Master's Thesis 2022 30 ECTS
Faculty of Science and Technology

Evaluation of Machine Learning Approaches for Prediction of Protein Coding Genes in Prokaryotic DNA Sequences

Yva Jacob Sandvik
Data Science

Preface

This thesis finalises my master's degree in Data Science at the Norwegian University of Life Sciences (NMBU). The topic was introduced by Kristian Hovde Liland, associate professor in Data Science at NMBU. A preliminary literature review was written during the autumn of 2021, also supervised by Kristian Hovde Liland. The work on this thesis started in January 2022 and is a continuation of the results obtained from the review. In hindsight, it has been a challenging semester, with some hurdles along the way, but also surprisingly fun, with some unforeseen new friendships at TF1-212. Most of all, it has been extremely educational.

There are several people I would like to thank for helping me through the different stages of writing this thesis. I wish to express my deepest appreciation to my supervisor, Kristian Hovde Liland, and my co-supervisor, Lars Snipen, for their knowledge, advice and time, throughout the whole process. It has been a pleasure to be guided by them, and all our meetings and discussions have both motivated me and contributed greatly to this thesis. I would also like to thank my family, for their encouragement and proofreading. A special thanks to my brother Yohann, for his remarkable willingness to help and guide, and my boyfriend Ola, for his patience and support.

Finally, I would like to thank everyone else who has made the last six years at NMBU an unforgettable experience.

[illegible]

Yva Jacob Sandvik
Ås, June 13, 2022

Abstract

According to the National Human Genome Research Institute the amount of genomic data generated on a yearly basis is constantly increasing. This rapid growth in genomic data has led to a subsequent surge in the demand for efficient analysis and handling of said data. Gene prediction involves identifying the areas of a DNA sequence that code for proteins, also called protein coding genes. This task falls within the scope of bioinformatics, and there has been surprisingly little development in this field of study, over the past years. Despite there being sufficient state-of-the-art gene prediction tools, there is still room for improvement in terms of efficiency and accuracy. Advances made within the field of gene prediction can, among other things, aid the medical and pharmaceutical industry, as well as impact environmental and anthropological research.

Machine learning techniques such as the Random Forest classifiers and Artificial Neural Networks (ANN) have proved successful at the task of gene prediction. In this thesis one deep learning model and two other machine learning models were tested. The first model implemented was the established Random Forest classifier. When it comes to the use of ensemble methods, such as the Random Forest classifier, feature engineering is critical for the success of such models. The exploration of different feature selection and extraction techniques underpinned its relevance. It also showed that feature importance varies greatly among genomes, and revealed possibilities that can be further explored in future work. The second model tested was the ensemble method Extreme Gradient Boosting (XGBoost), which served as a good competitor to the Random Forest classifier. Finally, a Recurrent Neural Network (RNN) was implemented. RNNs are known to be good with handling sequential data, therefore it seemed like a good candidate for gene prediction.

The 15 prokaryotic genomes used to train the models were extracted from the NCBI genome database. Each model was tasked with classifying sub-sequences of the genomes, called open reading frames (ORFs), as either protein coding ORFs, or non-coding ORFs. One challenge when preparing these datasets was that the number of protein coding ORFs was very small compared to the number of non-coding ORFs. Another problem encountered in the dataset was that protein coding ORFs in general are longer than non-coding ORFs, which can bias the models to simply classify long ORFs as protein coding, and short ORFs as non-coding. For these reasons, two datasets for each genome were created, taking each imbalance into account. The models were trained, tuned and tested on both datasets for all genomes, and a combination of genomes. The models were evaluated with regard to accuracy, precision and recall.

The results show that all three methods have potential and attained somewhat similar performance scores. Despite the fact that both time and data were limited during model development, they still yielded promising results. Considