al.*, 2018). For example, if the goal is to divide a set of objects into different classes, the purpose of the algorithm will be to discover how to tell if a specific data sample belongs to a certain class. This requires the algorithm to discover traits that are characteristic for a class. In other words, generalizing across individual data samples. Machine learning can broadly be divided into three paradigms; supervised learning, unsupervised learning, and semi-supervised learning (Tarca *et al.*, 2007). In supervised learning the algorithm is presented with both the training data and the correct class for the data. During training, the algorithm tries to generalize the classes based on the training data to allow it to come up with the correct class for new, unseen data samples. In unsupervised learning the algorithm is not provided with any classes, only the training data. The algorithm will then look for similarities between the data samples to define new classes. Semi-supervised learning is a combination of supervised and unsupervised learning. This paradigm is typically used when some of the training data have a known class but much of it does not (Tarca *et al.*, 2007).

### 2.2.2 The Dataset

#### 2.2.2.1 Data Pre-processing

According to Chicco (2017), the preparation of the dataset is one of the most important aspects of a successful project involving machine learning (Chicco, 2017). The dataset is the basis on which the predictive model is built. Therefore, an insufficient dataset will create an insufficient model. In this case, the dataset is a reference to the data that is used to train, test, and evaluate the model generated by the machine learning algorithm. The process of preparing the dataset is known as data pre-processing. The following paragraphs describe some of the main concerns for data pre-processing.

#### 2.2.2.2 Size

Machine learning algorithms benefit from having a large dataset on which to train (Chicco, 2017). Large datasets allow the design of a more accurate model as they can display more of the variety

that might exist within each class, making the model better at distinguishing intra-class variation and between-class variation. Ideally, the dataset should contain ten times as many data instances as there are data features.

#### 2.2.2.3 Cleanliness

A clean dataset is one where all corrupt, inconsistent, inaccurate, and outlier values have been removed. In this section we will go through different aspects of cleaning a database.

#### 2.2.2.4 Missing Values

Errors during data gathering can result in data records where parts of the record are missing. When this occurs, the records can be deleted, or filled in by the researcher. The process of guessing the value of missing data is called imputation (Lee, 2017). This can sometimes be necessary when the method being used requires that there be no missing values.

#### 2.2.2.5 Inconsistent values/noise

Inconsistent values do not make sense with what they represent. For example, a table of animals should not classify cod as being a mammal. Removing inconsistencies from the dataset reduces error and thus improves accuracy. Knowing that something is an inconsistency may require domain knowledge, such as the example of the cod. However, in some cases there are ways to detect that something is likely an inconsistency with little to no domain knowledge. Inconsistent data is also referred to as noise.

#### 2.2.2.6 Encoding Categorical Inputs

Many machine learning algorithms work better with numerical inputs than categorical inputs. Therefore, a challenge when working with a dataset with categorical inputs is converting these into numerical values. Some often-used methods are one-hot encoding and label encoder.

Label encoding transforms each value in a column to a number (Yadav, 2019). An issue with this approach is that when categories are represented by numbers, they can be compared in a way which does not make sense with what they represent. Take for example, a column of different cat-breeds denoted in text. When the names of the cat breeds are translated into numerical values, a

hierarchy/order is introduced between them where cat breed 1 and 2 can be considered more closely related than cat breed 1 and 9 for no other reason than the distance between the numerical labels. However, if the categories do have such relations between them, label encoding provides a simple way of representing this in the dataset. For example, a column of price levels, such as cheap, medium, expensive etc. Assuming these categories are given a numerical value in order it would be correct to assume that price level 0 and price level 1 are closer than price level 0 and price level 2. Therefore, label encoding is recommended for ordinal categories (Shrivastava, 2019).

One-Hot encoding transforms the categorical values into new columns containing a one or zero denoting true or false for each category (Yadav, 2019). This removes the ordering problem introduced by label encoding as there is no implied relation between the different categories. However, this approach introduces new columns to the dataset and can greatly expand the dataset if there are many categories. One-hot encoding is recommended for nominal categories (Shrivastava, 2019).

#### 2.2.2.7 Normalization

For datasets that contain numerical data normalization, it is often needed to transform the dataset into a common frame. This means transforming the values to fit into an interval with a minimum and maximum value. A typical interval is 0 to 1. Below we describe some techniques normalizing data.

- **Log Transform:** Log transform is a technique for dealing with skewed data. Statistical analysis often relies on the dataset following a normal distribution. When this is not the case, log transform fixes it by transforming each variable x into log(x) (Htoon, 2020). The transformation can use base two, base ten, or natural logarithms. The choice of base depends on the purpose of the analysis. For this to work however, the data must already be approximately following a log-normal distribution. Log-normal distribution means the logarithm of a continuous variable follows a normal distribution.
- **Scaling:** Inputs which are continuous variables are often scaled so that they have a mean of zero and a variance of one. This means the continuous variables have a value between 1 and -1. The purpose of scaling is to align different features to more similar magnitudes, as many methods work better when this is the case (Lee, 2017). For example, in neural networks the weights of the various nodes will have a more equal effect on variables with a similar magnitude.

#### 2.2.2.8 Splitting the Dataset

After creating a model, it is not irrelevant which data instances are used for testing. If the same data samples are used both during training and testing, the model will produce overtly optimistic results as the model has been trained to classify those exact data samples (Chicco, 2017). This problem can be avoided by splitting the dataset into different sets; one for training and another for testing. The lack of overlap between sets ensures that testing evaluates the model's accuracy in classifying new samples that do not exactly match the samples seen during training. It can also be argued that the dataset should be split into a third category as well; a validation set. In many cases the machine learning algorithm has one or several hyper-parameters which greatly impact the resulting model. Hyper-parameters are values that determine some part of how the algorithm operates or its initial values. For example, when using a K-nearest neighbour algorithm the value of K is a hyper-parameter. To arrive at the ideal value for the hyper-parameters, models are put through an optimization phase. This means training the model with different hyper-parameters to determine which value gives the best performance. The validation set is used during this optimization phase. A common way to split the dataset is to use 50% as the training set, 30% for the validation set, and 20% for the test set. However, should the dataset be too small to allow only 50% to be used for training, other approaches such as cross-validation can be the solution (Chicco, 2017).

#### 2.2.2.9 The Imbalanced Dataset Problem

An imbalanced dataset is one where one class is overrepresented compared to other classes in the dataset. In this scenario, it is easy to end up with a model that is prone to giving false positives for an overrepresented class, and false negatives for an underrepresented class. This is often a problem in bioinformatics as some microorganisms have received far more attention than others resulting in datasets where some classes have extensive records, while others have barely been recorded at all. There are several techniques that handle the imbalanced data problem, such as under-sampling or data class weighting (Chicco, 2017).

#### 2.2.2.10 Dimensionality of Input Data

Dimensionality refers to the number of input variables or features used in the machine learning algorithm. A feature captures some characteristics of an object. High dimensionality increases the complexity of predictive modelling and reduces the performance of the algorithm (Brownlee, 2020a). In machine learning projects this is often a major concern as they often include large amounts of

data, which coupled with high dimensional input create a real challenge when computational resources, such as memory and CPU time are limited. In cases where the dimensionality is greater than the number of samples, overfitting becomes very likely (Tarca *et al.*, 2007). The above-mentioned issues can be dealt with by using dimensionality reduction techniques to decrease the dimensionality. Dimensionality reduction can be divided into two categories: combining existing features into new ones, or feature selection. When working with k-mers (see section 2.3.1), each k-mer used as input is a dimension. Reducing dimensionality therefore means reducing the number of k-mers. Focusing on only those k-mers that are significant for classification can improve the performance and accuracy of the resulting predictive model. However, this requires defining and discovering which k-mers are significant. The goal of dimensionality reduction is ultimately feature space compression. Feature space compression means classification using fewer, but better features. This is an NP-hard problem. This means only a brute force approach can ensure the optimal solution is found. However, due to the size of the possible feature space, brute force is not feasible. Instead, various methods are used to approach an approximate solution which works for the project. These are discussed below.

- **Feature selection:** During feature selection, a subset of all features is removed and will not be used in the predictive model. The features which are removed should be the ones with the least positive impact on model accuracy. Feature selection methods are divided into two categories: filter and wrapper methods. Filter methods filter out features from the input space using some threshold to determine relevance. Wrapper methods decide which features to use by comparing the accuracy score of the resulting predictive model. Filter methods are simpler to implement but cannot discover the impact that combinations of features can have on model accuracy (Tarca *et al.*, 2007). Wrapper methods are more complex to implement and require more computational resources but can discover the benefits of combinations of features (Tarca *et al.*, 2007).

  Not all k-mers will be equally useful for classification. Some k-mers might be too common and will thus increase the dimensionality and computational costs without also providing meaningful information. A k-mer can also be removed if it is highly correlated with another to the point that it can reasonably be assumed the presence of one means the presence of the other. In this case one might remove one of these k-mers. Correlated k-mers, or correlated features in general, can be discovered using a correlation plot. The above are examples of filtering k-mers. Reducing k-mers using wrapper methods would require defining a method of subset selection, and a heuristic algorithm to determine when the optimal

subset selection has been found. When accurately determining the optimal subset is not feasible, an approximate heuristic function can be used rather than an exact heuristic function. An approximate heuristic function can be either deterministic or stochastic. A deterministic function always returns the same result, while a stochastic one can give a different result each run.

- **Combining features:** Combining features includes creating new features from existing ones, which then represent the original features they originated from.

  In the case of k-mers, these can be combined if there is a strong correlation between different k-mers. For example, if two k-mers are rarely present without the other also being present, the k-mers can be combined and counted as one feature rather than two separate features. Another option is to create a new feature to represent combinations of k-mers with a special relationship. A special relationship means the combination of certain k-mers being part of the same organisms holds some significant meaning. However, discovering such relationships would require expert domain knowledge and/or extensive testing.

### 2.2.3 Comparing Prediction Ability Between Models

Defining a standard metric on which to evaluate a model is essential when comparing multiple models on their prediction performance. Metrics allow for an objective evaluation that can be applied to all models equally and provide a measure of their prediction ability.

The most commonly used metric is classification accuracy (Mishra, 2020). Classification accuracy is the ratio between correct predictions to the total number of predictions. Accuracy can be calculated using the below formula.

$$Accuracy = \frac{Correct\ predictions}{All\ predictions}$$

Accuracy is a simple metric that is comparable across models. The main disadvantage to accuracy is that when the number of samples belonging to each class is not even, the evaluation can be misleading (Mishra, 2020). For example, if the network is classifying images of cats and dogs, where 90% of the data samples are images of cats, the model could achieve a 90% accuracy by classifying

everything as a cat. While this would be a correct score, a model that cannot differentiate between classes is not useful.

Another common metric is precision, also known as positive predictive values. Precision determines the percentage of positive predictions that are correct (Saxena, 2018). A positive prediction is when a sample has been predicted to be part of a class, rather than not part of a class. The formula for precision is displayed below.

$$Precision = \frac{CorrectPositivePredictions}{CorrectPositivePredictions + WrongPositivePredictions}$$

Precision gives an indication of how certain we can be that a positive classification is correct. A high precision would indicate that any sample the model has classified as being of a certain class has a high probability of being correct.

## 2.2.4 Neural Networks

### 2.2.4.1 Introduction

An artificial neural network is a supervised machine learning technique inspired by the neural networks in the human brain. When a person is subjected to different types of stimuli, such as seeing, smelling, tasting etc, certain neurons in their brain are activated, and those neurons may activate other neurons (BrainFacts/SfN, 2012). As there is a connection between stimuli and which neurons are activated it should be possible to predict something about the stimuli by looking at which neurons have been activated and how activated. This is the underlying idea that artificial neural networks are inspired by. One of the main benefits of neural networks is their ability to discover complex non-linear relations between data (Hammerstrom, 1993). Neural networks also show greater fault tolerance compared to other methods due to its ability to generalize (Hammerstrom, 1993). To generalize a network must discover those features that all or most of the samples from a certain class have in common. In this case, fault tolerance means the network is less sensitive to noisy data (see section 2.2.2.5). In general, neural networks are excellent when it comes to recognising trends and patterns and can be applied to a wide range problems (Abiodun *et al.*, 2018).

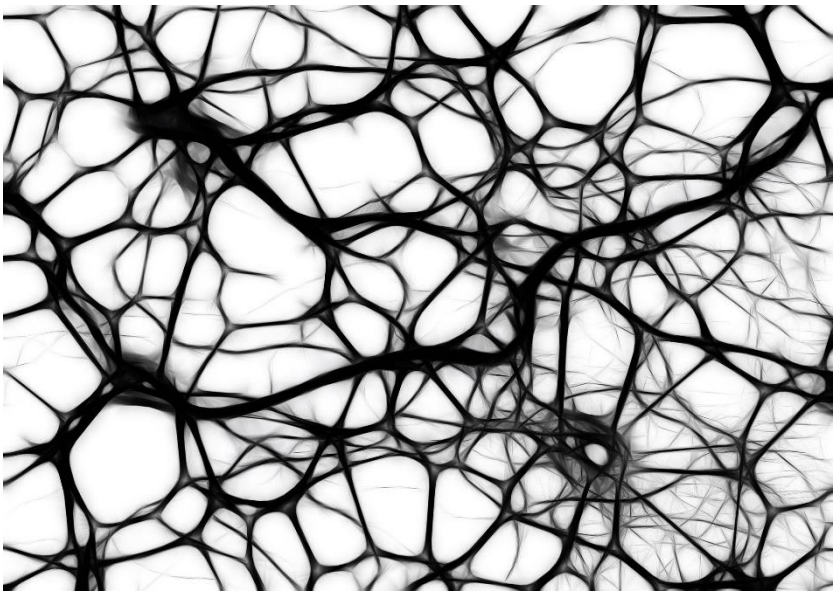

FIGURE 2.2: STYLIZED IMAGE OF NATURAL NEURAL NETWORKS IN THE BRAIN.

The artificial neural network is built up of several nodes, called neurons, ordered into layers (Tegmark, 2017). Each neuron takes input and performs some calculation on said input to produce some output. This output is then sent to the next layer. The number of layers varies greatly. However, all artificial neural networks must have an input layer, and an output layer. Typically, there will also be any number of layers in between, called hidden layers. Once the input has gone through every layer, the network attempts to place the input into a class. This is called predicting. For the rest of this section artificial neural networks will be known simply as neural networks.

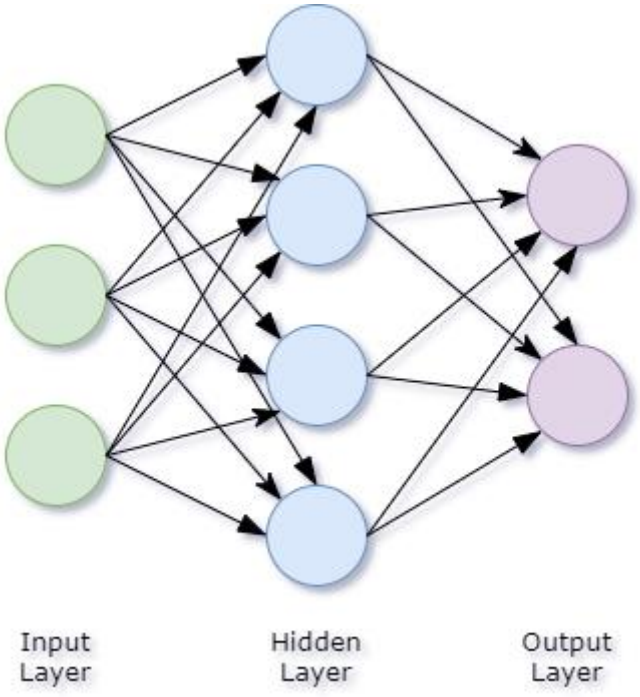


FIGURE 2.3: A SIMPLE NEURAL NETWORK WITH AN INPUT LAYER, OUTPUT LAYER, AND ONE HIDDEN LAYER.

#### 2.2.4.2 Training a Neural Network

All machine learning methods must be trained to generalize across classes allowing them to predict the correct class of new data. For neural networks this takes the form of weight regularization. The connections between neurons in the hidden layers of the network are coupled with a weight that determines the power of that neuron's input in affecting the final classification. As such, weights are a way of distinguishing important features. The weights also prevent small variations in the input from having excessive consequence on the result (Tarca *et al.*, 2007). When a neural network is being trained, the weights in the network are continuously being updated to try to find the combination of weights leading to the most accurate predictions. The following paragraphs will go into this process in greater detail.

Upon initialization of the neural network, the weights are arbitrarily assigned. Typically, these values follow a normal distribution with a mean of zero and a standard deviation of one. The inputs from the training set are then sent through the model. At each layer the input is transformed using an activation function (see section 2.2.3.3). The result of the activation function is then sent to the next layer. When the input has gone through the whole network, the network tries to predict the class of the input by coming up with a probability for each class. This probability is then scored using a loss function (see section 2.2.3.4) by comparing it to the correct probabilities. This process is repeated for every input in the training set. Then the weights in the network are updated based on the result. How the weights are altered depends on the chosen optimization algorithm (see section 2.2.3.5). One round of going through the whole training set, evaluating the results, and updating the weights is called an epoch. Training a neural network typically includes multiple epochs. When the network is improving through multiple epochs, we say that it is learning.

While the neural network trains on the entire training set during an epoch, typically it does not train on the entire training set at the same time. Instead, the training set is divided into batches, which are subsets of the training data. The model then trains on each batch separately. The number of times the network is trained per epoch is therefore the length of the training set divided by the batch size, or as a formula; $workouts\ per\ epoch = \frac{size\ of\ training\ set}{batch\ size}$ . Large batches allow the epoch to finish faster and reduces resource usage as it reduces the number of times the model must run. However, research has shown that large batches can make the model worse at generalizing, although it is uncertain why this is the case (Shen, 2018).

### 2.2.4.3 Activation Functions

#### 2.2.4.3.1 INTRODUCTION

Neurons have an activation function. The activation function determines the output of the neurons given the inputs from the neurons in the previous layer. The activation function takes in the weighted sum of the inputs and produces a new value to send to the next layer. The purpose of the activation function is to determine to what extent the neuron is "activated". Activation functions typically produce a value between a lower and an upper limit, for example 0 and 1. The closer the value is to the upper limit, the more "activated" the neuron is. The idea is that depending on the input, some neurons will be more activated than others, and similar inputs will activate the same neurons. This allows the model to perform a prediction on the input based on which neurons have been activated and which ones have not. This idea stems from how different neurons in the human brain are activated depending on the type of stimuli a person is subjected to.

#### 2.2.4.3.2 THE VANISHING GRADIENT PROBLEM

The vanishing gradient is a common issue when training a neural network. This problem occurs when the gradient being calculated from backpropagation is a very low number, meaning close to 0 (Wang, 2019). When the weights are updated, they are updated in proportion to the size of the gradient. Therefore, small gradients do not alter the model much which can cause it to stagnate and not learn effectively. The opposite of the vanishing gradient is the exploding gradient. This occurs when the gradient is very large, meaning greater than 1, which causes the weight to be drastically altered during training (Pykes, 2020). Such drastic changes can keep the network from finding the ideal weight for its nodes as they change too much with each iteration to get close to the weight that would result in the best outcome. Both the vanishing and exploding gradient problems are worse for deep neural networks in comparison to simpler networks (Brownlee, 2019a). This is because the value of the gradient is the product of a calculation that depends on the gradients later in the network. The more gradients the vanishing or exploding gradient depends on, the more likely it is that the value of the gradient will be pushed further in the same direction, low gradient becoming even lower and high gradient becoming even higher.

#### 2.2.4.3.3 TYPES OF ACTIVATION FUNCTIONS

There are two main types of activation functions: linear and non-linear. When the activation function is linear no transformation is performed on the input. Networks that use linear functions are easier

to train but are less capable of learning complex relationships in the data. Therefore, non-linear functions are often preferable. However, non-linear functions are vulnerable to the vanishing gradient problem. Two of the most common non-linear functions are sigmoid and tanh. There are, however, some functions that are not entirely linear or non-linear. These are called piecewise linear functions and are linear for only parts of the input space. A popular piecewise linear function is rectified linear unit (ReLU). The mentioned activation functions are described in greater detail below.

- The sigmoid function is also called the logistic function and can be mathematically described as below (Chaudhary, 2020).

$$f(x) = \frac{1}{1 + e^{-x}}$$

  Sigmoid transforms the input to be 0 ≤ and ≤ 1. The function for all possible inputs forms an S-shape with 0.5 as the midway-point. An issue to the sigmoid function is that it is especially vulnerable to the vanishing gradient problem due to the output not being zero-centred (Chaudhary, 2020).
- The hyperbolic tangent function (tanh) can be described mathematically as below (Chaudhary, 2020).

$$f(x) = \frac{e^{x} - e^{-x}}{e^{x} + e^{x}}$$

  The tanh function transforms the value of the input to be -1 ≤ and ≤ 1. The function for all possible inputs forms an S-shape with 0 as the midway-point. Tanh suffers from the vanishing gradient problem, but less than sigmoid due to the output being zero-centred (Chaudhary, 2020).
- Rectified linear unit (ReLU), can be described mathematically as below (Brownlee, 2019a).

$$f(x) = \max(0, x)$$

  The ReLU function returns the value of the input directly, or the value 0 if the value of the input is ≤ 0. This is a piecewise linear function and acts linear for half the input space, in this case for all inputs > 0. ReLU is not sensitive to the vanishing gradient problem as it acts like a linear function. However, since it is a non-linear function, it also maintains the ability to learn complex relationships in the data.

- Normalized exponential function (softmax) is often used as the last activation function to normalize the output from the network. Softmax takes a vector of values, applies the standard exponential function to each element in the vector, and then divides by the sum of the exponentials. The result is a vector where all elements are 0 ≤ and ≤ 1, and the sum of all elements is 1. The output looks like a set of probabilities for mutually exclusive classes. The softmax function can be described using the following formula (Wood, 2019).

$$softmax\ vector = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}}$$

$z_i$ are the elements from the input vector. $e^{z_i}$ is the exponential function applied to each element in the input vector. K is the number of classes in the training set. $\sum_{j=1}^{K} e^{z_j}$ is the normalization term.

#### 2.2.4.4 Loss Functions

A loss function is a measurement of the error that is achieved by comparing the output from the neural network with the correct output. The purpose of a loss function is to provide a score which summarizes the quality of the model, and where improvement in the score signifies an improvement in the model (Brownlee, 2019c). As the score provided by a loss function represents the error in the model, the goal of training is always to get the loss function to score as close to 0 as possible, and as we can never have a negative amount of loss, the score produced by a loss function is always ≥ 0.

- A common loss function is mean squared error (MSE). MSE provides a measurement of loss by taking the difference between the predicted output and the correct input for every data sample, then squaring each of them and finding the average. This can be summarized by the below function where $e$ is the difference between predicted and correct output and n is the number of data samples in the training set (Binieli, 2018). The advantage of MSE is that we do not get outlier predictions with large errors as MSE puts a lot of weight on such errors. However, in many cases reducing the error on outlier values is not a priority and we would rather have a model that performs better on most inputs.

$$mse = \frac{{e_0}^2 + \cdots + {e_n}^2}{n}$$

- Mean absolute error (MAE) is a slightly altered version of MSE that improves on MSE's disadvantages. The only difference is that in MAE takes the absolute value of e rather than square it. The formula is displayed below. MAE weights all errors on the same linear scale,

thus putting less emphasis on outliers. This results in a model that performs well on most data but can produce large errors when it comes to outliers (Seif, 2021).

$$mae = \frac{|e_0| + \cdots + |e_n|}{n}$$

- Categorical-cross entropy is a common loss function for deep neural networks on multi-class classification problems (Gordon-Rodriguez *et al.*, 2020). Categorical-cross entropy is based on calculating the difference between two probability distributions. This loss function has been found to have advantages over other functions such as squared-error (Vilone and Longo, 2021). Cross-entropy uses logarithms to increase the training speed for neural networks. The formula for categorical cross-entropy can be found below (Mody, 2020). C is class id, o is observation id, and p is probability.

$$-\sum_{c=1}^{M} y_{o,c} \log\left(p_{o,c}\right)$$

#### 2.2.4.5 Optimization Functions

Optimization algorithms change attributes of the neural network such as weights and learning rate to reduce the loss function (see section 2.2.3.4) (Doshi, 2020). There are several optimization algorithms that solve this in different ways. In this section we will describe some of these algorithms.

##### 2.2.4.5.1 GRADIENT DESCENT

Gradient descent is one of the most basic and common optimization algorithms (Doshi, 2020). The goal of gradient descent is to calculate the direction in which a weight should be altered to minimize the loss function. Another way of describing this is that gradient descent finds a local minimum on a differential function (Chauhan, 2020). The process works as follows; start with the initial value. Calculate the derivative of the function to find the direction of the slope. If the slope is positive, the weight should increase, and if the slope is negative the weight should decrease. A derivative of 0 means the algorithm has found the local minima. This process is repeated once in every epoch. After running gradient descent, the new weight can be described using the below formula.

$$w = w0 - \frac{loss}{w0} \times LR$$

W is the new weight, loss is the size of the error as calculated by the loss function, w0 is the current weight, and LR is the learning rate. The learning rate is typically a value between 0.001 and 0.0001.

Setting the learning rate too high risks skipping the ideal weight. However, setting the learning rate too low risks the model needing a lot more time to train.

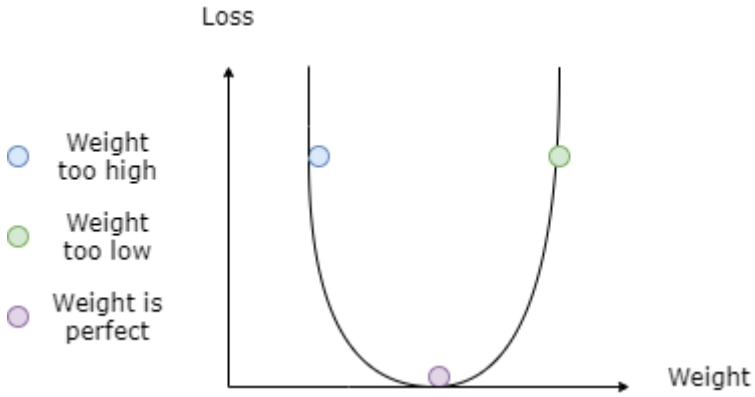


**FIGURE 2.4: STYLIZED FUNCTION ILLUSTRATING HOW TO DETERMINE THE DIRECTION IN WHICH THE WEIGHT SHOULD BE ADJUSTED.**

The main advantage of gradient descent is that it is simple to understand and calculate. Some important disadvantages, however, are that this algorithm may become stuck in local minima and is known to use a lot of memory (Chauhan, 2020).

#### 2.2.4.5.2 STOCHASTIC GRADIENT DESCENT

Stochastic gradient descent is a variant of gradient descent. The difference is that stochastic gradient descent updates the weights after calculating the loss on each training sample (Doshi, 2020). This means that if the training set contains 1000 samples, the weights will be altered 1000 times during one epoch. Some advantages to this approach are that it is faster (Paine *et al.*, 2013) and requires less memory than standard gradient descent and is less likely to be trapped in a local minima (Doshi, 2020). However, this algorithm may continue changing the weights even after finding a global minima (Doshi, 2020).

### 2.2.4.6 Over and Underfitting

Over- and underfitting refer to how well a trained machine learning model fits the training and test sets.

#### 2.2.4.6.1 OVERFITTING

A model that has overfitted the data, has learned to recognize and predict the training data, but failed to generalize about the categories it is predicting (Brownlee, 2018a). As a result, the model will

perform poorly on unseen data if it deviates slightly from training samples from the same category. If a model performs well during training, but poorly during testing, this is a sign of overfitting.

One way to reduce overfitting is by increasing the diversity of the training set. If the model is given many different samples from the same category, it becomes more likely that the model will discover the general features that define the category rather than the unique characteristics that define the individual.

If acquiring more training data is not an option, data augmentation is another alternative. Data augmentation includes artificially increasing the diversity of the training set by introducing modified versions of samples into the training set (Sikka, 2020). Overfitting can also be reduced by reducing the complexity of the model. A less complex model will be less capable of picking up subtle differences in the dataset and might therefore generalize better.

Regularization is a technique that can help reduce overfitting by reducing the complexity of the model (Brownlee, 2018a). This is achieved by altering the loss function (see section 2.2.3.4) to penalize for complexity, and by boosting the loss value when some weights are relatively large. This pushes the model towards smaller weights, which in turn lessens the importance of each layer. In some cases, a layer can have weights that are so small the layer becomes almost irrelevant.

Overfitting can also come about because of overtraining. Reducing training time can thus reduce overfitting.

A last technique for reducing overfitting is dropout (Versloot, 2019). Dropout prevents overfitting in neural networks by making the neural network randomly ignore a subset of the nodes in a layer.

#### 2.2.4.6.2 UNDERFITTING

Underfitting is the opposite of overfitting. A model that is underfitting the data performs poorly both on training data and test data (Brownlee, 2018a). Underfitting occurs when the model is not discovering the features needed to classify the data. Typically, underfitting can be a sign that the model is not sophisticated enough as compared to the data it is training on.

Underfitting can be improved by increasing the complexity of the model (Brownlee, 2018a). This can be done by increasing the number of layers, the number of nodes in the layers, or adding different types of layers.

Adding more features to the training set can also improve underfitting by giving the model more data to work with (Sikka, 2020).

### 2.2.5 Deep Learning

#### 2.2.5.1 What is Deep Learning?

Neural networks come in many forms. The simplest neural networks have only three layers. When neural networks get more layers, they are called deep neural networks. Deep neural networks are a form of deep learning. Deep learning is a subfield within machine learning consisting of techniques where larger architectures are used to represent higher level features than what is possible with other techniques. There is no clear divide between traditional neural network and deep learning as there is no commonly accepted size requirement for a neural network to be considered deep. However, it must have at least the regular three layers. Some popular deep learning techniques are; multilayer perceptron networks, convolutional neural networks, and recurrent neural networks (Brownlee, 2019d). Deep learning is a powerful tool when there are large datasets available to work with, and as such is ideal for big data. In a talk on deep learning, Ng stated that unlike traditional machine learning techniques whose performance plateau when enough data has been collected, deep learning techniques are able to continue improving their performance as more data is added (Ng, 2015). The implication is that deep learning will surpass traditional methods given enough data.

#### 2.2.5.2 Deep Neural Networks

##### 2.2.5.2.1 MULTILAYER PERCEPTRON NETWORK

A multilayer perceptron network is a deep neural network with at least one hidden layer. This is the most common deep neural network (Uniqtech, 2019). Multilayer perceptron networks are flexible and can be applied to a wide range of prediction problems (Brownlee, 2018b). Multilayer perceptrons can detect patterns that are too complex to be noticed by humans and other machine learning algorithms. A network with one hidden layer has been proven to be a universal approximator as long as it has a sufficient number of hidden nodes (Hornik, Stinchcombe and White, 1989).The disadvantages of multilayer perceptron networks are the same as for neural networks in general. Such as the lack of rules or best practises in determining the structure of the network, making the process of finding a suitable architecture time-consuming. Neural networks are also strict on the input format as they can only work on numerical data (Gupta, 2020). The structure of the input also has direct consequences on the network's performance increasing the burden of careful pre-processing (Gupta, 2020).

#### 2.2.5.2.2 CONVOLUTIONAL NEURAL NETWORK

A convolutional neural network is a deep neural network that includes convolutional layers. The neurons in a convolutional layer can extract higher-level features compared to standard fully connected layers (Nguyen *et al.*, 2016). The convolutional layer consists of maps of neurons whose size is equal to the dimensionality of the input (Angermueller *et al.*, 2016). Each neuron in the map is connected to a subset of the neurons from the previous layer. This subset is called the receptive field. Each of these neurons looks for a feature by computing the weighted sum of all inputs from its receptive field and applying an activation function. What each map does in principle, is to look through the input for a certain pattern. For example, the map could detect sequence motifs. A sequence motif is a short recurring pattern in a DNA sequence assumed to have some biological function. A convolutional neural network often consists of multiple convolutional layers allowing the network to learn to recognise increasingly complex features. Convolutional networks also often include pooling layers. Pooling layers create a summary of adjacent neurons. Typically, by finding the maximum or average value produced by the neurons. The use of pooling layers relies on the assumption that the exact position and frequency of features is irrelevant for the final classification (Angermueller *et al.*, 2016). Convolutional networks have been shown to be very accurate on image recognition problems and detect important features without human interaction. However, convolutional neural networks require a lot of training data and are computationally slow (Diaz, 2016). While convolutional models have mainly been popular for image recognition, research suggests it could have potential for taxonomic classification as well, even if more exploration is needed (Khawaldeh *et al.*, 2017). For example, in a project by Vue et al, a convolutional network outperformed a number of other machine learning techniques on fungal classification (Vu, Groenewald and Verkley, 2020).

Convolutional layers are what defines a convolutional neural network as this is where convolution occurs. A convolution is a linear operation where the input is multiplied by a set of weights (Albawi, Mohammed and Al-Zawi, 2017). The set of weights is known as a filter. Initially, the weights in the filter are randomly generated upon initialization, just like the weights in a standard neural network. During convolution, the input vector is divided into smaller windows of given length, which are multiplied with the filter. Each filter used during convolution can detect a certain pattern in the input due to having different weights. Hence, having many filters allows the network to detect more patterns in the input. The result, after applying the filters to the input, is collected in a feature map. The feature map is then passed to an activation function just like in a standard neural network.

When implementing a convolutional layer, the most important properties are the number of filters, kernel size, stride, and padding. Each filter in a convolutional layer can detect a different pattern. The number of filters therefore determines how many patterns that convolutional layer can detect. Kernel size determines the size of the windows being multiplied by the filters. The kernel size should be lower than the input size so the filters can be applied many times. Stride determines how much the window moves through the input before performing a convolution on the input. Typically, the stride is 1 to ensure all filters are applied to the entire input sequence and the filters do not miss patterns. Padding determines what happens when the window is near the edges of the input. For sequential input this is the start and end of the sequence. Padding adds extra data to the end of the input, thus increasing the number of convolutions. This allows the edges to be processed in the same fashion as the data in the middle of the input.

## 2.3 Nucleotide K-mers as Features in Machine Learning

### 2.3.1 Introduction to K-mers

Due to the length and number of DNA sequences, using the whole sequence in classification tasks requires enormous computational resources due in part to the high memory requirements (Bucak and Uslan, 2011). These requirements make the classification task expensive and depending on the size of the DNA sequences and reference database even infeasible. To lower the resource requirements, it is necessary to reduce the size of the input itself. This can be achieved using k-mers rather than complete sequences. K-mers are extracted pieces of DNA (Brihadiswaran, 2020). The k denotes the number of nucleotides in the k-mer. For example, an 8-mer contains eight nucleotides. By dividing the DNA sequence into k-mers and then using a subset of those k-mers as input for the classification algorithm, the computational requirements are drastically reduced. Another benefit of k-mers is that the requirements for genome completeness are lower. The DNA sequence may be incomplete with missing pieces, making it impossible to determine the correct order of the whole sequence. Such errors in the data matter less when the sequence is divided into k-mers. The classification also becomes less affected by indels. Indels are small insertion-deletions in an organism's DNA (Lin *et al.*, 2017). In a complete DNA sequence-indels can shift long sub-sequences, reducing the complete sequence's comparability with other sequences. When using k-mers however, only a limited number of k-mers are affected by the indel reducing the importance of such irregularities.

When sequencing DNA, the DNA strand can be read in either direction. Since the direction in which the DNA sequence is being read is irrelevant to classification, differentiating between a k-mer and its reverse-complement is typically not necessary as these are considered the same k-mer. Canonical k-mers is a method of ensuring a k-mer and its reverse complement are always represented in the same way. When a k-mer pair is treated as canonical, both k-mers will be represented by the lexicographically smallest k-mer (Clavijo, 2018).

### 2.3.2 K-mer Length

K-mer length can have a significant impact on the classification tasks, and there are advantages and disadvantages to choosing shorter or longer k-mers. In general longer k-mers contain more information as they are more detailed (Kaehler, 2017). They are also more likely to discover areas where shorter sequences are repeated multiple times in a row. For example, the nucleotides AT being repeated five times in a row. However, long k-mers increase memory requirements. The correlation between k-mer length and computational requirements is strong. This is because there is a direct relation between k-mer length, and the dimensionality of the input vectors used by the machine learning algorithm. The longer the k-mer, the more distinct k-mers exist in the dataset. The more distinct k-mers there are, the more k-mers need to be counted or otherwise represented in the input vector to represent the DNA sequence. For example, 16-mers creates the potential for up to 4 294 967 296 distinct k-mers, while 8-mers allows upwards of 65 536, and 3-mers create a maximum of only 64. In their article, LaPierre et al tested various k-mer lengths for their project on machine learning in metagenome-based disease prediction and empirically found k=12 lead to sufficiently significant k-mers for classification while still being computationally feasible (see section 2.1.5.4.1) (LaPierre *et al.*, 2019a). Significant k-mers were discovered by conducting statistical tests on the abundance of each k-mer by calculating the p-value of each k-mer using Student's t-test. They then sorted the k-mers by p-value and retained the top 1000 k-mers.

### 2.3.3 K-mer Representation for Machine Learning

#### 2.3.3.1 Introduction

Machine learning models cannot work on raw, textual data. Instead, they require the input data to be represented in a numerical format (Brownlee, 2020b). Therefore, it is necessary to find a numerical way of representing k-mers. There are two main approaches to representing k-mers; presence/absence and k-mer counting. Presence/absence represents each k-mer as an equivalent of

a boolean value signifying whether the k-mer is present or absent in the DNA. DNA sequences are presumed to be in the same class if they contain the same k-mers. K-mer counting represents the DNA by providing a count for each distinct k-mer. The idea is that DNA sequences that have the same k-mers at similar frequencies are more similar than those that do not. This similarity is assumed to signify taxonomic similarity and is used to classify those sequences as belonging to the same class. K-mer counting and presence/absence are both popular methods of assessing similarity between sequences of DNA in taxonomic classification projects. However, there is no commonly accepted best practice as to how either method should be implemented. The size of the representations, the length of the k-mers, the data structures to base the representation on etc vary between projects. In this section we will go through some of the different ways to implement k-mer representation in practice.

#### 2.3.3.2 Issues When Representing K-mers

There are several issues to be considered when implementing k-mer counting or k-mer presence/absence. In this section we will present three major concerns which the selected solution must address.

- K-mer counts can be represented as a vector of numerical values where each position in the vector represents a distinct k-mer that exists in the dataset and the numerical value is the count for that k-mer. Presence/absence would be represented in the same way except the numerical value would be binary. This method is simple to implement and results in vectors that are easy to compare and no k-mers are overlooked. However, a major concern with this solution is the computational requirements. A dataset where we use k=8 can contain as many as 65 536 distinct k-mers. Each DNA sequence would therefore potentially have to be represented by a vector of length up to 65 536. For longer k-mers the vectors would have to be even longer. In a dataset with many organisms to compare, working with such large data quickly becomes computationally infeasible. There is the distinct possibility that these vectors would to a large extent contain zeros, assuming most k-mers are present in a limited number of the genomes. The consequence could be an aggravation of the vanishing gradient problem as inputs of zero would result in the gradients calculated to update the weights being small (see section 2.2.4.3.2).
- When considering how to reduce the computational cost of k-mer representation, it is necessary that any solution must not compromise the comparability of the representation vectors. For example, a simple way of reducing the length of vectors, is to only include those k-mers that are present in that DNA sequence. If the DNA sequence includes a lot of

repeated k-mers this would significantly reduce the needed vector length. However, this would mean that the value at a specific position in one vector, is not necessarily comparable to the value at the same position in a different vector as they might not be counting the same k-mer. This solution would also not work for presence/absence-based representation, as it would result in identical vectors. This leads to the requirement that the sequences must be represented in one of the two following ways.

1. All sequences are represented by vectors of fixed length that contain the same k-mers in the same order. Each element in the vector is then comparable to the element in the same position in all other vectors.
2. The sequences are represented by vectors of varying length that contain different k-mers in an order that is not necessarily the same. The elements in the vector denote which k-mer they represent.

### 2.3.4 K-mer Extraction and Selection

Having presented the issues relevant when implementing k-mer representation in the previous section, this section is dedicated to presenting some of the ways in which k-mer counting has been represented in previous works. Generally, all implementations of k-mer representation must reduce the number of k-mers to a level that is computationally feasible to work with. However, this reduction of input data can come at the price of decreased accuracy as potentially valuable information may be lost. The challenge is finding a solution that manages to identify and preserve the k-mers that are useful for classification, and discard those that are not.

Methods for selecting a subset of k-mers:

- The purpose of MinHash sketching is to create small, representative signatures for genomes (Berlin *et al.*, 2015). These signatures can be compared to determine the similarity between different genomes using techniques such as the Jaccard index. The first step in creating a signature, is creating hashes for each k-mer in the sequence using locality sensitive hashing (LSH). LSH is different from ordinary hashing in that it produces similar hash values for similar input data. The implication being that it can reasonably be assumed k-mers whose hash values are closer are more similar than k-mers where the difference between their hash values is greater. Then, a set of the n k-mers with the lowest hash value are extracted as a subset. This subset is called a sketch and serves as the genome signature. There are two

alternatives to implementing MinHash sketching. The first alternative is to select the elements with the lowest hash value in the whole sequence for the signature. The other alternative is to divide the sequence into subsets and take out the lowest hash value from each subset. The first alternative would result in signatures being more similar across genomes, while the second retains more information on the composition of the whole genome. There are several tools that rely on MinHash sketching. Some of these are Mash (Ondov *et al.*, 2016) and sourmash (Brown and Irber, 2016).

- HULK stands for Histosketching Using Little K-mers (Carrieri *et al.*, 2019). HULK reduces sequences to updatable histosketches of the k-mer spectrum. A histosketch is a data structure where a set of fixed size sketches are maintained over a streaming histogram to approximate similarity between histograms (Yang *et al.*, 2017). An important aspect of the histosketch is that it includes mechanisms for gradually forgetting older sketches when new histograms are added to the structure by giving less weight to sketches as more new sketches are added. The histosketch is then used as input for the machine learning models. HULK works by first converting the sequence into overlapping k-mers. These k-mers are then hashed uniformly over a set of bins. The frequency of each bin is then used as an approximation for the k-mer frequency. This way HULK represents the sequence as a fixed-length k-mer spectrum data structure. The placement of k-mers in each bin is determined by a consistent weighting sampling scheme and its resulting hash values. The resulting data structures can be considered fixed length feature vectors and can be used as input for the machine learning models.

Methods for reducing k-mer space by removing k-mers deemed less important:

- PhenotypeSeeker (Aun *et al.*, 2018)is a program that does three things; identifies phenotype-specific k-mers, generates k-mer based statistical models for predicting a given phenotype, and predicts the phenotype of given bacterial sequences. As such, PhenotypeSeeker is not a method of representing a sequence of k-mers, but an entire pipeline with a built-in prediction model. PhenotypeSeeker is described as memory efficient and easy to use. This model is not based on discovering k-mer frequency, but rather k-mer presence. However, a k-mer is only marked as present if it is found in the sequence a minimum of five times.
- De Brujin Graph Genome-Wide Association Studies (DBGWAS) is an extended k-mer based metagenomic method that was developed to discover genetic variants linked to distinct phenotypes (Jaillard *et al.*, 2018). In DBGWAS, important k-mers are discovered by

representing all k-mers across all genomes in the dataset as a single De Brujin graph. The graph can remove the redundancy of consecutive k-mers by combining them into one and provides a way of visualizing the context for significant k-mers. Each node is individually tested for association with a phenotype. For the nodes found to be significant, subgraphs are extracted which display their local genomic environment. The topology, metadata, and annotation of these subgraphs can then be used to locate important k-mers in the test data and categorize them as having the phenotype connected to the sub-graph in question.

# Chapter 3: Methods

## 3.1 Database

### 3.1.1 Overview of Database

The choice of database can have significant consequences for the results as it provides the data the model will be working on. The Genome Taxonomy Database (GTDB) (*Genome Taxonomy Database*, 2021) (Parks *et al.*, 2021) was selected for this project as it is quite extensive and contains many samples of whole genomes. GTDB is an initiative that aims to establish a standardized microbial taxonomy based on genome phylogeny (see section 2.1.4). There are several versions of this database. For this project we used the 2021 release of the database that can be downloaded from the official website (The GTDB Team, 2021). The database covers bacteria and archaea. GTDB includes data on marker genes, metagenomes, and metatranscriptomes. However, as the topic of the project is metagenomic classification, only the sub-folder 202.0/auxillary_files/ gtdbtk_r202_data.tar.gz of the database was used in this project, as it contains only whole genome data. Henceforth, when referring to the database only the aforementioned sub-folder of the GTDB database is included. In total the database contains 47 894 samples, of which 2 339 are archaea and 45 555 are bacteria.

While cleaning the database is important to get good data to work on, the database used for this project was of high quality and thus did not require any cleaning. The genomes in the database had a completeness between 50% and 100%, with the majority being closer to 100% than 50% (*Genome Taxonomy Database*, 2021). Almost all genomes also had a low level of contamination. Only eight genomes were considered low quality.

### 3.1.2 Splitting the Database

As discussed in section 2.2.2.8, splitting the dataset is important to avoiding unrealistically optimistic results. The database was therefore split into three datasets; a training set, a validation set, and a test set. It was determined that the set sizes should follow common practise, meaning 50% of the database was allocated as the training set, 30% as the validation set, and 20% as the testing set. As

the database is quite large it was not necessary to consider approaches such as cross-validation to use more of the database for training.

A Python class was written to select which files in the database would be used for training, validation, and testing. The class can be found under DivideDatabase/divide_database.py. The program worked as follows; Initially the whole database was allocated to training. The program read the file gtdb_taxonomy.tsv (see section 3.1.1) to find the names of all data files. Then the program calculated how many files should be in the validation set based on the length of the training set and randomly selected files from the training set until the calculated validation set quota had been filled. These files were then moved from the training set to the validation set. Then, the program calculated how many files should be in the test set based on the length of the training and validation sets combined. Files were then randomly selected until there were enough to fill the test set quota. These files were then moved from the training set to the test set. The result was a training set containing approximately 50% of all files in the dataset, a validation set containing approximately 30% of the dataset, and a test set containing approximately 20% of the dataset. The main logic of this program is shown in figure 3.1. The figure shows the part of the program where the length of the validation and test sets are calculated, and files are randomly selected until both sets contained the correct number of files. Initially, the whole dataset was allocated for the training set. Then, the length of the validation set was calculated using the following formula: $size\ of\ validation\ set = \frac{size\ of\ training\ set}{100} * 30$. Those files were then removed from the training set and added to a separate validation set. The length of the test set was then calculated using the following formula: $size\ of\ test\ set = \frac{size\ of\ training\ set + size\ of\ validation\ set}{100} * 20$, where the size of the training and validation sets combined was equal to the size of the whole dataset. The test files were then removed from the training set, resulting in three non-overlapping sets containing 50% (training), 30% (validation), and 20% (testing) of the whole dataset.

```
# Select random files for the validation set.
validation_set = []
size_of_validation_set = math.floor((len(training_set) / 100) * 30)
for i in range(0, size_of_validation_set):
    selected_file = random.randint(0, len(training_set))
    validation_set.append(training_set[selected_file]) #Add file to validation set.
    del training_set[selected_file]

# Select random files for the test set.
test_set = []
size_of_test_set = math.floor(((len(training_set) + len(validation_set)) / 100) * 20)
for i in range(0, size_of_test_set):
    selected_file = random.randint(0, len(training_set))
    test_set.append(training_set[selected_file]) #Add file to test set.
    del training_set[selected_file]
```

**FIGURE 3.1: CODE TO SPLIT THE DATASET INTO TRAINING, VALIDATION, AND TEST SETS.**

## 3.2 Tools

### 3.2.1 Jellyfish

Developing a program that can count k-mers efficiently and without using a lot of computational resources is a time-consuming task. Therefore, in this project the Jellyfish tool was utilized for this purpose instead of developing bespoke software. Jellyfish is a program that is specifically developed for fast and memory-efficient k-mer counting (gmarcais, 2021) (Marçais and Kingsford, 2011). Jellyfish is a command-line program that runs on FASTA-files containing DNA sequences. It provides commands to count k-mers with a series of options such as setting the k-mer length, only including high frequency k-mers etc. There are several releases of Jellyfish available. In this project we used version 2.3.0.

### 3.2.2 Sourmash

Sourmash is a tool to compute hash sketches from DNA sequences (see section 2.3.4). Sourmash allows the user to compute the hash sketches, as well as compare them to each other and produce a dendrogram of the results (Brown and Irber, 2016). The computational requirements are lower than many other similar programs which is useful when working on a large database. Sourmash is a command-line tool that runs on FASTA files containing DNA sequences. It is also a Python library. The

purpose of using sourmash is to not have to implement MinHash for those representation methods that rely on it (see section 3.6.1). In this project we used version 4.2.2 of sourmash.

### 3.2.3 Tensorflow

Tensorflow is a python library, and an interface for implementing and executing machine learning algorithms, including neural networks (Abadi *et al.*, 2015). It is a popular platform used by large tech-companies such as Google, AirBnB, Twitter etc. Keras is a deep learning API running on top of TensorFlow (Chollet, 2015). Keras provides functionality to build different machine learning models quickly due to the array of inbuilt features and algorithms, as well as functionality for testing and evaluating models. In this project all neural networks were build using Keras (see section 3.7), and the 2.6.0 version of TensorFlow was used.

## 3.3 Computational Resources

### 3.3.1 Saga

Due to the size, and subsequent computational requirements of running the project, programs related to preparing the dataset and running the machine learning algorithms were run on a supercomputer named Saga owned by UNINETT Sigma2 (Sigma2/NRIS, 2021). UNINETT Sigma2 is a company that provides high-performance computing (HPC) and large-scale storage to researchers in Norway (*Uninett Sigma2*, 2021). The technical specification of Saga is described in table 3.1.

TABLE 3.1: SAGA TECHNICAL SPECIFICATION

| Details | Saga |
|---|---|
| System | Hewlett Packard Enterprise – Apollo 2000/6500 Gen10 |
| Number of cores | 16 064 |
| Number of nodes | 364 |
| Number of GPUs | 32 |
| CPU type | Intel Xeon-Gold 6138 2.0 GHz / 6230 2.1 GHz<br>Intel Xeon-Gold 6130 2.1 GHz<br>Intel Xeon-Gold 6126 2.6 GHz |
| GPU type | NVIDIA P100, 16 GiB RAM |
| Total max floating point performance, double | 645 Teraflops/s(CPUs) + 150 Teraflop/s (GPUs) |
| Total memory | 97.5 TiB |
| Total NVMe+SSD local disc | 89TiB + 60 TiB |
| Total parallel filesystem capacity | 1 PB |
| Operation System | Linux |

### 3.3.2 Lenovo Laptop

The programs that were not executed on Saga, were run on a personal laptop. The specification of said laptop is described in figure 3.2.

TABLE 3.2: LENOVO LAPTOP TECHNICAL SPECIFICATION

| Details | Lenovo Laptop |
|---|---|
| Model | Lenovo YOGA 720-13IKB |
| Processor | Intel® Core™ i5-8250U CPU @ 1.60GHz 1.80GHz |
| RAM | 8GB |
| Operation System | Windows 10 |

## 3.4 Developed Software

The code developed for this project was written in Python. Python was selected due to offering pre-existing tools such as Tensorflow, that makes implementing artificial neural networks simpler. All the code for this project is available on GitHub at: https://github.com/uio-bmi/TaxClassUsingML. In some parts in this thesis there are descriptions of where particular code can be found. The descriptions

provided assume that one is looking at the venv/SourceCode folder in the project, not the root folder.

## 3.5 Selecting K-mer Length

Before finding k-mers in the database, it is necessary to select a k-mer length. K-mer length can impact both the accuracy and performance of our models (see section 2.3.2). Therefore, it is useful to know how we can expect k-mer length to impact factors such as how many k-mers are found in the genome, what k-mer frequencies are typical at that k-mer length etc, which are factors that play a role in prediction accuracy and performance.

To explore the importance of k-mer length on the genomes in the GTDB database, the Jellyfish program was used to count k-mers at different lengths and calculate some statistics (see section 3.2.1). The statistics that were calculated were the number of unique k-mers, distinct k-mers, total number of k-mers and highest k-mer frequency. Unique k-mers refers to the number of k-mers that appear only once in the genome. Distinct k-mers are all the different k-mers in the genome, meaning each k-mer is counted once even if there are multiple instances of the same k-mer. The total number of k-mers is all k-mers in the genome including those that are identical. Highest k-mer frequency is the frequency of the k-mer that appears most often in the genome. Since the correct direction of each k-mer was uncertain, these were counted canonically (see section 2.3.1).

Ten files were selected randomly from the training set. Table 3.6 contains a table with the name of the randomly selected files and their correct classification as stated in the document gtdb_taxonomy.tsv.

**TABLE 3.3: RANDOM FILES SELECTED FOR K-MER LENGTH IMPACT ANALYSIS**

| File name | Classification |
|---|---|
| GCA_000204585 | Nitrosarchaeum limnae |
| GCA_003250455 | UBA11579 sp003250455 |
| GCA_009693985 | CAIYRG01 sp009693985 |
| GCA_009694295 | SHVJ01 sp009694295 |
| GCA_013003245 | JABDJX01 sp013003245 |
| GCA_902520385 | GCA-2718035 sp902520385 |
| GCF_000243095 | Cupriavidus basilensis_D |
| GCF_003122485 | Eubacterium_I ramulus_A |
| GCF_007845205 | SDRK01 sp007845205 |
| GCF_904066215 | Microbacterium sp002456035 |

The k-mers in each file was counted using the below jellyfish command.

```
jellyfish count -m X -s 100M -C -t 10 -o kmerCounts.jf genomeFile.fna
```

The important parts of this command are as follows:

jellyfish: invoke jellyfish tool

count: command to count k-mers

-m X: X denotes the k-mer length and was run as $X \epsilon \{4, 6, 8, 12, 16, 32, 64\}$

-s 100M: use a hash with 100 million elements during counting

-t 10: use 10 parallel threads during execution

-C: count canonical k-mers (see section 2.3.1)

-o kmerCounts.jf: new jellyfish file which is created to store the results

genomeFile.fna: the file containing the genome that is being counted

This process was repeated for seven different k-mer lengths; 4, 6, 8, 12, 32, and 64. The below jellyfish command was then used to gather some statistics about the k-mers.

```
jellyfish stats kmerCounts.jf
```

The important parts of this command are as follows:

jellyfish: invoke jellyfish tool

stats: command to compute statistics about k-mers

kmerCounts.jf: jellyfish file with k-mer counts to compute statistics on

Figures 4.1 to 4.8 in chapter 4.1 displays the results for each k-mer length. The figures list the number of unique and distinct k-mers, as well as the total amount of k-mers and the highest k-mer frequency for each file at different k-mer lengths.

## 3.6 Generating Data Input

### 3.6.1 MinHash Sketches

MinHash signatures were explained in section 2.3.4. To recap, MinHash sketching creates small representative signatures for genomes that can be compared to determine similarity without comparing full genomes. The goal is to reduce data input size, by replacing long genomes with short signatures, while still retaining comparability. As this technique addresses two of the main problems that faced k-mer representation in this project, it was determined that MinHash sketching should be tested as an alternative for representing k-mers. Preparing MinHash signatures for training required two main steps, creating the signatures, and transforming them to a suitable format for the machine learning models.

#### 3.6.1.1 Creating Signatures Using Sourmash

Rather than implement MinHash sketching from scratch, the sourmash program (see section 3.2.2) was used to create the MinHash sketches. A virtual environment was created on Saga (see section 3.3.1) containing the sourmash program. A batch script was then written to loop through every file in the training set and use sourmash to create a signature, resulting in a set containing one sourmash signature for every file in the original training set. The signature was set to be based on 12-mers as sourmash signatures is a k-mer presence/absence-based method rather than k-mer counting. The length was set to be 3000, which is estimated to be roughly 0.1% of the total number of 12-mers in a genome (see section 5.1). It was determined that the lowest hash values should be chosen rather than selecting the lowest hash from subsets of the genome (see section 2.3.4). The reason being that the goal of this project was to classify genomes, not characterize them. Therefore, it does not matter if the representation of the genome is not balanced in terms of which parts of the genome are represented. Another reason is that since the space of possible k-mers is so large, ideally, we would like to find common k-mers that can be used to classify many different genomes. Selecting the lowest hashes regardless of position should logically result in a greater commonality in the selected k-mers. The exact command used to generate each signature is described below.

sourmash sketch dna -p num=3000, k=12 genomeFile -o signature.sig

The important parts of the command are:

sourmash: invoke sourmash tool

sketch: command to create MinHash sketches

dna: specifies the type of data in the file to make a sketch from

-p num=3000: specifies the size of the signature

k=12: specifies that the signature should be made from 12-mers

genomeFile: the file to make a signature from

-o signature.sig: the newly created sourmash file to store the signature

The command was run on Saga (see section 3.3.1) using the batch script in/RepresentationApproaches/HashSketch/CreateData/create_sourmash_sign.sh.

#### 3.6.1.2 Combine Signatures to Select K-mers

After completing section 3.6.1.1, we are left with a new training set consisting of signature files. Each signature contains 3000 of the smallest hashes computed by sourmash as well as some meta-data. Figure 3.2 contains an example of such a file. The file is the signature computed from file GCA_002204705.1_genomic.fna.gz. The contents have been shortened for display purposes. The important part of this file is the array called “mins”, which is the signature itself. The hash values within the signature are in increasing order. The signature is not yet comparable to other signatures as an element in one signature does not necessarily have any relation to an element at the same position in a different signature. Before the signatures can be used for machine learning, they must be made comparable by being made into presence/absence vectors where each position denotes a specific k-mer. This requires knowing the number of different k-mers. As the hash value for identical k-mers is always the same, and sourmash retained the lowest hash values for each signature, we can expect there to be some overlap in the k-mers selected for each signature. To know the required length to represent every k-mer in the signatures, it is necessary to know the number of distinct k-mers in total. Meaning, we need to know how many different k-mers are found in the whole set of signatures for all genomes.

```
[
 {"class":"sourmash_signature",
  "email":"",
  "hash_function":"0.murmur64",
  "filename":"./TrainingSet/GCA_002204705.1_genomic.fna.gz",
  "license":"CC0",
  "signatures":
   [
    {
     "num":3000,
     "ksize":12,
     "seed":42,
     "max_hash":0,
     "mins": [652945146928, ..., 35288263831051861],
     "md5sum":"62be95041741409c447bd141e3d456d4",
     "molecule":"dna"
    }
   ],
   "version":0.4
  }
]
```


**FIGURE 3.2: STYLIZED CONTENTS OF SIGNATURE FILE COMPUTED FROM GCA_002204705.1_GENOMIC.FNA.GZ.**

A python class was written to find the number of distinct k-mers. The class can be found in /RepresentationApproaches/PrepareModelInput/CombineSignatures/combine_signatures.py. The problem is solved by looping through every file in the set of signatures and adding the signature's hash values to a file. A hash value is only added if it is not already in the file. The result is thus a file containing the hash of every distinct k-mer in the signature set. This logic can also be summed up by the code in figure 3.3. The result of combining all signature hash values can be found in section 4.2.

The code was run on Saga (see section 3.3.1) using a python main class which can be found in /RepresentationApproaches/HashSketch/PrepareModelInput/CombineSignatures/main.py. The batch script combine_signatures.sh, which can be found in the same folder, was used to run the program.

```
# Method loops over all files and finds their signature, then adds the k-mers
# in the signature to the all_sign file.
@staticmethod
def loopOverSignatures():
    for file in os.listdir("./Signatures/"):
        print("Working on file " + file + "...")
        #Get signature from signature file
        content = open(os.path.join("./Signatures/", file)).read()[1:-1]
        content = json.loads(content)
        signature = content["signatures"][0]["mins"]
        #Add each k-mer in the signature to the all_sign file.
        for kmer in signature:
            CombineSignatures.__addKmer(str(kmer))
        os.rename("./Signatures/" + file, "./Finished/" + file)
        print("Finished with file " + file)
```

**FIGURE 3.3: THE CODE USED TO COMBINE THE HASH VALUES FROM MINHASH SIGNATURES.**

### 3.6.1.3 Transform Signatures

Section 3.6.1.2 resulted in a file containing every distinct hash value in the set of signatures. The highest hash values in the file were then removed leaving a file of 40 000 hash values. The purpose of this reduction was to reduce the length of the representation vectors to lower the computational requirements for running the program. The final step was then to use said file and the set of signatures to produce a set of representative and comparable vectors for each signature to be used for machine learning. This means transforming the individual signature in each file into a vector where each position is correlated to a distinct k-mer, and each element in the vector is $\epsilon \{0, 1\}$ where a 0 denotes the k-mer is not present in the signature and a 1 denotes that it is present. The required vector length to represent all distinct k-mers is the same as the number of distinct k-mers present in the file generated in section 3.6.1.2. K-mer presence should be represented by a 1, rather than hold the original hash value as experiments have shown machine learning models tend to perform better on numerical data that is scaled down to be 0 ≤ and ≤ 1 (see section 2.2.2.7).

To transform the signatures, a python class was written which takes a set of sourmash signature files and the file of all hash values generated in section 3.6.1.2 and transforms the signatures to vector representations ready for machine learning. This class can be found under RepresentationApproaches/MinHash/PrepareModelInput/signature_transformer.py. The first step was creating a list of all hash values from the combined hashes document. The order in the list was the same as in the original document, increasing numerical. The next step was going through each

signature in the training set and creating a presence/absence representation based on the contents of the signature and the list of all hash values. Creating presence/absence representations was done using the __createBinaryVector method, which takes a signature and creates a binary vector were each element in the vector represent one of the hashes, and the value $\epsilon \{0, 1\}$ to indicate whether that hash is present in the signature. This method can be seen in figure 3.3. The result was a list of binary vectors to replace the signatures.

```
# Method takes a signature and returns its binary vector.
def __createBinaryVector(self, signature):
    binaryVector = []
    for kmer in self.kmers:
        if int(kmer) in signature:
            binaryVector.append(1)
        else:
            binaryVector.append(0)
    return binaryVector
```

**FIGURE 3.4: CODE TO CREATE PRESENCE/ABSENCE VECTOR REPRESENTATION OF MINHASH SIGNATURES.**

The code was run on Saga (see section 3.3.1).

### 3.6.2 Random K-mers

Considering the length of the genomes in the database, and subsequent number of k-mers, it is not feasible to use every k-mer in the database as input. Moreover, methodically selecting a subset of k-mers is still challenging and requires substantial effort and computational resources. Therefore, it is reasonable to consider to what extent methodical k-mer selection is worthwhile. For this reason, in the method described in this section, the k-mers were selected arbitrarily. The goal was to determine whether more complex methods, such as MinHash sketching and discriminative k-mers are worth the extra time and computational resources. The method described in this section was named random k-mers because the k-mers to be counted were selected randomly with no input from the database. This included generating a list of arbitrary k-mers, and then look for only those k-mers in every genome in the database. The random k-mers method was used to make a comparison between k-

mer counting and presence/absence as the implementation of both is very similar. This similarity reduces the number of factors which impact the result and therefore allows for more direct comparison between the two representation methods.

### 3.6.2.1 Generate Random K-mers

To create arbitrary k-mers, a python class was written. The class found under RepresentationApproaches/RandomKmers/CreateData/random_kmers.py. The class contains the method generateRandomKmers which generates a list of random k-mers of given length using the __generateRandomKmer method. Once enough k-mers have been generated all the k-mers are added to a FASTA file named random_kmers.fa. A k-mer and its complement are treated as the same k-mer since we will count canonical k-mer in the next step (see section 3.6.2.1). The method __generateRandomKmer is displayed in figure 3.5. __generateRandomKmer takes the desired k-mer length and then randomly chooses between the four nucleotides adding them together, until we have produced a complete k-mer.

```
# Method generates a random k-mer of given length.
@staticmethod
def __generateRandomKmer(kmerLength):
    randomKmer = ""
    nucleotides = ["A", "C", "T", "G"]
    for index in range(0, kmerLength):
        randomNucleotide = nucleotides[random.randint(0, 3)]
        randomKmer = randomKmer + randomNucleotide
    return randomKmer
```

**FIGURE 3.5: CODE TO CREATE A RANDOM K-MER.**

When going through the process of generating random k-mers, it is possible to generate identical k-mers. In this case the copy k-mer should not be included as this would result in fewer k-mers than what has been determined. To avoid this problem, the program continuously generates random k-mers until it has filled the given quota for random k-mers. This logic is shown in figure 3.6 showing how a list of k-mers is generated and then written to a FASTA file.

```
# Method generates a list of randomly generated kmers.
@staticmethod
def generateRandomKmers(numberOfKmers, kmerLength):
    randomKmers = []
    while len(randomKmers) < numberOfKmers:
        randomKmer = KmerGenerator.__generateRandomKmer(kmerLength)
        if randomKmer not in randomKmers or randomKmer[::-1] not in randomKmers:
            randomKmers.append(randomKmer)
    file = open("random_kmers.fa", "a")
    for kmer in randomKmers:
        file.write(">\n" + kmer + "\n")
    file.close()
    return randomKmers
```

**FIGURE 3.6: METHOD GENERATERANDOMKMERS THAT GENERATES A LIST OF RANDOM K-MERS AND WRITES THEM TO A FASTA FILE.**

The code was run on the Lenovo laptop (see section 3.3.2) using a python main class which can be found in RepresentationApproaches/RandomKmers/CreateData/main.py. The method generateRandomKmers takes two parameters; number of k-mers to generate and k-mer length. The program was run once with the number of k-mers set to 40 000 to get vectors of the same length as the MinHash vectors (see section 3.6.1) and the k-mer length to 8 for the k-mer counting representation, and once with k-mer length set to 12 for the k-mer presence/absence representation (see section 5.1).

#### 3.6.2.2 Count Generated K-mers

Once the file of randomly generated k-mers was completed, the next step was counting all those k-mers that match one of the generated k-mers. This was done using jellyfish (see section 3.2.1). For each file in the training set, the subset of k-mers were counted using a jellyfish command of the below format. This was done once with 8-mers and once with 12-mers.

jellyfish count -m X -s 100M -C -t 10 -o result.jf –if randoms.fa fileToCount.fna

The important parts of this command are as follows:

jellyfish: invoke jellyfish tool

count: command to count k-mers

-m X: sets the k-mer length, command run with x $\epsilon$ {8, 12}

-C: count canonical k-mers

-o result.jf: new jellyfish file which is created to store the results

--if randoms.fa: the randomly generated k-mers to look for

fileToCount.fna: the file being counted

The result was that each file in the training set was replaced with a jellyfish file containing counts for all the randomly generated k-mers. However, as the results were saved in jellyfish files, they were not readable by other programs. The files were therefore exported to FASTA files using the below jellyfish command.

```
jellyfish dump previousResult.jf > readableResult.fa
```

The important parts of the command are as follows:

jellyfish: invoke jellyfish tool

dump: command to export results to different format

previousResult.jf: jellyfish file with unreadable results

readableResult.fa: new FASTA file with readable results

Figure 3.7 shows an example of what the readableResult.fa file might look like. Each k-mer entry contains one line with the k-mer frequency, identified by the symbol ">", and the k-mer itself underneath.

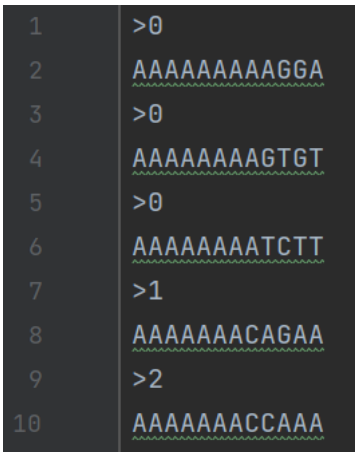


**FIGURE 3.7: EXAMPLE OF WHAT A K-MER COUNT FILE LOOKS LIKE.**

Running jellyfish to count the random k-mers was executed on Saga (see section 3.3.1) using a batch script. The batch script can be found under RepresentationApproaches/ RandomKmers/ CreateData/count_subset_kmers.sh.

### 3.6.2.3 Transform Input

The result of step one was a database of FASTA files containing the desired k-mer counts. However, the machine learning models cannot take FASTA files as input. Their contents must instead be transformed. For this purpose, a new python class was written. The class can be found under /RepresentationApproaches/RandomKmers/ PrepareModelInput/ count_transformer.py. The program goes through every FASTA file in a directory. For each file, it reads the k-mer counts and creates a vector where each position in the vector represents one of the random k-mers, and the value at that position the count for that particular k-mer. After that, the program diverges for the count and presence/absence-based methods.

For the k-mer count method, the counts had to be scaled correctly. Machine learning models tend to benefit from their numerical input being scaled 0 ≤ and ≤ 1 (see section 2.2.2.7). To achieve this, each element in the vector was re-calculated using the below formula (Brownlee, 2019b). This was implemented in the method __scaleCountVector, shown in figure 3.8. Min and max in the formula refer to the greatest and lowest value in the vector.

$$scaled\ value = \frac{unscaled\ value - \min(x)}{\max(x) - \min(x)}$$

```
# Method scales a k-mer count vector so that all values are between 0 and 1.
def __scaleCountVector(self, vector):
    scaled_vector = []
    max_value = max(vector)
    min_value = min(vector)
    scaled_value = 0
    for value in vector:
        if max_value - min_value != 0:
            scaled_value = (value - min_value) / (max_value - min_value)
        scaled_vector.append(scaled_value)
    return scaled_vector
```

**FIGURE 3.8: CODE THAT SCALES K-MER FREQUENCIES IN A VECTOR TO BE BETWEEN 0 ≤ AND ≤ 1.**

For the k-mer presence/absence method, the counts had to be made binary. The method scalePresenceVector takes a vector and alters any value > 0 to a 1, resulting in a vector where every value is $\epsilon\ \{0, 1\}$. This method is shown in figure 3.9.

```
# Method takes k-mer count vector and returns a binary vector denoting only
# presence/absence.
def __scalePresenceVector(self, vector):
    binary_vector = []
    for value in vector:
        if value > 0:
            binary_vector.append(1)
        else:
            binary_vector.append(0)
    return binary_vector
```

FIGURE 3.9: CODE THAT MAKES A REPRESENTATION VECTOR BINARY.

The code was run on Saga (see section 3.3.1).

### 3.6.3 Discriminative K-mers

MinHash sketching and random k-mers both reduce the input space by selecting k-mers in a random or semi-random way. The discriminative k-mers method aims to look closer at each genome individually to identify important k-mers. This method was inspired by CLARK, which discovers important k-mers by discovering all k-mers in the database, and then removing any common k-mers leaving only a subset containing discriminative k-mers unique to a single genome (see section 2.1.5.4.1). In this section, we attempted to do the same. As the goal was to find unique k-mers, this was a presence/absence-based method as k-mer frequency is irrelevant.

#### 3.6.3.1 Counting K-mers for Each Genome

To compare and discover unique k-mers, the first step was getting all the k-mers. This step was done by using jellyfish to count the k-mers of each file in the training set. The actual k-mer frequencies produced by jellyfish are not relevant for discovering discriminative k-mers. However, using this approach means getting a list of every k-mer in the genome without having to write any code. The jellyfish command used is described below. The k-mer length was set to 12 as this is a presence/absence-based method (see section 5.1).

```
jellyfish count -m 12 -s 100M -C -t 10 -o result.jf fileToCount.fna
```

The important parts of this command are as follows:

jellyfish: invoke jellyfish tool

count: command to count k-mers

-m 12: sets the k-mer length to 12

-s 100M: use a hash with 100 million elements during counting

-C: count canonical k-mers

-t 10: use 10 parallel threads during execution

-o result.jf: new jellyfish file which is created to store the results

fileToCount.fna: the file being counted

The result was that each file in the training set was replaced by a jellyfish file containing counts for all k-mers. However, as the results were saved in jellyfish files, they were not readable by other programs. The files were therefore exported to FASTA files using the same jellyfish dump command as in 3.6.2.2.

Running jellyfish to count all k-mers was executed on Saga (see section 3.3.1) using a batch script. The batch script can be found under /RepresentationApproaches/UniqueKmers/ CreateData/jellyfishing.sh.

#### 3.6.3.2 Finding Discriminative K-mers

The previous step resulted in a training set of files containing k-mer frequencies for all k-mers in the genome. The next step was comparing these files to find k-mers that are unique to each genome. A python class was written for this purpose. The class can be found under RepresentationApproaches/ UniqueKmers/CreateData/unique_kmer_selector.py. The program compares all files in a directory two by two and removes any common k-mer between them. Once the program is finished, every file in the directory contains only k-mers that are unique to said file.

```
# Method goes through every file in the Counts folder and removes common kmers.
def stripFiles(self):
    files = self.__getFileNames("Counts")

    #For each file in the folder...
    for i in range(len(files)):
        print("Working on file " + i + files[i])
        starter_file = files[i]
        comparison_pointer = i + 1 #Points to file we are comparing starter_file to

        #Compare every file to the starter file, finding common kmers
        while comparison_pointer < len(files):
            comparison_file = files[comparison_pointer]
            self.__findUselessKmersInFile(starter_file)
            self.__findUselessKmersInFile(comparison_file)
            self.unique_kmers.clear()
            self.__removeUselessKmers(starter_file)
            removal_pointer = comparison_pointer

            if len(self.useless_kmers) > 0:
                #Remove common kmers with starter file from every other file.
                while removal_pointer < len(files):
                    self.__removeUselessKmers(files[removal_pointer])
                    removal_pointer = removal_pointer + 1

            self.unique_kmers.clear()
            comparison_pointer = comparison_pointer + 1

        os.rename(starter_file, "./Finished/" + starter_file.replace("./Counts/", ""))
```

**FIGURE 3.10: METHOD STRIPFILES THAT GOES THROUGH EVERY FILE IN A DIRECTORY AND REMOVES ANY COMMON K-MERS.**

The main logic was implemented in the method stripFiles, which can be seen in figure 3.10. This method loops through every file in the directory and compares it to every other file in the directory. The comparison is organized in such a way that the method compares one file to all other files, then compares the next file to all files except the first one etc. An example of this pattern is displayed in figure 3.11. The number of comparisons made during the program can thus be calculated using the following formula: $comparisons = \frac{n^2-n}{2}$, where n is the number of files in the directory. During comparison, the common k-mers are removed in the same way. Common k-mers between the first and second files are removed from all files, then common k-mers between the first and third files are removed from all files etc. When comparing, a k-mer and its complement are considered the same k-mer.

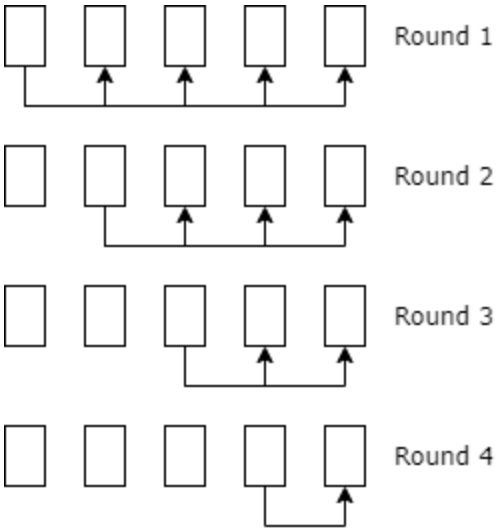


**FIGURE 3.11: ILLUSTRATION OF HOW FILES ARE COMPARED TO FIND DISCRIMINATIVE K-MERS.**

The program to find discriminative k-mers was run on Saga using the batch script RepresentationApproaches/UniqueKmers/CreateData/find_unique_kmers.sh (see section 3.3.1). The training set was divided into subsets of roughly 300 files. The discriminative program was then run on each subset. Each subset was then merged with another subset, and the program rerun. This process continued until the program had been run on the entire re-merged training set.

#### 3.6.3.3 Transforming Input

The previous step resulted in the files in the training set being drastically reduced in size. The next step was combining these files to create a standard for the input vectors. The remaining k-mers were all added to the file unique_kmers.fna using the code found in RepresentationApproaches/ UniqueKmers/PrepareModelInput/CombineKmers. Once the file was ready, the next step was using this file to create presence/absence vectors. This was accomplished using an adapted version of the code used to transform MinHash signatures into presence/absence vectors in section 3.6.1.3. This code can be found under RepresentationApproaches/UniqueKmers/PrepareModelInput/ TransformKmers.

### 3.6.3 Classification Labels

When training the machine learning models on the inputs generated in sections 3.6.1 – 3.6.3 the models need to know the correct classification for each data sample in the training set. The correct

data label for each file is found in the file gtdb_taxonomy.tsv. This file contains a table with every file in the database and its corresponding classification. The classification labels are of the below format.

*d_Bacteira;p_Firimutes_A;c_Clostridia;o_Peptostreptococcales;f_Peptostreptococcaceae;g_Romboutsia;s_Romboutsia hominis*

As can be seen in the above label, the classification labels include the classification of a genome from species to domain level. By extracting different parts of the label, we can thus classify an organism at any level in the taxonomic tree (see section 2.1.2). Ideally, we would like to classify at a low level, such as species, rather than a high level, such as domain, as this would be more precise. Therefore, in this project we will aim to classify at species and genus level. While classifying at species level would be more precise than genus level, it is also harder. This is because the lower the label is on the taxonomic tree, the more subtle the differences between genomes are likely to be. Our models might therefore struggle to accurately classify at the species level. Classifying on genus level will provide information on the models' ability to classify, while species level provides information on how sensitive the models are.

The classification labels should not be used as they are for machine learning since they are strings, and models tend to perform better on numerical data (see section 2.2.2.7). The classification labels were thus transformed using one-hot encoding. As discussed in section 2.2.2.6, one-hot encoding transforms categorical data into vectors containing columns of ones and zeroes denoting true or false for each category. One-hot encoding does not imply an ordering between classes and is therefore suitable for our database as there is not a logical way in which to numerically order all the genomes.

To encode the categorical data a python class was written. The class can be found under RepresentationApproaches/label_maker.py. Firstly, the program finds the correct classification for all training files from the document gtdb_taxonomy.tsv, using the method getSpeciesDictionary. This method results in a dictionary that can be used to look up the correct classification of each file. At this stage, one must decide if the classification should be at genus or species level as this method will alter the classification labels if classification should be at genus level by manipulating the classification string. This method is shown in figure 3.12. Next, a list of all unique classes in the training set is made using. A one-hot encoding vector is created for each file using the method getOneHotEncoding, shown in figure 3.13. The result is a list of binary vectors to replace the classification labels.

The method described in the previous paragraph was used to encode the classification labels of the training set regardless of how the genomes themselves were represented. Sections 3.6.1 – 3.6.3 describe in detail three different methods for representing genomes. All three methods used the same labels transformed in the same way as described in this section.

```
# Method takes a label and list of all classes and returns a one-hot-encoding
# vector of the label.
@staticmethod
def getOneHotEncoding(label, classes):
    vector = []
    for elem in classes:
        if label == elem:
            vector.append(1)
        else:
            vector.append(0)
    return vector
```

FIGURE 3.12: CODE THAT ONE-HOT ENCODES A CLASSIFICATION LABEL.

```
# Method creates a dictionary with all the file names and corresponding species labels
# from the gtdb_taxonomy.tsv file.
@staticmethod
def getSpeciesDictionary(useSpecies):
    labels = {}
    labels_file = open("../gtdb_taxonomy.tsv", encoding="utf8")
    label_reader = csv.reader(labels_file, delimiter="\t", quotechar='"')
    for row in label_reader:
        if useSpecies:
            label = row[1]
        else:
            label = row[1].split(";s__", 1)[0]
        labels[row[0].replace("GB_", "").replace("RS_", "")] = label
    return labels
```

FIGURE 3.13: METHOD GETSPECIESDICTIONARY THAT SPLITS THE CLASSIFICATION STRING DEPENDING ON TAXONOMIC LEVEL AND CREATES A DICTIONARY OF ALL CLASSES IN THE DATASET.

## 3.7 Developing Neural Networks

The relation between classes of microorganisms depends on their evolutionary history. Adding precise labels to such data is challenging both because of the complexity of such relations, and the inherent fluidity of gene pools as evolution is an ever-present process. Especially for prokaryotes that can reproduce both through mitosis and alter their hereditary material through horizontal gene transfer (see section 2.1.2). As presented in section 2.2.4.1, some promising research has been conducted on using neural networks for classification. Neural networks also have some tolerance for noisy data and can identify complex relationships. As such, it is a suitable method for taxonomic classification of microbial genomes. The domain is highly complex, and the instability of prokaryote gene pools requires a method that can perform well on data that might include significant amounts of intra-class variation. As neural networks are less vulnerable to noise, due to being better at generalizing, it logically follows that they should also be able to generalize well over highly varied data.

### 3.7.1 Selecting Algorithms

Sections 2.2.4.3.3 - 2.2.4.5 explain activation, loss, and optimization algorithms. Each of the three algorithm categories can have a significant impact on the performance of the neural network and how efficient it is to train.

The rectified linear unit (ReLU) algorithm was used for the activation function for all layers in the neural networks except the output layers. ReLU can learn complex data while also not being vulnerable to the vanishing gradient problem (see section 2.2.4.3.3). As such, ReLU provides the benefit associated with non-linear activation algorithms without the downside.

The softmax activation function was used for the output layer for each network. Softmax normalizes the input vector, resulting in a vector containing mutually exclusive probability scores for each class (see section 2.2.4.3.3). The benefit of using this activation algorithm is that it makes the final output more easily interpretable. The probability score provides more information than just a class prediction. Comparing the predicted class probability to the probabilities of other classes can be an indication of how confident the model is in its prediction.

Categorical cross-entropy was used for the loss function. This loss function has shown to perform well on multi-class classification tasks and make the model learn faster than most other loss functions (see section 2.2.4.3.3).

Stochastic gradient descent was used for the optimization algorithm. Important benefits of this algorithm are that it uses less memory and computes faster than a lot of other algorithms. Considering the size of the dataset used for this project (see section 3.1.1), this is not an insignificant factor.

## 3.7.2 Building Neural Networks

### 3.7.2.1 Standard Neural Network

Neural networks are explained in detail in section 2.2.4. To recap, a neural network is a machine learning algorithm based on the neural networks of the brain. A neural network contains a set of nodes ordered into layers. The network can be trained to predict outcomes on data by learning the correct weight to add to different inputs. Neural networks can be expensive to train, both in terms of time and computational resources such as memory. Therefore, it would be ideal to achieve excellent prediction accuracy with a small network as these are less expensive to train.

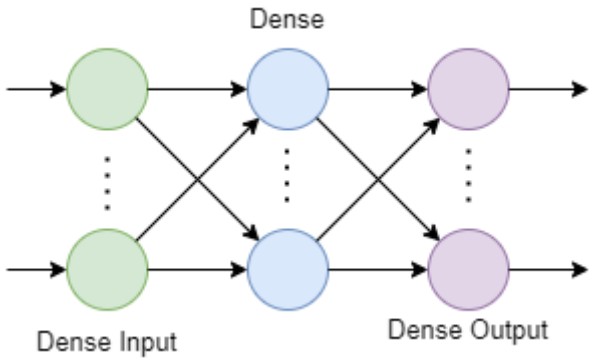


FIGURE 3.14: STANDARD NEURAL NETWORK MODEL STRUCTURE.

Keras was used to build a small neural network (see section 3.2.3). The implementation of the neural network can be seen in figure 3.14. This model consists of three layers, the input layer, one hidden layer, and the output layer. At this size, the network could technically be considered a minimal multilayer perceptron (see section 2.2.5.2.1) but will be referred to as a standard neural network to

distinguish it from deeper neural networks discussed in later sections. The layers of the network are dense layers. A dense layer is a fully connected layer where each node in one layer is connected to all nodes in the next. Dense layers are the most common type of layer in a neural network. The input and hidden layers have the same number of nodes as the length of the training set input vectors, while the output layer has one node for each class in the training set. An illustration of this structure can be seen in figure 3.15.

```
# Method returns a small, standard neural network.
def getStandardNeuralNetwork(self):
    return Sequential([
            layers.Dense(self.features, input_shape=(self.features,), activation='relu'),
            layers.Dense(self.features, activation='relu'),
            layers.Dense(self.classes, activation='softmax')
    ])
```

FIGURE 3.15: CODE IMPLEMENTING THE STANDARD NEURAL NETWORK.

### 3.7.2.2 Multilayer Perceptron Network

Multilayer perceptron networks are deep neural networks with at least one hidden layer (see section 2.2.5.2.1). A network that contains more nodes and layers is capable of learning more complex features from the data compared to a simpler network. In this project, the input data is very large and the relation between data samples comes from their evolutionary history and arbitrary individual differences. This makes the input complex and thus suitable for deep learning. How deep the networks should be however, is difficult to determine.

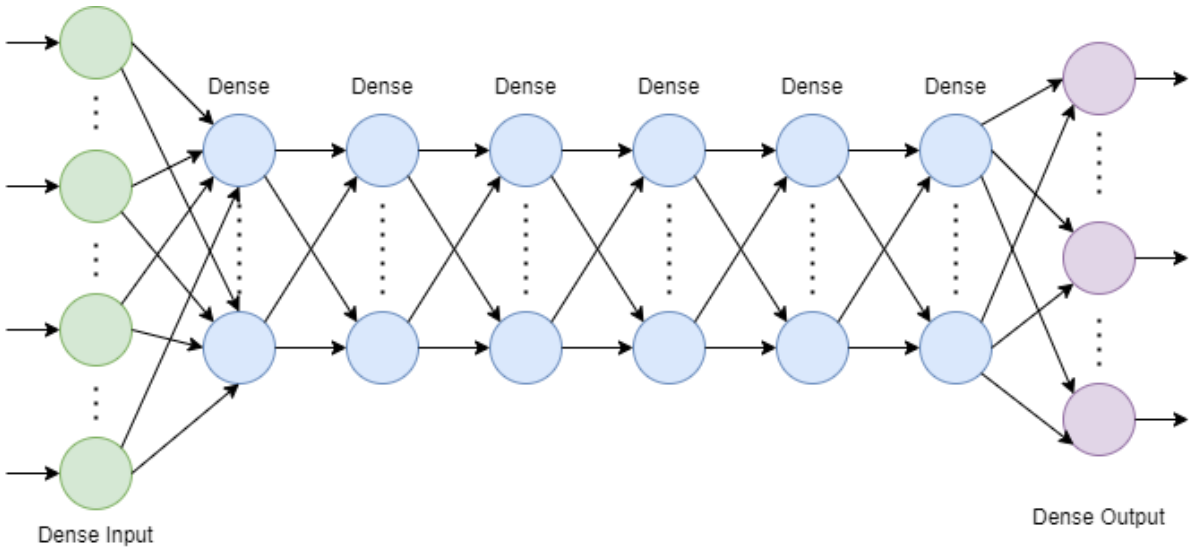


FIGURE 3.16: MULTILAYER PERCEPTRON NEURAL NETWORK MODEL STRUCTURE.

The multilayer perceptron network contains eight layers in total, an input layer, output layer, and six hidden layers. All the layers are dense layers. The number of nodes in the input layer is the same as the length of the input vectors (40 000), while the number of nodes in each hidden layer is half the length of the input vectors (20 000). Finally, the output layer has the same number of nodes as there are classes in the dataset. Figure 3.17 shows the implementation of the network, while figure 3.16 shows the structure.

```
# Method returns a multilayer neural network.
def getPerceptronNeuralNetwork(self):
    return Sequential([
        layers.Dense(self.features, input_shape=(self.features,), activation='relu'),
        layers.Dense(self.features/2, activation='relu'),
        layers.Dense(self.features/2, activation='relu'),
        layers.Dense(self.features/2, activation='relu'),
        layers.Dense(self.features/2, activation='relu'),
        layers.Dense(self.features/2, activation='relu'),
        layers.Dense(self.features/2, activation='relu'),
        layers.Dense(self.classes, activation="softmax")
    ])
```

**FIGURE 3.17: CODE IMPLEMENTATION OF MULTILAYER PERCEPTRON NEURAL NETWORK.**

### 3.7.2.3 Convolutional Neural Network

Convolutional neural networks are neural networks that include convolutional layers (see section 2.2.5.2.2). Convolutional layers are known for their ability to extract high level features that standard neural networks miss. While the technique has mostly been utilized for image recognition, some promising results exist for its use on sequential data. Thus, it was determined that this would be a suitable network type to test for taxonomic classification of microorganisms.

The convolutional neural network developed for this project contains nine layers; four convolutional layers, two dropout layers, a flatten layer, and two dense layers. The order of the layers are as follows; two convolutional layers, dropout layer, two convolutional layers, dropout layer, flatten layer, two dense layers. This structure can be seen in figure 3.18. The convolutional layers all contained five filters and a kernel size of 10. Kernel size is the size of the windows that are multiplied by the filters (see section 2.2.5.2.2). The stride was set to 1 and the padding type is ‘same’. Padding type ‘same’ means that padding will be added to the input to get a feature map of the same size as the input. The dropout layers were added to reduce overfitting. Dropout layers randomly select some of the nodes from their previous layers to ignore, resulting in information being lost. The dropout

layers were given a dropout rate of 50%, meaning each node had a 50% chance of being dropped rather than having its output passed to the next layer. Dense layers require that the input be one-dimensional. To ensure this was still the case after convolutions, the flatten layer was added in after the convolutional layers. The flatten layer concatenates the output to a one-dimensional structure and thus ensures there are no issues when passing on data to the dense layers. The implementation of the neural network can be seen in figure 3.19.

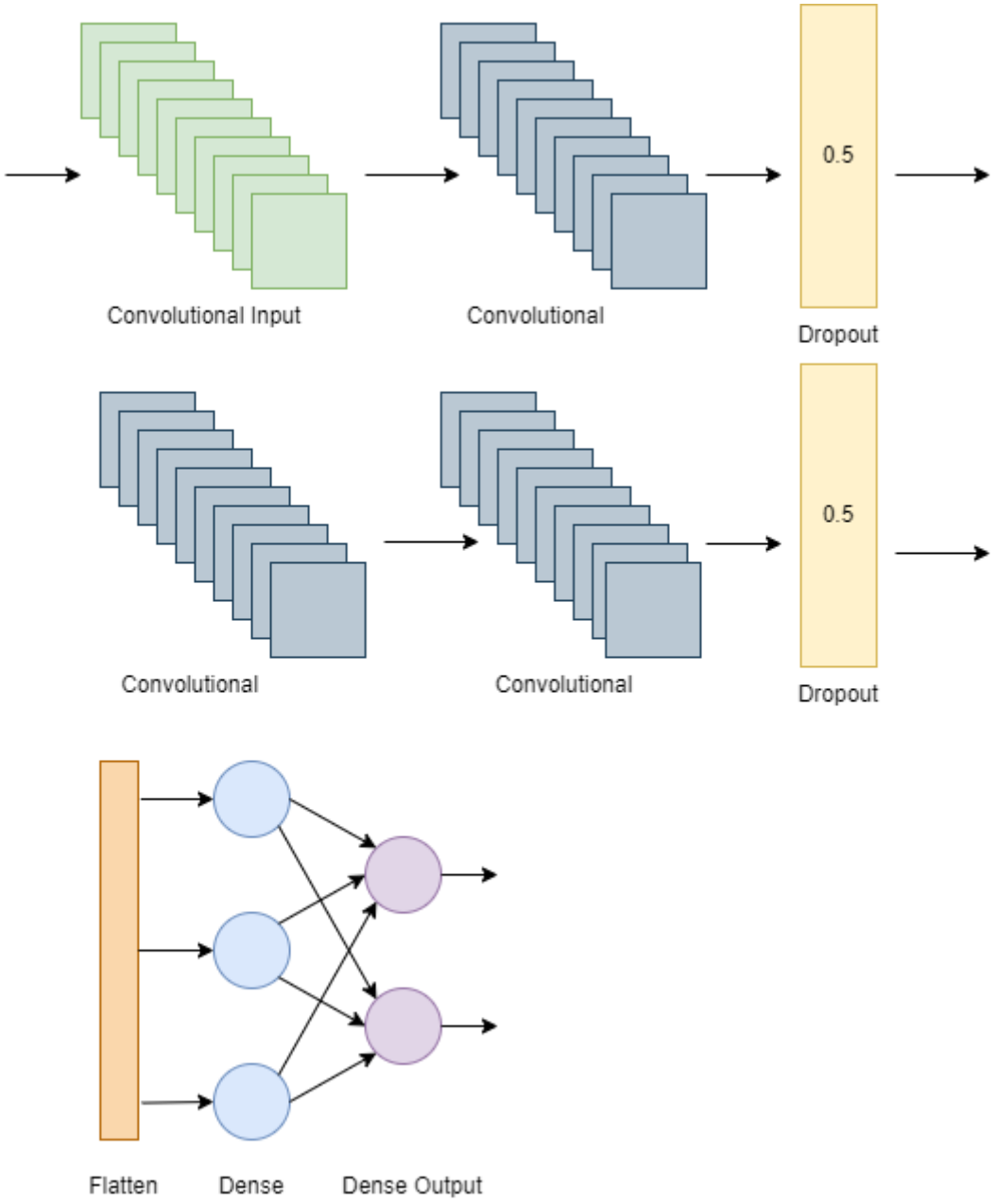


FIGURE 3.18: CONVOLUTIONAL NEURAL NETWORK MODEL STRUCTURE.

```
# Method returns a convolutional neural network.
def getConvolutionalNeuralNetwork(self):
    return tf.keras.Sequential([
        layers.Conv1D(filters=5, strides=1, activation="relu", kernel_size=10, padding='same'),
        layers.Conv1D(filters=5, strides=1, activation="relu", kernel_size=10, padding='same'),
        layers.Dropout(0.5),
        layers.Conv1D(filters=5, strides=1, activation="relu", kernel_size=10, padding='same'),
        layers.Conv1D(filters=5, strides=1, activation="relu", kernel_size=10, padding='same'),
        layers.Dropout(0.5),
        layers.Flatten(),
        layers.Dense(self.features, activation='relu'),
        layers.Dense(self.classes, activation="softmax")
    ])
```

**FIGURE 3.19: CODE IMPLEMENTING CONVOLUTIONAL NEURAL NETWORK.**

### 3.7.4 Training the Neural Networks

The neural networks were all trained on Saga (see section 3.3.1). To make the desired comparisons, eight copies were made of each of the three neural networks, resulting in a total of 24 models. Each model was trained on data produced using one of the representation methods explained in section 3.6, either at genus or species level. Tables 3.4 to 3.6 display the names of each model and what representation method and classification level they were trained on.

**TABLE 3.4: LIST OF STANDARD NEURAL NETWORK MODELS.**

| Standard Neural Networks | Genus | Species |
|---|---|---|
| MinHash | SMHG | SMHS |
| Random 8mers | SR8G | SR8S |
| Random 12mers | SR12G | SR12S |
| Discriminative k-mers | SDIG | SDIS |

**TABLE 3.5: LIST OF MULTILAYER PERCEPTRON NEURAL NETWORK MODELS.**

| Multilayer Perceptron Networks | Genus | Species |
|---|---|---|
| MinHash | MMHG | MMHS |
| Random 8mers | MR8G | MR8S |
| Random 12mers | MR12G | MR12S |
| Discriminative k-mers | MDIG | MDIS |

TABLE 3.6: LIST OF CONVOLUTIONAL NEURAL NETWORK MODELS.

| Convolutional Neural Networks | Genus | Species |
|---|---|---|
| MinHash | CMHG | CMHS |
| Random 8mers | CR8G | CR8S |
| Random 12mers | CR12G | CR12S |
| Discriminative k-mers | CDIG | CDIS |

All models were trained with only 10 epochs to reduce overfitting, and a learning rate of 0.1 (see section 2.2.4.2). They were trained on a training set and validated during training using a validation set. Before training, the samples in the training set were shuffled so that samples were passed to the model in a different order then they were in the training set. The code in figure 3.20 shows the method used to train each network.

```
# Method trains a model
def trainModel(model, name):
    # Build model
    model.compile(optimizer=tf.keras.optimizers.SGD(learning_rate=0.1),
                  loss=tf.keras.losses.CategoricalCrossentropy(from_logits=False),
                  metrics=['accuracy', tf.keras.metrics.Precision()]
                  )
    # Train model
    model.fit(training_set, training_labels, epochs=10, shuffle=True,
              validation_data=(validation_set, validation_labels), verbose=2)
    model.save("Models/" + name + ".h5")
```

FIGURE 3.20: CODE USED TO TRAIN NEURAL NETWORKS.

## 3.7.5 Comparing the Neural Networks

When comparing neural networks, it is necessary to define some metric that can be applied to all networks equally. For our networks, we determined to measure them using accuracy and precision (see section 2.2.3). Accuracy is the rate of correct classification. Precision is the rate of positive classifications that are correct. A positive classification means the model has determined that the probability of the sample belonging to a class is above a given threshold. Precision can be considered a measure of the model's confidence for all positive classifications, while accuracy is a measure of the model's classification ability overall.

We used Tensorflow's inbuilt functions to measure the accuracy and precision on all our models (see section 3.2.3). The metrics returned by these functions are generalizations over all classes the model works on and can thus be used to summarize the performance of the model in general. We elected to use the default 50% as the threshold for positive classification. This means that for a sample to be classified as a member of a certain class, the probability must be ≥ 50%.

# Chapter 4: Results

## 4.1 K-mer Length Experiments

When using DNA strands as input for machine learning models, splitting them into k-mers is useful as it lowers the required computational resources and need for genome completeness. K-mer length can have significant impact on the result when training a model to perform a classification task.
As described in chapter 3.5, some exploration was performed on a randomly selected set of genome files from the RTGB database to examine the impact of k-mer length. Ten genome files were randomly selected from the database. The tool jellyfish was used to count k-mers and discover the number of unique k-mers, distinct k-mers, total number of k-mers, and k-mer max-count. A unique k-mer is one that appears in the genome only once. Distinct k-mers are all k-mers where repeated k-mers are counted once, total k-mer are all k-mers in the genome including repeated k-mers, and max count is the highest k-mer frequency found in the genome. The results are presented in tables 4.1 – 4.8. Tables 4.1 – 4.7 contains a row for all ten files from the experiment run with the same k-mer length, while table 4.8 contains the average between the ten files for each k-mer length.

The meaning of the column is as follows (see section 3.5):

Unique: number of k-mers occurring exactly once

Distinct: number of different k-mers, counting repeated k-mers once

Total: the number of k-mers in the genome including repeated k-mers

Max count: the frequency of the most frequent k-mer

**TABLE 4.1: RESULTS OF K-MER LENGTH IMPACT ANALYSIS ON 4-MERS.**

| Species | Unique | Distinct | Total | Max count |
|---|---|---|---|---|
| Nitrosarchaeum limnae | 0 | 136 | 1 743 007 | 76 165 |
| UBA11579 sp003250455 | 0 | 136 | 3 120 377 | 54 583 |
| CAIYRG01 sp009693985 | 0 | 136 | 2010864 | 69 511 |
| SHVJ01 sp009694295 | 0 | 136 | 2 101 272 | 62 550 |
| JABDJX01 sp013003245 | 0 | 136 | 3 427 308 | 67 828 |
| GCA-2718035 sp902520385 | 0 | 136 | 778 678 | 42 891 |
| Cupriavidus basilensis_D | 0 | 136 | 8 544 947 | 294 399 |
| Eubacterium_I ramulus_A | 0 | 136 | 3 486 817 | 97 982 |
| SDRK01 sp007845205 | 0 | 136 | 523 454 | 19 781 |
| Microbacterium sp002456035 | 0 | 136 | 3 457 087 | 126 481 |

In table 4.1 we see that at k-mer length 4 there were no unique k-mers in any of the genomes. The k-mer frequencies were high. All genomes contained 136 distinct k-mers. The space of possible canonical k-mers can be calculated using the following formula: $k - mer\ space = \frac{4^k + 4^{\frac{k}{2}}}{2}$, where k is k-mer length. For 4-mers this results in a possible k-mer space of 136, which is the same as the number of distinct k-mers we found in table 4.2. This means all possible 4-mers are present in the genome.

**TABLE 4.2: RESULTS OF K-MER LENGTH IMPACT ANALYSIS ON 6-MERS.**

| Species | Unique | Distinct | Total | Max count |
|---|---|---|---|---|
| Nitrosarchaeum limnae | 0 | 2 080 | 1 742 855 | 11 297 |
| UBA11579 sp003250455 | 0 | 2 080 | 3 119 205 | 6 506 |
| CAIYRG01 sp009693985 | 0 | 2 080 | 2 010 728 | 11 608 |
| SHVJ01 sp009694295 | 4 | 2 076 | 2 101 062 | 9 168 |
| JABDJX01 sp013003245 | 0 | 2 080 | 3 426 704 | 8 477 |
| GCA-2718035 sp902520385 | 0 | 2 079 | 778 544 | 8 605 |
| Cupriavidus basilensis_D | 0 | 2 080 | 8 544 105 | 41 598 |
| Eubacterium_I ramulus_A | 0 | 2 080 | 3 486 271 | 11 447 |
| SDRK01 sp007845205 | 0 | 2 080 | 523 368 | 2 270 |
| Microbacterium sp002456035 | 0 | 2 079 | 3 457 083 | 19 524 |

In table 4.2 we see the first unique k-mers when the k-mer length was 6. The number of distinct k-mers were 2080 or 2079 for all genomes. 2080 I the possible space of canonical 6-mers, meaning most genomes contained all possible 6-mers.

**TABLE 4.3: RESULTS OF K-MER LENGTH IMPACT ANALYSIS ON 8-MERS.**

| Species | Unique | Distinct | Total | Max count |
|---|---|---|---|---|
| Nitrosarchaeum limnae | 885 | 32 294 | 1 742 703 | 1 560 |
| UBA11579 sp003250455 | 25 | 32 892 | 3 118 033 | 1 068 |
| CAIYRG01 sp009693985 | 537 | 32 622 | 2 010 592 | 2 401 |
| SHVJ01 sp009694295 | 1 951 | 29 944 | 2 100 852 | 1 806 |
| JABDJX01 sp013003245 | 29 | 32 888 | 3 426 100 | 1 276 |
| GCA-2718035 sp902520385 | 2 170 | 31 137 | 778 410 | 1 258 |
| Cupriavidus basilensis_D | 2 | 32 895 | 8 543 263 | 7 250 |
| Eubacterium_I ramulus_A | 128 | 32 830 | 3 485 725 | 1 358 |
| SDRK01 sp007845205 | 1 825 | 31 724 | 523 282 | 368 |
| Microbacterium sp002456035 | 1 914 | 30 951 | 3 457 079 | 4 044 |

In figure 4.3, we see that at k-mer length 8 all genomes included unique k-mers. The number of distinct k-mers were in in the range from 29 944 to 32 895 range. This means the genomes covered most of all of the possible k-mer space.

**TABLE 4.4: RESULTS OF K-MER LENGTH IMPACT ANALYSIS ON 12-MERS.**

| Species | Unique | Distinct | Total | Max count |
|---|---|---|---|---|
| Nitrosarchaeum limnae | 820 577 | 1 137 665 | 1 742 399 | 61 |
| UBA11579 sp003250455 | 1 472 514 | 2 093 003 | 3 115 689 | 50 |
| CAIYRG01 sp009693985 | 856 053 | 1 230 577 | 2 010 320 | 82 |
| SHVJ01 sp009694295 | 773 139 | 1 191 631 | 2 100 432 | 67 |
| JABDJX01 sp013003245 | 1 488 146 | 2 205 843 | 3 424 892 | 317 |
| GCA-2718035 sp902520385 | 458 310 | 571 891 | 778 142 | 61 |
| Cupriavidus basilensis_D | 1 525 791 | 2 998 955 | 8 541 579 | 209 |
| Eubacterium_I ramulus_A | 1 479 936 | 2 204 635 | 3 484 633 | 118 |
| SDRK01 sp007845205 | 398 547 | 452 282 | 523 110 | 25 |
| Microbacterium sp002456035 | 718 380 | 1 366 990 | 3 457 071 | 116 |

In figure 4.4, we see that at length 12 the number of distinct k-mers were getting much closer to the total number of k-mers. There was also a drastic increase in unique k-mers and decrease in k-mer frequencies. However, the majority of k-mers still appeared more than once. There was more variation in the number of distinct k-mer compared to the earlier shorter k-mers. However, the median was around 1 200 000, which was only around 14% of the possible k-mer space of 8 390 656.

**TABLE 4.5: RESULTS OF K-MER LENGTH IMPACT ANALYSIS ON 16-MERS.**

| Species | Unique | Distinct | Total | Max count |
|---|---|---|---|---|
| Nitrosarchaeum limnae | 1 702 125 | 1 720 647 | 1 742 095 | 14 |
| UBA11579 sp003250455 | 3 007 547 | 3 058 843 | 3 113 345 | 31 |
| CAIYRG01 sp009693985 | 1 952 088 | 1 979 436 | 2 010 048 | 28 |
| SHVJ01 sp009694295 | 2 033 767 | 2 064 677 | 2 100 012 | 19 |
| JABDJX01 sp013003245 | 3 352 977 | 3 386 784 | 3 423 684 | 278 |
| GCA-2718035 sp902520385 | 759 111 | 767 938 | 777 874 | 9 |
| Cupriavidus basilensis_D | 7 662 700 | 8 060 357 | 8 539 895 | 37 |
| Eubacterium_I ramulus_A | 3 392 371 | 3 428 052 | 3 483 541 | 110 |
| SDRK01 sp007845205 | 519 554 | 5 211 206 | 522 938 | 5 |
| Microbacterium sp002456035 | 3 214 889 | 3 328 688 | 3 457 063 | 45 |

In table 4.5, we see that at k-mer length 16, the vast majority of k-mers were unique as the difference between the unique and distinct column are modest. With a median of 3 328 688, and a possible k-mer space of 2 147 516 416, the distinct k-mers covered 0.15% or the possible k-mer space.

**TABLE 4.6: RESULTS OF K-MER LENGTH IMPACT ANALYSIS ON 32-MERS.**

| Species | Unique | Distinct | Total | Max count |
|---|---|---|---|---|
| Nitrosarchaeum limnae | 1 734 332 | 1 737 032 | 1 740 879 | 13 |
| UBA11579 sp003250455 | 3 062 209 | 3 082 958 | 3 103 969 | 5 |
| CAIYRG01 sp009693985 | 2 006 052 | 2 007 426 | 2 008 960 | 8 |
| SHVJ01 sp009694295 | 2 093 403 | 2 095 430 | 2 098 332 | 6 |
| JABDJX01 sp013003245 | 3 396 253 | 3 407 381 | 3 418 852 | 165 |
| GCA-2718035 sp902520385 | 772 256 | 774 426 | 776 802 | 5 |
| Cupriavidus basilensis_D | 8 382 704 | 8 454 684 | 8 533 173 | 15 |
| Eubacterium_I ramulus_A | 3 452 197 | 3 461 788 | 3 479 173 | 96 |
| SDRK01 sp007845205 | 521 780 | 522 011 | 522 250 | 3 |
| Microbacterium sp002456035 | 3 441 747 | 3 449 084 | 3 457 031 | 11 |

In table 4.6, we see that at k-mer length 32 only a fraction of k-mers were not unique. The k-mer frequencies were falling below double digits, indicating that non-unique k-mers appeared at much lower frequencies than in the earlier, shorter k-mers.

**TABLE 4.7: RESULTS OF K-MER LENGTH IMPACT ANALYSIS ON 64-MERS.**

| Species | Unique | Distinct | Total | Max count |
|---|---|---|---|---|
| Nitrosarchaeum limnae | 1 735 252 | 1 736 572 | 1 738 447 | 10 |
| UBA11579 sp003250455 | 3 068 932 | 3 077 055 | 3 085 217 | 3 |
| CAIYRG01 sp009693985 | 2 006 128 | 2 006 436 | 2 006 436 | 3 |
| SHVJ01 sp009694295 | 2 093 560 | 2 094 192 | 2 094 972 | 4 |
| JABDJX01 sp013003245 | 3 397 918 | 3 403 488 | 3 409 188 | 68 |
| GCA-2718035 sp902520385 | 771 508 | 773 083 | 774 658 | 2 |
| Cupriavidus basilensis_D | 8 408 248 | 8 462 475 | 8 519 733 | 8 |
| Eubacterium_I ramulus_A | 3 459 988 | 3 464 991 | 3 470 437 | 7 |
| SDRK01 sp007845205 | 520 838 | 520 856 | 520 874 | 2 |
| Microbacterium sp002456035 | 3 444 103 | 3 450 424 | 3 456 967 | 6 |

In table 4.7, we see that at k-mer length 64, the difference between unique and distinct k-mers was very modest, in some genomes it was as small as a few thousands. At this length, almost all k-mers were unique. The number of distinct k-mer were a small fraction of the possible k-mer space.

**TABLE 4.8: AVERAGE VALUES FOR EACH K-MER LENGTH, SUMMARY OF TABLES 4.1 - 4.7.**

| Average | Unique | Distinct | Total | Max count |
|---|---|---|---|---|
| 4-mers | 0,0 | 136,0 | 2 919 381,1 | 91 217,1 |
| 6-mers | 0,4 | 2 079,4 | 2 918 992,5 | 13 050,0 |
| 8-mers | 946,6 | 32 017,7 | 2 918 603,9 | 2 238,9 |
| 12-mers | 2 324 139,3 | 1 545 347,0 | 2 917 826,7 | 110,6 |
| 16-mers | 2 759 712,9 | 3 300 662,8 | 2 917 049,5 | 57,6 |
| 32-mers | 2 886 293,3 | 2 899 222,0 | 2 913 942,1 | 32,7 |
| 64-mers | 2 890 647,5 | 2 898 957,2 | 2 907 692,9 | 11,3 |

In table 4.8, we see the averages for the number of unique k-mers, distinct k-mers, total number of k-mers, and k-mer frequencies. The results show that there is a strong link between k-mer length and the number of distinct or unique k-mers. This is due to longer k-mers allowing for greater variation as the number of possible k-mers increases exponentially with k-mer length. This difference also greatly impacts the highest frequency as longer k-mers are less likely to be repeated in the same genome. However, the impact on the total number of k-mers is modest as the average total is in the 2 900 000 to 2 920 000 range meaning the total varies with roughly 20 000 k-mers, which is less than a 1% difference. This total can also be used to calculate the number of nucleotides in the genome using the following formula: $genome\ length = total + k - 1$, where total is the total number of k-mers, and k is the k-mer length. For example, selecting the average total number of 12-mers from table

4.11 is section 4.1 gives a genome length of: $2\ 917\ 826{,}70 + 12 - 1 = 2\ 917\ 837{,}70$. The files for this test were selected randomly from different parts of the database, yet considering the modest difference in total k-mer they are similar in length. This indicates that the genomes in the database are of similar length. However, a sample of ten files it too small to generalize across a database of 47 894. The results also show that the number of distinct k-mers in the genome do not increase in tandem with the space of possible k-mers. Instead, when k-mers become longer they cover a smaller portion of the the possible k-mer space.

## 4.2 Distinct K-mers in MinHash Signatures

MinHash is a method of selecting k-mers. MinHash computes hash values for each k-mer in a genome and retains the 3000 lowest hash values in a genome signature. The goal of using MinHash was to find a subset of all the k-mers that could be used to classify many genomes with a reduced k-mers space. In section 3.6.1, MinHash signatures were created for every genome in the training set. Then, the hash values in those signatures were combined into one document to find every distinct hash in the training set of signatures, where a distinct hash means we count each hash only once, even if it is repeated. The combination resulted in a set of 80 052 hash values. Since the hash values represent k-mers, this means the set of signatures contained 80 052 distinct k-mers. As the training set contained 29 938 MinHash signatures, this equates to each signature contributing on average three unique k-mers. In this case, a unique k-mer is one that appears in only one signature. Each signature originally contained a selection of 3000 hash values. The fact that the combination of all distinct hash values resulted in a set of 80 052 hash values, implies a significant overlap of k-mers in the signatures. If there was no overlap, the number of different hash values would be 23 938 * 3000 = 71 814 000, as the training set contains 23 938 samples.

## 4.3 Finding Discriminative K-mers

Discriminative k-mers is a method of selecting k-mers that are unique to a single genome. The purpose of discriminative k-mers is to find a set of k-mers that can be used to uniquely identify individual genomes. In this project, described in section 3.6.3, discriminative k-mers were found by going through all files in the training set and comparing them to find k-mers unique to each genome.

Each file was then stripped to only include one discriminative k-mer per genome. These k-mers were then combined into one document holding all discriminative k-mers. Out of all 23 938 files in the training set, only 5 147 contained any unique k-mers. This means the discriminative k-mers came from 21,5% of the training set. This resulted in a 5 147 elements long set of discriminative k-mers to cover a dataset of size 47 894.

## 4.4 Prediction Results

We developed three types of neural networks: a standard neural network, a multilayer neural network, and a convolutional neural network (see section 3.7.2). Standard neural networks are small networks containing only three layers. Small networks have lower computational requirements and are simpler to train. Multilayer perceptron networks are deep networks with more layers. These are harder to train but can learn more complex relationships than standard networks. Convolutional neural networks contain convolutional layers. Convolutional networks are even harder to train, but they can discover higher level relations compared to other types of models. The purpose of implementing three different networks was to determine what type of network would be the most suitable for taxonomic classification on prokaryotic genomes (section 2.2.4.1 and section 2.2.5.2). In addition to different types of models, we also wanted to compare methods of selecting and representing k-mers to the models. The methods we implemented were MinHash, random 8mers, random 12mers, and discriminative k-mers (see section 3.6). MinHash, random 8mers and random 12mers were selected because these are different ways of randomly or semi-randomly selecting k-mers, while discriminative k-mers was selected because this method carefully selects each k-mer and is thus the opposite of the previous three methods. To compare these k-mer representation methods, we duplicated each of the three models so that each k-mer representation method was tested on each type of model, resulting in 24 distinct models. These 4 models were trained on and evaluated on a training set, then evaluated on a test set to assess each model's ability to classify previously unseen data. To evaluate each model, we measured them on their accuracy and precision. We used these metrics as they give a general overview of each model's ability to classify in a way that allow the models to be directly, and objectively compared. All models were trained and tested on both genus level and species level to determine how sensitive the model would be in detecting the more subtle differences between samples at species level compared to genus level. The results are displayed in tables 4.9 to 4.12.

**TABLE 4.9: TRAINING METRICS AT GENUS LEVEL.**

| Training genus | Model | Standard Neural Network | Multilayer Perceptron Neural Network | Convolutional Neural Network |
|---|---|---|---|---|
| Accuracy | MinHash | 99.8% | 37.2% | 1.6% |
| | Random 8mers | 99.9% | 35.9% | 1.5% |
| | Random 12mers | 99.3% | 37.7% | 1.5% |
| | Discriminative k-mers | 21.4% | 14.7% | 0.2% |
| Precision | MinHash | 99.9% | 80.9% | 34.0% |
| | Random 8mers | 99.9% | 81.5% | 28.2% |
| | Random 12mers | 99.6% | 81.2% | 41.6% |
| | Discriminative k-mers | 89.3% | 73.1% | 32.0% |

The results of evaluating the models on the training set at genus level show that the standard neural network scored high on accuracy and precision for most representation approaches, while the multilayer perceptron network scored a little lower, and the convolutional network had very low metrics (see table 4.9).

**TABLE 4.10: TESTING METRICS AT GENUS LEVEL.**

| Testing genus | Model | Standard Neural Network | Multilayer Perceptron Neural Network | Convolutional Neural Network |
|---|---|---|---|---|
| Accuracy | MinHash | 10.1% | 18.5% | 0.5% |
| | Random 8mers | 47.3% | 18.4% | 1.8% |
| | Random 12mers | 49.8% | 8.2% | 0.8% |
| | Discriminative k-mers | 0% | 0% | 0% |
| Precision | MinHash | 9.6% | 83.5% | 0% |
| | Random 8mers | 95.8% | 21.6% | 0% |
| | Random 12mers | 96.2% | 7.9% | 0% |
| | Discriminative kmers | 0% | 0% | 0% |

In table 4.10 we see the results of evaluating each model on a test set at genus level. The scores on the test set were generally lower compared to on the training set. We also see that the models could not correctly classify the discriminative k-mers at all.

**TABLE 4.11: TRAINING METRICS AT SPECIES LEVEL.**

| Training species | Model | Standard Neural Network | Multilayer Perceptron Neural Network | Convolutional Neural Network |
|---|---|---|---|---|
| Accuracy | MinHash | 1.0% | 0% | 0% |
| | Random 8mers | 0.0002% | 0.0004% | 0% |
| | Random 12mers | 19.9% | 0.0004% | 0% |
| | Discriminative k-mers | 19.3% | 0.002% | 0% |
| Precision | MinHash | 0% | 0% | 0% |
| | Random 8mers | 3.1% | 0% | 0% |
| | Random 12mers | 11.8% | 0% | 0% |
| | Discriminative k-mers | 12.1% | 0% | 0% |

At species level, we see that the results of the training set evaluation were lower as compared to genus level (see table 4.11). The convolutional network, in particular, struggled and received accuracy and precision scores of 0% for all representation methods.

**TABLE 4.12: TESTING METRICS AT SPECIES LEVEL.**

| Training species | Model | Standard Neural Network | Multilayer Perceptron Neural Network | Convolutional Neural Network |
|---|---|---|---|---|
| Accuracy | MinHash | 0% | 0% | 0% |
| | Random 8mers | 0% | 0% | 0% |
| | Random 12mers | 0% | 0% | 0% |
| | Discriminative k-mers | 0% | 0% | 0% |
| Precision | MinHash | 0% | 0% | 0% |
| | Random 8mers | 0% | 0% | 0% |
| | Random 12mers | 0% | 0% | 0% |
| | Discriminative k-mers | 0% | 0% | 0% |

In table 4.12, we see the results of testing each model at species level. In the table we can see that every model received scored of 0% regardless of representation method.

Tables 4.9 to 4.12 display the results of evaluating each model developed in this project. Overall, we can see that the convolutional neural network was the poorest model, and discriminative k-mers the poorest representation method, as these received the lowest metrics. All models achieved higher

accuracy and precision at genus level as compared to species level. Also, no model was able to correctly classify any sample at species level on a test set.

## 4.5 Resource Usage

Three types of neural networks were built for this project (see section 3.7); a standard neural network, a multilayer neural network, and a convolutional neural network. These were trained on four different representation methods: MinHash, random 8mers, random 12mers, and discriminative k-mers (see section 3.6), resulting in 24 distinct neural networks. Training all these models required significant computational resources in terms of memory and running time. Tables 4.13 and 4.14 contain an overview of the running time of each model, as well as the average memory usage. These metrics cover the resources used when each model was training on a training set of representation vectors made using the MinHash, random 8mer, random 12mer or discriminative k-mer methods.

**TABLE 4.13: TRAINING RESOURCE USAGE AT GENUS LEVEL.**

| Training genus | Model | Standard Neural Network | Multilayer Perceptron Neural Network | Convolutional Neural Network |
|---|---|---|---|---|
| Average Memory | MinHash | 51GB | 54GB | 45GB |
| | Random 8mers | 51GB | 57GB | 45GB |
| | Random 12mers | 51GB | 49GB | 44GB |
| | Discriminative k-mers | 51GB | 50GB | 45GB |
| Running Time | MinHash | 2d 1h 48min | 2d 10h 7min | 3d 11h 38min |
| | Random 8mers | 2d 3h 18min | 3d 2h 7min | 2d 12h 49min |
| | Random 12mers | 2d 8h 7min | 3d 1h 12min | 2d 14h 31min |
| | Discriminative k-mers | 2d 1h 7min | 3d 0h 23min | 2d 12h 35min |

When training the models at genus level, in general each model used between 44 and 57GB and required 2 – 3 days of training (see table 4.13).

**TABLE 4.14: TRAINING RESOURCE USAGE AT SPECIES LEVEL.**

| Training species | Model | Standard Neural Network | Multilayer Perceptron Neural Network | Convolutional Neural Network |
|---|---|---|---|---|
| Average Memory | MinHash | 76GB | 68GB | 62GB |
| | Random 8mers | 59GB | 51GB | 52GB |
| | Random 12mers | 51GB | 68GB | 62GB |
| | Discriminative k-mers | 51GB | 68GB | 62GB |
| Running Time | MinHash | 3d 14h 42min | 2d 22h 57min | 7d 0h 0min |
| | Random 8mers | 3d 1h 52min | 1d 7h 50min | 6d 1h 3min |
| | Random 12mers | 2d 20h 9min | 2d 23h 12min | 7d 4h 46min |
| | Discriminative k-mers | 3d 3h 42min | 2d 18h 13min | 6d 20h 36min |

When trained at species level, we can see that each model used between 51 – 76GB and ran for 2 – 7 days (see table 4.14).

Tables 4.13 and 4.14 show the computational resources used while training each of the neural networks. The results show that training at species level required more memory and longer run times. This was especially true for the convolutional neural network that generally took twice as long to train at species level. However, testing each model once train was much faster. In general, testing each model on a test set of 9 576 samples took between 25 to 40 minutes.

**TABLE 4.15: OVERVIEW OF THE MEMORY REQUIRED TO STORE EACH MODEL ONCE TRAINED.**

| models | Representation approach/Model | Standard Neural Network | Multilayer Perceptron Neural Network | Convolutional Neural Network |
|---|---|---|---|---|
| Genus | MinHash | 14GB | 18GB | 2GB |
| | Random 8mers | 14GB | 18GB | 2GB |
| | Random 12mers | 14GB | 18GB | 2GB |
| | Discriminative k-mers | 14GB | 18GB | 2GB |
| Species | MinHash | 19GB | 20GB | 4GB |
| | Random 8mers | 9GB | 8GB | 4GB |
| | Random 12mers | 19GB | 20GB | 4GB |
| | Discriminative k-mers | 19GB | 20GB | 4GB |

The memory required to store each model once finished is displayed in table 4.15. We can see that the size of the finished models varied depending on model type and representation method.

Convolutional models had the lowest memory requirements, while multilayer perceptron networks had the highest. We also see that at species level, random 8mer models required less memory as compared to models trained on other representation methods.

## 4.6 Assessing the Role of Chance

Machine learning is a form of black-box programming, meaning we are not in control of all factors that impact the result. Since machine learning programs learn autonomously, it is difficult to determine the role that random chance plays in the result. Factors such as the initial value of the weights in a neural network can create different results each time the network is trained. This uncontrollability reduces the certainty that the variations observed in the results are due to the differences purposely introduced, such as different model designs and k-mer representations. To assess whether the results could be considered to reflect the factors we wanted to check rather than random variation, the standard neural networks were re-trained and tested for each representation method. The purpose of this was to compare the new results to the previous ones from when the models were first trained to assess the role of chance in our observations.

**TABLE 4.16: TRAINING METRICS ON STANDARD NEURAL NETWORKS USED TO ASSESS THE IMPACT OF RANDOM VARIATION.**

| Training genus | Representation method/Model | Standard Neural Network |
|---|---|---|
| Accuracy | MinHash | 96,8% |
| | Random 8mers | 99.3% |
| | Random 12mers | 99.0% |
| | Discriminative k-mers | 18.3% |
| Precision | MinHash | 99.8% |
| | Random 8mers | 99.9% |
| | Random 12mers | 99.4% |
| | Discriminative k-mers | 90.6% |

When testing the standard neural networks at genus level, most metrics were in the 90% range, except for the accuracy for discriminative k-mers (see table 4.16).

**TABLE 3.17: TESTING METRICS ON STANDARD NEURAL NETWORK USED TO ASSESS THE IMPACT OF RANDOM VARIATION.**

| Testing genus | Representation method/Model | Standard Neural Network |
|---|---|---|
| Accuracy | MinHash | 16.2% |
| | Random 8mers | 42.3% |
| | Random 12mers | 48.3% |
| | Discriminative k-mers | 0% |
| Precision | MinHash | 79.2% |
| | Random 8mers | 95.7% |
| | Random 12mers | 96.4% |
| | Discriminative kmers | 0% |

Testing the standard neural networks on a test set resulted in random 12mers receiving the highest accuracy and precision, while discriminative k-mers' accuracy and precision were both 0% (see table 4.17).

Compared to the first time the standard neural networks were run, the metrics received the second time were not hugely different. Just like the first time, random 8mers received a higher accuracy than 12mers during training, while random 12mers had a higher accuracy during testing.

# Chapter 5: Discussion

Microorganisms play an intricate role in health and the ecosystem. Thus, being able to identify which microorganisms are present in a sample, can be very valuable. For example, the presence of a certain family may indicate disease. However, performing taxonomic classification on these domains can be particularly difficult due to them having unstable gene pools as they develop rapidly and can exchange genes through horizontal gene transfer. Due to modern sequencing methods, there is a lot of genomic information available for prokaryotes. However, analysing and using large quantities of data is challenging. Researchers are working on using machine learning to solve this problem. While several tools have been developed for this purpose, the research comparing the underlying method these tools use is limited, making it difficult to formulate any general guidelines of how machine learning for taxonomic classification of microorganisms should be implemented. Most of the tools are also developed for marker gene analysis, while other sequencing methods such as metagenomic analysis are lagging behind. In this master thesis, we have attempted to answer this question, by classifying whole genomes for prokaryotes. We have done this by implementing three different neural networks: a standard neural network, a multilayer perceptron network, and a convolutional neural network, and compared their ability to classify on four different methods of representing microbial genomes: MinHash, random 8mers, random 12mers, and discriminative k-mers. These were then measured on their accuracy and precision to determine what model and what microbial representation method is the most suitable. The goal was not to develop a new tool to compete with existing tools, but to make observations on the interaction between neural network and representation method to determine what are the most promising methods to pursue for future development.

## 5.1 K-mer Length

A k-mer is an extracted piece of DNA of length k (section 2.3.1). Some of the main reasons to use k-mers to represent DNA sequences, are the lowering of required computational resources and genome completeness. The k-mer length can have significant impact on the accuracy during classification tasks and should thus be selected carefully. For example, longer k-mers contain more information and can thus increase accuracy, but they also require more computational resources as

compared to shorter k-mers (Kaehler, 2017). When used for machine learning k-mers are represented as binary presence/absence vectors, or count vectors denoting k-mer frequency.

In this project, several tests were run on ten randomly selected genome files to get an indication as to how k-mer length would impact factors such as the number of distinct and unique k-mers in the genomes (section 3.5 and 4.1). From k-mer length 12 and up, the difference between unique k-mers and total number of k-mers was drastically reduced compared to k-mer lengths 8 and below. The implication is that when the k-mer length is 12 or greater, more of the k-mers in the genome will be unique. When most of the of k-mers in a genome only appear once, there is little purpose in knowing the k-mer frequency as almost all k-mers will have a frequency of 1. Therefore, when representing a genome as k-mers it would be more efficient to use a presence/absence-based representation when the k-mer length is 12 or above. For k-mer lengths below 12, however, a k-mer counting based method is more suitable as most k-mers appear at frequency > 1. Shorter k-mers contain less information, but including the count makes them more informative. Shorter k-mers also means the possible number of k-mers is much smaller, increasing the likelihood that identical k-mers will be present across genomes. A presence/absence-based method on these short k-mers could therefore result in input vectors which are near identical even if the genomes they represent are not.

Based on the reasoning in this section, k-mer counts should be represented by k-mers of length 8 or below, while presence/absence-based k- should be represented by k-mers of length 12 or above. Longer k-mers increase memory requirements (see section 2.3.2) and the use of long k-mers is therefore a trade-off with computational resources. Therefore, presence/absence-based k-mers should be based on 12-mers, as this length primarily produces unique k-mers suited for absence/presence representation while also being the theoretically least resource demanding long k-mer. 12-mers is also the same length that LaPierre et al (LaPierre *et al.*, 2019b) (see section 2.3.2) found to be a suitable trade-off in their experiments on a similar project, indicating that 12-mers as the first length to produce primarily unique k-mers is not specific to the database and exploration performed in this project. Computational resources are less of a concern when selecting a k-mer length for k-mer counts as these will use short k-mers. There are, however, concerns about selecting too short k-mers as these are less likely to catch repeats of short sequences. A short sequence, such as ATA is likely to appear in a genome containing millions of nucleotides, while the sequence TATATATA is less likely, and thus more informative. To mitigate this issue, we should use the longest

short k-mers, which is 8-mers. 8-mers are short enough to garner the benefits of short k-mers, but as the longest of the short k-mers should be less prone to the negative aspects.

According to Chor et al (2009), in prokaryotes the k-mer frequencies follow a unimodal distribution (Chor *et al.*, 2009). Unimodal means that the distribution has a clear peak where values increase before the peak and decrease after it. This may be the reason that from k-mer lengths 12 and longer, the k-mers in the genome for the most part do not cover most of the possible space of k-mers. If the function on either side of the peak is steep, that means even in a genome with many k-mers, most of the k-mers are part of the same limited subset. As such, the real space of k-mers is far smaller than the theoretical space of k-mers. Based on the results in section 4.1, it appears that as the k-mer length increases, the k-mers in the genome are covering a smaller percentage of the space of possible k-mers.

## 5.2 Representation Methods

The neural networks were trained on four separate methods of representing k-mers; MinHash, random 8mers, random 12mers, and discriminative k-mers (see section 3.6). MinHash selected k-mers using locality-sensitive hashing and then retaining the 3000 lowest hashes for each genome. These hashes were used as a basis to create 40 000 elements long presence/absence vectors for each genome. The goal was to select a subset of k-mers whose presence could be used to classify many genomes due to being shared across genomes. Random 8mers and random 12mers was based on generating 40 000 arbitrary k-mers of length 8 and 12. The random 8mers were counted for each genome and turned into k-mer count vectors. The random 12mers were counted for each genome and turned into presence/absence vectors. The purpose of random 8- and 12mers was to assess whether more complex selection methods, such as MinHash, is worth the extra effort, or if simpler methods, such as generating random k-mers can serve just as well. These methods were also included to make a comparison between a k-mer presence/absence-based approach and a k-mer counting-based approach. Discriminative k-mers were implemented by counting all 12mers in the training set, then removing all k-mers that existed in more than one genome. Then, this set was scaled down to one k-mer per genome, and each genome into a presence/absence vector covering the discriminative k-mers.

During training on the standard neural network at genus level, the model learned to classify random 12mer vectors quickest. After five epochs, the random 12mer had achieved an accuracy on the training data of 94%, while random 8mers and MinHash were both at 65%. The final accuracy score achieved on training data were 99.9% for random 8mers, 99.8% for MinHash, 99.3% for random 12mers, and 21.4% for discriminative k-mers. All representations achieved a precision of around 99%, except for the discriminative k-mers at 89.3%. Discriminative k-mers performed much worse than the others. For the rest, k-mer representation had limited impact. However, when the models were tested on the testing set the representations mattered more. Random 12mers achieved the highest accuracy and precision at 49.8% and 96.2% respectively. MinHash fared far worse with accuracy and precision scores of 10.1% and 9.6%. Discriminate k-mers did not classify anything correctly and thus has accuracy of precision scores of 0%.

On the genus-level multilayer perceptron neural network, discriminative k-mers once more fared the worst with accuracy 14.7% and precision 73.1%. Except for discriminative k-mers, random 8mers achieved the lowest accuracy during training at 35.9%, while MinHash had the highest accuracy at 37.2%. MinHash had the lowest precision, at 80.9%, compared to random 8mers at 81.5% and 12mers at 81.2%. Testing the model, however, yielded the opposite result. MinHash had the best result with an accuracy of 18.5% and precision of 83.5%. In fact, the MinHash model achieved a higher precision during testing as compared to training.

The differences between the accuracy scores for the genus-level convolutional neural network were modest, ranging from 0.2% to 1.6%. Random 12mers achieved the highest precision score at 42.6%. This was the only metric where discriminative k-mers were not the worst with a precision of 32.0%. However, discriminative k-mers still has the lowest accuracy at 0.2%. Random 8mers performed best on the test set with an accuracy of 1.8%, while discriminative k-mers achieved the worst accuracy at 0%. All representations had a precision of 0%.

Interestingly, the performance of each representation method appears to be largely model dependant, as no method is consistently better than the others. However, discriminative k-mers is consistently worse than the other three. The purpose of MinHash was to find a set of k-mers that are present in many genomes reducing the needed input vector length for classification. It is unexpected that randomly selected k-mers outperformed k-mers selected using MinHash sketching on two out of

three models. The k-mers in MinHash were purposely selected with the aim to find k-mers that exist in many genomes and can thus be part of a reduced subset of k-mers able to classify a wide range of genomes. In random 8- and 12mers however, the k-mers were selected without even looking at the dataset. There was no guarantee that the k-mers were present in the dataset at all, only a statistical probability that at least some of them were. The vectors in the random methods were the same length as the MinHash vectors, and the models were trained with the same parameters. The MinHash vectors were also pre-processed in the same way as the random 12mer vectors as they are both presence/absence-based representations with 12mers. When the hash values from the MinHash signatures were combined it became apparent that there was significant overlap in the k-mers in the genome (see section 4.2). It is possible that this overlap allowed the random methods to achieve the same as the MinHash sketches. By that, we mean that some of the randomly generated k-mers were present in many genomes and thus were suitable for classification using a reduced input vector, just like MinHash aims to do. Another factor that might have played a role, is the unreliability of absence in a MinHash presence/absence vector. When the MinHash signatures were made, a k-mer was marked as present if its hash value was present in a genome's signature. As such, we can thrust that if a k-mer is marked present in the MinHash vector, it is also present in the original genome. However, we cannot make such a claim for k-mers marked absent. For example, we have two genomes that contain four k-mers each. We would like to make a signature with the two lowest hash values in each genome. The hash values computed for the two genomes form the vectors [1, 9, 4, 7] and [8, 4, 2, 1]. The signatures for each genome would thus be [1, 4] and [1, 2]. These are then combined into the set [1, 2, 4] and compared with the signatures to form the presence/absence vectors [1, 0, 1] for genome one and [1, 1, 0] for genome two. However, for the second vector to be a correct representation of genome two, the vector should be [1, 1, 1] as both the genome and combined set contains the hash value 4. Considering the signatures lengths in our training set constitute roughly 0,01% of the total number of k-mer in each genome, it must be considered highly likely that such inaccuracies are present in the training set. However, the error should be reduced since the combined set was cut down to 40 000 hash values as this reduces the overall number of k-mers marked absent in the representation vectors.

When comparing random 8mers and random 12mers, each method outperformed the other on one of the models. This makes it harder to draw any conclusion as to whether k-mer presence/absence or k-mer counting is the superior method. Therefore, which method is preferable must be decided not simply by their metrics, but on the quality of the model on which they outperformed the other method, because it would be best to pursue the k-mer representation method that performed the

best on the most promising model. However, it must be mentioned that on the model where random 8mers outperformed random 12mers, the gap in the methods' performance was larger than when 12mers outperformed 8mers. Also, in their article, Vinje et al (2015) found that they were able to improve performance on one of their k-mer representation methods by adding k-mer counting rather than just using k-mer presence/absence (Vinje *et al.*, 2015). However, the method in the article was based on naïve Bayes classifier and was tested on 16S marker genes. A factor that might have impacted the result for random 8mers, was the decision to transforms the k-mer frequencies by scaling rather than using log transform. Log transform is a suitable transformation method for re-scaling the count in the input vector when the counts follow a log-normal distribution (see section 2.2.2.7). According to Choer et al (2009), log-normal distribution is a suitable model for describing the k-mer frequencies in prokaryotes (Chor *et al.*, 2009). This means the random 8mer vectors could have been transformed differently which may have altered the results.

In general, discriminative k-mers performed very badly during testing. The most likely explanation is that the discriminative k-mers were too discriminative. The discriminative k-mers came from only 5 147 files, and as such represented 21,5% of the training set. This means that most of the genomes were not unique enough to contribute a unique k-mer, even though the space of possible k-mers was 16 777 216 as the method was based on 12-mers. As a result, the models had nothing to work with when classifying most of the genomes as their entire input vectors would be nothing but zeros. In section 5.1, we discussed how the k-mer distribution in prokaryotic genomes do not cover the whole space of possible k-mers, and the greater the k-mer length, the less k-mer space is covered. This must logically have been a contributing factor as to why discriminative k-mers did not work as a representation method. The genomes are generally too similar for this method to work, at least on 12-mers. Solving this would require either increasing the likelihood that each genome contains a discriminative k-mer, or implementing a less strict approach to discovering discriminative k-mers. The first approach could be achieved, for example, by lengthening the k-mers. Each additional nucleotide exponentially increases the possible k-mer space and thus the number of potentially unique k-mers. However, as we discovered, the longer the k-mers get, they cover less of the k-mer space. Also, as the k-mers are selected on an individual level, generalization across genus may be difficult, and this approach would only increase the required computational resources. The second approach could be achieved in several ways. One way would be to select discriminative k-mers at genus level rather than individual level, keeping k-mers that are unique to a genus group rather than an individual

sample. This approach relies on the assumption that samples in the same genus are more likely to share k-mers. Another way would be to not select k-mers that are necessarily unique, but instead just unusual. For example, counting the frequency of every k-mer in the dataset, then retaining the most unusual k-mers below a certain threshold. This would be an example of a filter method (see section 2.2.2.4). Another option would be to use a wrapper method, for example randomly selecting a subset of the k-mers in the training set, train them on a few epochs, evaluate the accuracy, then do the same with a new subset of the k-mers, repeating the process until eventually we have selected the optimal subset. However, this would require a long run-time to achieve. It must also be mentioned that the discriminative k-mer vectors were 5 147 elements long, making them far shorter compared to the other methods with 40 000 elements long vectors, thus decreasing the overall amount of information the models received for each genome.

## 5.3 Models

The models developed for the classification task were a standard neural network, a multilayer neural network, and a convolutional neural network (see section 2.2.4.1 and section 2.2.5.2). Standard neural networks are the simplest neural networks as they are relatively small. The networks are less complex than the others and can thus not learn as complex features as more complex networks. However, they are the least expensive to train in terms of computational requirements. Multilayer perceptron networks are deep neural networks with more layers than a standard neural network. Thus, they can learn more complex relationships. Convolutional neural networks contain convolutional layers. Convolutional layers can extract high-level data as compared to multilayer perceptron networks and are known for their usefulness in image recognition.

Comparing the measured accuracy from training and testing, we can see that the standard neural networks had the largest gap between achieved accuracy during testing compared to training. This large gap indicates a high level of overfitting. Interestingly, the larger multilayer neural network had a smaller gap indicating less overfitting despite that it would be logical to think a more complex neural network would learn the data better and thus be more prone to overfitting. Generally, less complex models are less capable of detecting subtle differences in the data (see section 2.2.4.6.1). This makes them less likely to overfit the data, and more able to generalize. However, in this case we seem to

have gotten the opposite result. One possible explanation is that the smaller hidden layers in the multilayers neural networks more effectively forced the network to generalize as it could not pass on every input to the next layer. Considering the small size of the networks compared to the large dataset they were trained on, there is the distinct possibility the standard networks are oversaturated. By that, we mean that there are only so many patterns a network can learn, and this is limited by its size. As a result, when the network is further trained it does not learn more patterns, but rather switches what patterns it has learned.

The multilayer perceptron networks did not achieve the same accuracy during training or testing as the standard neural network. The precision during testing, however, was substantially better when averaging out between different k-mer representations. The gap between the metrics on the training and test sets were smaller than the standard neural network, indicating less overfitting. We can draw from this that while the standard neural network learned the training data, the multilayer perceptron network to a greater extent learned to generalize the data. While the result was sub-par compared to the standard neural network, the purpose of training is to have the network learn to generalize across groups, and the multilayer perceptron performed better at that despite achieving lower metrics.

The convolutional neural networks overall had the poorest accuracy scores. Possibly, the reason for this is that the convolutional layers are not suited to the type of input. While convolutional networks are powerful in discovering complex relationships, they are designed for input where the order of features in an input vector is meaningful. Therefore, they are commonly used in image recognition where a feature is combined with its neighbours to form a shape. As such, the convolutional layers are a form of feature combination method (see section 2.2.2.4). In our project this technique did not translate well when applied to microbial sequential data. However, the networks might have fared better on a different k-mer representation method. As convolutional networks put a lot of emphasis on combining neighbouring features, we might have gotten a better result if the k-mer vectors retained the ordering from the original genome. This would make filters in the convolutional layer more meaningful, as the combining of neighbouring k-mers would result in filters that represented a longer k-mer. As such, a suitable representation method would be a presence/absence vector where each position represented a position in the genome, and the value identified a specific k-mer. However, this method would be computation heavy due to the long vectors required to represent an entire genome. Another issue would be adapting the input vectors to model specific requirements,

such as making all input vectors the same length. The genomes in the dataset are not of uniform length. Thus, with the tools and resources used in this project, representing every genome length in its entirety would require most vectors to be padded to the length of the greatest genome, probably drastically increasing the computational requirements. To get a better result with the three representation methods we already have, it might be necessary to make alterations to the structure of the convolutional models. For example, the model contained two dropout layers intended to reduce overfitting. As the gap between the accuracy on the training and test sets was small, the model succeeded in reducing overfitting. However, the accuracy on both was low, perhaps indicating too much emphasis on reducing overfitting in the model design.

There is the possibility that coincidence has impacted the results. Training a neural network is a form of black box programming. We are not in control of, or even aware of the specificities of how the trained model works once trained. This is one of the disadvantages of neural networks, and machine learning in general. Different values of the initial weights in the network can impact the result. The learning rate used to train the neural network was 0.1. This is relatively high (see section 2.2.4.5.1). This choice was made to speed up the learning process. However, a higher learning rate makes the model less likely to discover the global maxima. The high learning rate also increases the likelihood that chance played a role in the result as the model weights were changed more drastically per epoch creating a less consistent learning curve. The standard neural network was rerun on all three representation methods to determine the impact of random variation in the results (see section 4.6). The results of the re-training and testing showed that while the metrics were altered slightly, they kept to the same levels as the first time. Meaning, that which representation method received the highest and lowest metrics did not change. This indicates that the random variation in the the standard neural network is not large enough to garner criticism as to the validity of our observations. However, this does not mean this is not the case for the multilayer or convolutional neural networks.

## 5.4 Species Classification

All models were better at classifying at genus level across all representation methods. This is not surprising as the differences at species-level are bound to be more subtle. Another important factor is that the dataset contained only one example of each species. The result was that the models could

not generalize across a group of individuals, only learn to recognize that single individual. Since the models were tested on a test set that contained none of the samples from the training set, during testing the models were presented only with species they had not seen before. Thus, they should logically not be able to classify any of them correctly. Therefore, all models got accuracy and precision scores of 0% regardless of k-mer representation method. Ideally, the models should have had the option to classify a sample as belonging to no class, meaning the sample did not belong to any of the classes the models had seen before. In this case, if the models classified everything in the test set as no class, they would have 100% accuracy, even if this result is not very helpful. Based on the results, it is unknown if the model is sensitive enough to classify at species level or not due to the inadequacy of the training and test data. A solution would be to increase the number of samples for each species. This could be achieved either by getting more species data from a different database, or artificially increasing the sample size using data augmentation (see section 2.2.4.6.1). However, the observations made about which models are most suited for classification should be the same for both genus and species-level. Logically, there is no reason the standard neural networks should overfit the genus level-data, but not the species-level data as in practice the only difference is the size of the input vectors and number of classes (genus or species) to map the vectors to.

## 5.5 Our Project in the Wider Field

In an article about fungal taxonomic classification, researchers found convolutional neural networks to outperform several other types of models (Vu, Groenewald and Verkley, 2020). The k-mers were represented as k-mer frequency vectors, just like in our random k-mer count representation method. It is therefore reasonable to assume the original order of the k-mers in the genome was not retained in the vectors. Yet, the convolutional neural network outperformed the other models at family and higher taxonomic levels. This challenges the earlier presented idea that it is the lack of a meaningful relationship between neighbouring features that lead to the convolutional models performing poorly. It is necessary to note, however, that there are several differences between our convolutional models and theirs, such as our models including dropout layers. Also, their model was trained on fungi marker genes rather than bacterial and archaea whole genomes, and they used k-mer sizes 4, 6, and 8, rather than 8 and 12.

Kraken 2 is a newer version of the classification tool Kraken (Wood, Lu and Langmead, 2019). In an article comparing the performance of Kraken and Kraken 2, several existing tools were tested on

various metrics, including precision (Wood, Lu and Langmead, 2019). The tools being tested on nucleotide classification were Kraken 2, earlier versions of Kraken, CLARK, Kaiju, and Centrifuge (see section 2.1.5.4.1). For all tools, the precision was around 99%. The tools compared in this article, are all k-mer based, metagenomic genus-level classifiers that were tested on prokaryotic data. As the highest precision achieved by our models on a test set was 96.2% (random 12mers on the standard neural network), this would indicate that our models are not up to the standard of current, already existing classifiers. It is worthy to note, however, that the precision score achieved by the other classifiers was based on a test set of 40 genomes, while our models were tested on 9 576. We can thus have more confidence in the precision score of our own models as they were tested on a wider array of inputs. Kraken 1 and 2 used k-mers lengths 31 and 35. In our project, we used k-mer lengths 8 and 12, to balance the trade-off between the greater informative nature of long k-mers and computational resources (see section 5.1). The superior precision score of Kraken 1 and 2 challenges the outcome of this consideration as it indicates the advantages of long k-mers should have been given more weight as compared to concerns over computational resources.

In this project, we decided to work on taxonomic classification for metagenomic analysis as this type of classification is less common than marker gene analysis and thus needs the research more. However, since working on whole genomes requires a lot of computational resources, we must consider whether it is worth it. In our own project, the classifiers required between 44 and 76GB of memory during training, and once trained required between 2 to 20GB to be stored (see section 4.5). Species-level classifiers required more memory than genus-level. On species level, the classifiers trained on random 8mers required less memory compared to the other models. Perhaps due to the shorter k-mers they were trained on. In an article by Ye et al (2019), comparisons were made between 20 different classification tools on the same database (Ye *et al.*, 2019). This list included both metagenomic and marker gene classifiers. They found that metagenomic classifiers using k-mers longer than 30 had the best scores, while marker-gene based tools performed less well. However, they also found that metagenomic classifiers required from tens to hundreds of gigabytes to run, while the memory requirements for marker gene classifiers were lower. As such, the memory requirements for our models were low for metagenomic classifiers. Perhaps due to being trained on shorter k-mers compared to most other metagenomic classifiers, just as we saw a connection between k-mer length and memory requirements in our models. The metagenomic classifiers were fast during testing, despite their memory requirements (Ye *et al.*, 2019). While our own models were

slow to train, they worked fast on the test set (see section 4.5). Overall, it seems that if enough memory is available, metagenomic classifiers are more promising than marker gene classifiers.

# Chapter 6: Conclusion

## 6.1 Models

High accuracy and precision on training data indicates standard neural networks are good at learning to recognize a microbial dataset. However, these models display a high level of overfitting, indicating limited capacity for generalizing on a large dataset. Thus, there is little purpose in pursuing these models for classification of metagenomic data. Multilayer perceptions are less precise and accurate than standard neural networks. However, lower levels of overfitting indicate these models might be better at generalizing. Further research is required to determine the exact structure these should have for an optimal result, such as the ideal number of layers and nodes per layer. However, these models being better at generalizing make them a more promising venue than standard neural networks to explore further as a method for metagenomic analysis. Convolutional neural networks do not work well for the k-mer representation methods used in this project. Possibly, because of the lack of meaningful relation between a single feature in the input vector and its closest neighbours. However, more success might be possible with this type of model if k-mer vectors retain the ordering of the k-mers in the genome. Thusly, using convolutional neural networks to classify metagenomic data is worth pursuing if the data can be represented by vectors where the ordering of elements reflect the ordering of those same elements in the genome. Overall, deep neural networks hold more promise than traditional neural networks for handling a domain such as microbial classification.

## 6.2 Representation Methods

Random 12mers achieved the highest accuracy score on a test set. However, this was on the standard neural network. In section 6.1, we concluded that standard neural networks were not worth pursuing further for taxonomic classification in favour of multilayer perceptron networks. Thus, it would be better to select the representation method that fared best on the multilayer perceptron network, which was MinHash. MinHash's accuracy and precision scores where the highest during testing compared to the random methods. Especially precision where MinHash achieved a score roughly four times higher than the second highest score. Random 8mers, while performing significantly worse than MinHash, achieved an accuracy and precision score more than twice as high as random 12mers. As discussed in section 5.3, there is the possibility that the results

are partly arbitrary, as we did not check for impact of coincidences by re-training the models or re-generating new random k-mers. However, if we assume the role of chance was limited, we can conclude the following. Structured selection of k-mers is preferable to random selection. However, when this is not possible, k-mer counting is preferable to k-mer presence/absence when the input vectors are of equal length. On the usefulness of discriminative k-mers, our project is inconclusive. The results indicate this approach is not worthwhile, however a different implementation may garner a different result.

## 6.3 Genus and Species

During testing at genus level, all models managed to classify some samples correctly. This proves the models can generalize at genus level for all three representation methods. However, at species level no model managed to classify a single sample correctly. This is due to the training and testing sets being insufficient for species-level models. Therefore, no sound conclusion can be made about the models' ability to classify at species-level other than that this requires further research.

## 6.4 Further Work

Multilayer perceptron neural networks show promise as a method for taxonomic classification. However, as the accuracy and precision achieved in the project is relatively low, more experimentation is needed to increase these metrics to a level that is acceptable for real world usage. Running experiments on the same dataset but with different variations to the multilayer perceptron network could lead to the building of a better neural network. Variations that should be tested are adding/removing layers and varying number of nodes per layer.

Expanding the comparisons made in this project to other types of neural networks could allow for a more complete recommendation for what neural network to use for taxonomic classification of microorganisms. The results produced in this project can easily be expanded upon by implementing new types of models, such as deep belief networks, and testing them on the same dataset. Similarly, the results could benefit from the inclusion of more k-mer representation methods. Particularly,

exploring representation vectors that retain the order in which each k-mer is found in the genome. This would encourage the use of models such as recurrent neural networks, that are particularly suitable for input vectors where the ordering of elements is meaningful, due to their ability to treat the input as a more complete sequence. Encoding k-mers using one-hot encoding or embedding could be a starting point. In an article about the development of the classification model DeepMicrobes, researchers found that embedding improved model performance over one-hot encoding (Liang *et al.*, 2020).

The classification labels in the project were encoded using one-hot encoding. The reason for this was that there is no obvious way to sort the training samples numerically, and a numerical label can affect the model performance as the models interpret labels with similar numerical values as being more closely related. If we could encode the labels in such a way that the difference between the numerical labels reflected the level of closeness on the evolutionary tree, we could use this to our advantage to improve model performance. However, designing such a numerical labelling system could prove a substantial task. Especially, if the model is intended to keep learning new classes after the initial training as the labelling system would need the flexibility to accommodate this.

The discussion left ambiguity as to the worth of convolutional neural networks for taxonomic classification. In the results, the convolutional neural networks proved the least suited as they achieved the lowest metrics. More research is required to determine if convolution is unsuited for classification tasks in this domain, or if they have potential but require a different approach. Testing convolutional neural networks with a genome representation method more in line with what these networks were originally designed for could give a different result.

In this project, we were not able to properly assess classification at species-level due to the insufficient dataset. To mediate this one might re-attempt running the models from this project on a different dataset or use methods such as data augmentation to make the current dataset more sufficient.

It is not uncommon for genome sequences to be incomplete, meaning the genome is divided into smaller parts, or parts of the genome are missing. In this project, testing has only been performed on

whole genomes that were highly complete. Preferably, the models should classify incomplete genomes as well as they would then be usable for a wider array of data samples. In this project, we did not test the models on incomplete data samples. This would be beneficial however, for future research. Either the models could be tested on a different dataset with less complete genomes, or we could artificially lower the quality of the dataset already in use, by manipulating the whole genome data files. Bringing in a new dataset would be simpler, and the fragmentations in the genomes would be realistic as they are real. Manipulating the current dataset would be a more arduous approach but limits the number of factors impacting the result as the genomes are the same, except incomplete. Logically, the models should perform better at incomplete data compared to models trained on marker genes, as the models trained on whole genomes would be less reliant on a specific area in the genome being preserved.

In this project, we classified microorganisms based on their DNA. However, this is not the only alternative. Historically, the first system for taxonomic classification was based on phenotype, not genotype (Hugenholtz *et al.*, 2021). Phenotype refers to traits such as appearance, behaviour, anatomy etc, while genotype refers to the genetics themselves (Austin, 2021). Today, genotype is often preferred as a basis for classification due to being more stable and genomic variance can reveal/prove evolutionary relationships. However, prokaryotes have the option of horizontal gene transfer were individuals swap genes. Such transfers can happen between prokaryotes of different species, and on rare occasions, different domains. As a result, prokaryotes can end up with misleading genes that in a purely genome-based classification system could lead to individuals being mislabelled. Classification based on phenotype, however, is also not ideal as phenotypes in prokaryotes mostly do not reflect evolutionary relationships. Either way, prokaryotes are especially difficult to classify. Ideally, classification should thus be based on several factors, such as both genotype and phenotype. Incorporating phenotypic data is outside the scope of this project. However, this combined approach may produce more robust and trustworthy classification of prokaryotes.

# Bibliography


Abadi, M. *et al.* (2015) 'TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems', p. 19.

Abiodun, O.I. *et al.* (2018) 'State-of-the-art in artificial neural network applications: A survey', *Heliyon*, 4(11), p. e00938. doi:10.1016/j.heliyon.2018.e00938.

Administrator of CD Genomics Blog (2018) '18S rRNA and Its Use in Fungal Diversity Analysis | CD Genomics Blog', *CD Genomics Blog*. Available at: https://www.cd-genomics.com/blog/18s-rrna-and-its-use-in-fungal-diversity-analysis/ (Accessed: 2 December 2020).

Albawi, S., Mohammed, T.A. and Al-Zawi, S. (2017) 'Understanding of a convolutional neural network', in *2017 International Conference on Engineering and Technology (ICET)*. *2017 International Conference on Engineering and Technology (ICET)*, pp. 1–6. doi:10.1109/ICEngTechnol.2017.8308186.

Angermueller, C. *et al.* (2016) 'Deep learning for computational biology', *Molecular Systems Biology*, 12(7), p. 878. doi:10.15252/msb.20156651.

Aun, E. *et al.* (2018) 'A k-mer-based method for the identification of phenotype-associated genomic biomarkers and predicting phenotypes of sequenced bacteria', *PLOS Computational Biology*, 14(10), p. e1006434. doi:10.1371/journal.pcbi.1006434.

Austin, C. (2021) *Phenotype*, *Genome.gov*. Available at: https://www.genome.gov/genetics-glossary/Phenotype (Accessed: 24 November 2021).

Berlin, K. *et al.* (2015) 'Assembling large genomes with single-molecule sequencing and locality-sensitive hashing', *Nature Biotechnology*, 33(6), pp. 623–630. doi:10.1038/nbt.3238.

Binieli, M. (2018) *Machine learning: an introduction to mean squared error and regression lines*, *freeCodeCamp.org*. Available at: https://www.freecodecamp.org/news/machine-learning-mean-squared-error-regression-line-c7dde9a26b93/ (Accessed: 21 October 2021).

BrainFacts/SfN (2012) *The Neuron*. Available at: https://www.brainfacts.org:443/brain-anatomy-and-function/anatomy/2012/the-neuron (Accessed: 6 October 2021).

Brandt, A. and Vilcinskas, A. (2013) 'The Fruit Fly Drosophila melanogaster as a Model for Aging Research', *Advances in Biochemical Engineering/Biotechnology*, 135, pp. 63–77. doi:10.1007/10_2013_193.

Brihadiswaran, G. (2020) 'Bioinformatics 1: K-mer Counting', *The Startup*, 2 July. Available at: https://medium.com/swlh/bioinformatics-1-k-mer-counting-8c1283a07e29 (Accessed: 5 October 2021).

Brown, C.T. and Irber, L. (2016) 'sourmash: a library for MinHash sketching of DNA', *Journal of Open Source Software*, 1(5), p. 27. doi:10.21105/joss.00027.

Brownlee, J. (2018a) 'How to Avoid Overfitting in Deep Learning Neural Networks', *Machine Learning Mastery*, 16 December. Available at: https://machinelearningmastery.com/introduction-to-regularization-to-reduce-overfitting-and-improve-generalization-error/ (Accessed: 6 October 2021).

Brownlee, J. (2018b) 'When to Use MLP, CNN, and RNN Neural Networks', *Machine Learning Mastery*, 22 July. Available at: https://machinelearningmastery.com/when-to-use-mlp-cnn-and-rnn-neural-networks/ (Accessed: 8 November 2021).

Brownlee, J. (2019a) 'A Gentle Introduction to the Rectified Linear Unit (ReLU)', *Machine Learning Mastery*, 8 January. Available at: https://machinelearningmastery.com/rectified-linear-activation-function-for-deep-learning-neural-networks/ (Accessed: 6 October 2021).

Brownlee, J. (2019b) 'How to use Data Scaling Improve Deep Learning Model Stability and Performance', *Machine Learning Mastery*, 3 February. Available at: https://machinelearningmastery.com/how-to-improve-neural-network-stability-and-modeling-performance-with-data-scaling/ (Accessed: 27 October 2021).

Brownlee, J. (2019c) 'Loss and Loss Functions for Training Deep Learning Neural Networks', *Machine Learning Mastery*, 27 January. Available at: https://machinelearningmastery.com/loss-and-loss-functions-for-training-deep-learning-neural-networks/ (Accessed: 11 October 2021).

Brownlee, J. (2019d) 'What is Deep Learning?', *Machine Learning Mastery*, 15 August. Available at: https://machinelearningmastery.com/what-is-deep-learning/ (Accessed: 18 June 2021).

Brownlee, J. (2020a) 'Introduction to Dimensionality Reduction for Machine Learning', *Machine Learning Mastery*, 5 May. Available at: https://machinelearningmastery.com/dimensionality-reduction-for-machine-learning/ (Accessed: 3 February 2021).

Brownlee, J. (2020b) 'Why Data Preparation Is So Important in Machine Learning', *Machine Learning Mastery*, 14 June. Available at: https://machinelearningmastery.com/data-preparation-is-important/ (Accessed: 5 October 2021).

Bucak, I. and Uslan, V. (2011) 'Sequence alignment from the perspective of stochastic optimization: A survey', *Turkish J of Electrical Engineering & Computer Sciences*, 19. doi:10.3906/elk-1002-410.

Carrieri, A.P. *et al.* (2019) 'A Fast Machine Learning Workflow for Rapid Phenotype Prediction from Whole Shotgun Metagenomes', *Proceedings of the AAAI Conference on Artificial Intelligence*, 33(01), pp. 9434–9439. doi:10.1609/aaai.v33i01.33019434.

Chaudhary, M. (2020) 'Activation Functions: Sigmoid, Tanh, ReLU, Leaky ReLU, Softmax', *Medium*, 28 August. Available at: https://medium.com/@cmukesh8688/activation-functions-sigmoid-tanh-relu-leaky-relu-softmax-50d3778dcea5 (Accessed: 6 October 2021).

Chauhan, N. (2020) 'Optimization Algorithms in Neural Networks', *KDnuggets*. Available at: https://www.kdnuggets.com/optimization-algorithms-in-neural-networks.html/ (Accessed: 7 October 2021).

Chicco, D. (2017) 'Ten quick tips for machine learning in computational biology', *BioData Mining*, 10(1), p. 35. doi:10.1186/s13040-017-0155-3.

Chollet, F. (2015) *Keras*. Available at: https://keras.io/ (Accessed: 18 November 2021).

Chor, B. *et al.* (2009) 'Genomic DNA k-mer spectra: models and modalities', *Genome Biology*, 10(10), p. R108. doi:10.1186/gb-2009-10-10-r108.

Clavijo, B. (2018) *k-mer counting, part I: Introduction | BioInfoLogics*. Available at: https://bioinfologics.github.io/post/2018/09/17/k-mer-counting-part-i-introduction/ (Accessed: 26 October 2021).

Diaz, R. (2016) 'Re: What is the pros and cons of Convolutional networks?' Available at: https://www.researchgate.net/post/What-is-the-pros-and-cons-of-Convolutional-neural-networks/56deedc3b0366d37691aba73/citation/download.

Doshi, S. (2020) *Various Optimization Algorithms For Training Neural Network*, *Medium*. Available at: https://towardsdatascience.com/optimizers-for-training-neural-network-59450d71caf6 (Accessed: 7 October 2021).

Edgar, R.C. (2016) 'SINTAX: a simple non-Bayesian taxonomy classifier for 16S and ITS sequences', *bioRxiv*, p. 074161. doi:10.1101/074161.

*Genome Taxonomy Database* (2021). Available at: https://gtdb.ecogenomic.org/ (Accessed: 18 October 2021).

Gittleman, J.L. (2016) *Phylogeny*, *Encyclopedia Britannica*. Available at: https://www.britannica.com/science/phylogeny (Accessed: 2 December 2020).

gmarcais (2021) *Jellyfish*. Available at: https://github.com/gmarcais/Jellyfish (Accessed: 18 October 2021).

Gordon-Rodriguez, E. *et al.* (2020) 'Uses and Abuses of the Cross-Entropy Loss: Case Studies in Modern Deep Learning', *arXiv:2011.05231 [cs, stat]* [Preprint]. Available at: http://arxiv.org/abs/2011.05231 (Accessed: 28 November 2021).

Gupta, S. (2020) *Advantages and Disadvantages of Artificial Neural Networks*, *Asquero*. Available at: https://www.asquero.com/article/advantages-and-disadvantages-of-artificial-neural-networks (Accessed: 8 November 2021).

Hammerstrom, D. (1993) 'Neural networks at work', *IEEE Spectrum*, 30(6), pp. 26–32. doi:10.1109/6.214579.

Hornik, K., Stinchcombe, M. and White, H. (1989) 'Multilayer feedforward networks are universal approximators', *Neural Networks*, 2(5), pp. 359–366. doi:10.1016/0893-6080(89)90020-8.

Htoon, K.S. (2020) *Log Transformation: Purpose and Interpretation*, *Medium*. Available at: https://medium.com/@kyawsawhtoon/log-transformation-purpose-and-interpretation-9444b4b049c9 (Accessed: 1 February 2021).

Hugenholtz, P. *et al.* (2021) 'Prokaryotic taxonomy and nomenclature in the age of big sequence data', *The ISME Journal*, 15(7), pp. 1879–1892. doi:10.1038/s41396-021-00941-x.

Jaillard, M. *et al.* (2018) 'A fast and agnostic method for bacterial genome-wide association studies: Bridging the gap between k-mers and genetic events', *PLOS Genetics*, 14(11), p. e1007758. doi:10.1371/journal.pgen.1007758.

Kaehler, R. (2017) 'K-MER ANALYSIS PIPELINE FOR CLASSIFICATION OF DNA SEQUENCES FROM METAGENOMIC SAMPLES', p. 122.

Khawaldeh, S. *et al.* (2017) 'Taxonomic Classification for Living Organisms Using Convolutional Neural Networks', *Genes*, 8(11), p. 326. doi:10.3390/genes8110326.

Kim, D. *et al.* (2016) 'Centrifuge: rapid and sensitive classification of metagenomic sequences', *Genome Research* [Preprint]. doi:10.1101/gr.210641.116.

Knight, R. *et al.* (2018) 'Best practices for analysing microbiomes', *Nature Reviews Microbiology*, 16(7), pp. 410–422. doi:10.1038/s41579-018-0029-9.

Koonin, E. (2010) *The Two Empires and Three Domains of Life in the Postgenomic Age*, *Nature*. Available at: https://www.nature.com/scitable/topicpage/the-two-empires-and-three-domains-of-14432998/ (Accessed: 21 November 2021).

LaPierre, N. *et al.* (2019a) 'MetaPheno: A critical evaluation of deep learning and machine learning in metagenome-based disease prediction', *Methods*, 166, pp. 74–82. doi:10.1016/j.ymeth.2019.03.003.

LaPierre, N. *et al.* (2019b) 'MetaPheno: A critical evaluation of deep learning and machine learning in metagenome-based disease prediction', *Methods*, 166, pp. 74–82. doi:10.1016/j.ymeth.2019.03.003.

Larrañaga, P. *et al.* (2006) 'Machine learning in bioinformatics', *Briefings in Bioinformatics*, 7(1), pp. 86–112. doi:10.1093/bib/bbk007.

Lee, A. (2017) 'Lecture 5: Data Preprocessing, Imputation and Feature Engineering', p. 40.

Liang, Q. *et al.* (2020) 'DeepMicrobes: taxonomic classification for metagenomics with deep learning', *NAR Genomics and Bioinformatics*, 2(1). doi:10.1093/nargab/lqaa009.

Lin, M. *et al.* (2017) 'Effects of short indels on protein structure and function in human genomes', *Scientific Reports*, 7(1), p. 9313. doi:10.1038/s41598-017-09287-x.

Lu, J. and Salzberg, S.L. (2020) 'Ultrafast and accurate 16S microbial community analysis using Kraken 2', *bioRxiv*, p. 2020.03.27.012047. doi:10.1101/2020.03.27.012047.

Marçais, G. and Kingsford, C. (2011) 'A fast, lock-free approach for efficient parallel counting of occurrences of k-mers', *Bioinformatics*, 27(6), pp. 764–770. doi:10.1093/bioinformatics/btr011.

Menzel, P., Ng, K.L. and Krogh, A. (2016) 'Fast and sensitive taxonomic classification for metagenomics with Kaiju', *Nature Communications*, 7(1), p. 11257. doi:10.1038/ncomms11257.

Mishra, A. (2020) *Metrics to Evaluate your Machine Learning Algorithm*, *Medium*. Available at: https://towardsdatascience.com/metrics-to-evaluate-your-machine-learning-algorithm-f10ba6e38234 (Accessed: 10 November 2021).

Mody, P. (2020) 'Loss Functions for Computer Vision Models', *Medium*, 28 December. Available at: https://prerakmody.medium.com/loss-functions-for-computer-vision-models-8b4f7578766f (Accessed: 28 November 2021).

Ng, A. (2015) *Why should you care about deep learning?* Available at: https://www.youtube.com/watch?v=O0VN0pGgBZM (Accessed: 16 March 2021).

Nguyen, N.G. *et al.* (2016) 'DNA Sequence Classification by Convolutional Neural Network', *Journal of Biomedical Science and Engineering*, 09(05), p. 280. doi:10.4236/jbise.2016.95021.

Official Greengenes database website (2020) *The Greengenes Database*, *Greengenes*. Available at: https://greengenes.secondgenome.com/ (Accessed: 2 December 2020).

Official SILVA Website (2020) *Documentation*. Available at: https://www.arb-silva.de/documentation/ (Accessed: 2 December 2020).

Ondov, B.D. *et al.* (2016) 'Mash: fast genome and metagenome distance estimation using MinHash', *Genome Biology*, 17(1), p. 132. doi:10.1186/s13059-016-0997-x.

Ounit, R. *et al.* (2015) 'CLARK: fast and accurate classification of metagenomic and genomic sequences using discriminative k-mers', *BMC Genomics*, 16(1), p. 236. doi:10.1186/s12864-015-1419-2.

Paine, T. *et al.* (2013) 'GPU Asynchronous Stochastic Gradient Descent to Speed Up Neural Network Training', *arXiv:1312.6186 [cs]* [Preprint]. Available at: http://arxiv.org/abs/1312.6186 (Accessed: 17 November 2021).

Parks, D.H. *et al.* (2020) 'A complete domain-to-species taxonomy for Bacteria and Archaea', *Nature Biotechnology*, 38(9), pp. 1079–1086. doi:10.1038/s41587-020-0501-8.

Parks, D.H. *et al.* (2021) 'GTDB: an ongoing census of bacterial and archaeal diversity through a phylogenetically consistent, rank normalized and complete genome-based taxonomy', *Nucleic Acids Research* [Preprint], (gkab776). doi:10.1093/nar/gkab776.

Pei, A. *et al.* (2009) 'Diversity of 23S rRNA Genes within Individual Prokaryotic Genomes', *PLoS ONE*, 4(5). doi:10.1371/journal.pone.0005437.

Pradhan, D. *et al.* (2019) 'Chapter 4 - High-throughput sequencing', in Misra, G. (ed.) *Data Processing Handbook for Complex Biological Data Sources*. Academic Press, pp. 39–52. doi:10.1016/B978-0-12-816548-5.00004-6.

Pykes, K. (2020) *The Vanishing/Exploding Gradient Problem in Deep Neural Networks*, *Medium*. Available at: https://towardsdatascience.com/the-vanishing-exploding-gradient-problem-in-deep-neural-networks-191358470c11 (Accessed: 6 October 2021).

Saxena, S. (2018) 'Precision vs Recall', *Medium*, 13 May. Available at: https://medium.com/@shrutisaxena0617/precision-vs-recall-386cf9f89488 (Accessed: 12 November 2021).

Seif, G. (2021) *Understanding the 3 most common loss functions for Machine Learning Regression*, *Medium*. Available at: https://towardsdatascience.com/understanding-the-3-most-common-loss-functions-for-machine-learning-regression-23e0ef3e14d3 (Accessed: 21 October 2021).

Shen, K. (2018) 'Effect of batch size on training dynamics', *Mini Distill*, 19 June. Available at: https://medium.com/mini-distill/effect-of-batch-size-on-training-dynamics-21c14f7a716e (Accessed: 6 October 2021).

Shreiner, A.B., Kao, J.Y. and Young, V.B. (2015) 'The gut microbiome in health and in disease', *Current opinion in gastroenterology*, 31(1), pp. 69–75. doi:10.1097/MOG.0000000000000139.

Shrivastava, S. (2019) *Anomaly detection on a categorical and continuous dataset*, *Medium*. Available at: https://medium.com/@shreyash0023/anomaly-detection-on-a-categorical-and-continuous-dataset-d5af7aa287d2 (Accessed: 1 February 2021).

Sigma2/NRIS (2021) *Saga — Sigma2 documentation*. Available at: https://documentation.sigma2.no/hpc_machines/saga.html (Accessed: 18 October 2021).

Sikka, M. (2020) *Balancing the Regularization Effect of Data Augmentation*, *Medium*. Available at: https://towardsdatascience.com/balancing-the-regularization-effect-of-data-augmentation-eb551be48374 (Accessed: 6 October 2021).

Tarca, A.L. *et al.* (2007) 'Machine Learning and Its Applications to Biology', *PLOS Computational Biology*, 3(6), p. e116. doi:10.1371/journal.pcbi.0030116.

Tegmark, M. (2017) 'Life 3.0: being human in the age of artificial intelligence', in. New York: Alfred A. Knopf, pp. 71–81.

The Editors of Encyclopedia Britannica (2020) *Genetic marker*, *Encyclopedia Britannica*. Available at: https://www.britannica.com/science/genetic-marker (Accessed: 2 December 2020).

The GTDB Team (2021) *GTDB Data - /releases/release202/*. Available at: https://data.gtdb.ecogenomic.org/releases/release202/ (Accessed: 18 October 2021).

'Tree of life (biology)' (2021) *Wikipedia*. Available at: https://en.wikipedia.org/w/index.php?title=Tree_of_life_(biology)&oldid=1045851568 (Accessed: 18 November 2021).

*Uninett Sigma2* (2021). Available at: https://www.sigma2.no/ (Accessed: 18 October 2021).

Uniqtech (2019) 'Multilayer Perceptron (MLP) vs Convolutional Neural Network in Deep Learning', *Data Science Bootcamp*, 13 June. Available at: https://medium.com/data-science-bootcamp/multilayer-perceptron-mlp-vs-convolutional-neural-network-in-deep-learning-c890f487a8f1 (Accessed: 8 November 2021).

Versloot, C. (2019) 'What is Dropout? Reduce overfitting in your neural networks', *MachineCurve*, 16 December. Available at: https://www.machinecurve.com/index.php/2019/12/16/what-is-dropout-reduce-overfitting-in-your-neural-networks/ (Accessed: 6 October 2021).

Vilone, G. and Longo, L. (2021) 'A Quantitative Evaluation of Global, Rule-Based Explanations of Post-Hoc, Model Agnostic Methods', *Frontiers in Artificial Intelligence*, 4, p. 717899. doi:10.3389/frai.2021.717899.

Vinje, H. *et al.* (2015) 'Comparing K-mer based methods for improved classification of 16S sequences', *BMC Bioinformatics*, 16(1), p. 205. doi:10.1186/s12859-015-0647-4.

Vu, D., Groenewald, M. and Verkley, G. (2020) 'Convolutional neural networks improve fungal classification', *Scientific Reports*, 10(1), p. 12628. doi:10.1038/s41598-020-69245-y.

Wang, C.-F. (2019) *The Vanishing Gradient Problem*, *Medium*. Available at: https://towardsdatascience.com/the-vanishing-gradient-problem-69bf08b15484 (Accessed: 6 October 2021).

Wang, H., Ma, C. and Zhou, L. (2009) 'A Brief Review of Machine Learning and Its Application', in *2009 International Conference on Information Engineering and Computer Science*. *2009 International Conference on Information Engineering and Computer Science*, pp. 1–4. doi:10.1109/ICIECS.2009.5362936.

Woese, C.R. and Fox, G.E. (1977) 'Phylogenetic structure of the prokaryotic domain: the primary kingdoms', *Proceedings of the National Academy of Sciences of the United States of America*, 74(11), pp. 5088–5090. doi:10.1073/pnas.74.11.5088.

Wood, D.E., Lu, J. and Langmead, B. (2019) 'Improved metagenomic analysis with Kraken 2', *Genome Biology*, 20(1), p. 257. doi:10.1186/s13059-019-1891-0.

Wood, T. (2019) *Softmax Function*, *DeepAI*. Available at: https://deepai.org/machine-learning-glossary-and-terms/softmax-layer (Accessed: 11 October 2021).

Yadav, D. (2019) *Categorical encoding using Label-Encoding and One-Hot-Encoder*, *Medium*. Available at: https://towardsdatascience.com/categorical-encoding-using-label-encoding-and-one-hot-encoder-911ef77fb5bd (Accessed: 1 February 2021).

Yang, D. *et al.* (2017) 'HistoSketch: Fast Similarity-Preserving Sketching of Streaming Histograms with Concept Drift', in, pp. 545–554. doi:10.1109/ICDM.2017.64.

Yarza, P. *et al.* (2014) 'Uniting the classification of cultured and uncultured bacteria and archaea using 16S rRNA gene sequences', *Nature Reviews Microbiology*, 12(9), pp. 635–645. doi:10.1038/nrmicro3330.

Ye, S.H. *et al.* (2019) 'Benchmarking Metagenomics Tools for Taxonomic Classification', *Cell*, 178(4), pp. 779–794. doi:10.1016/j.cell.2019.07.010.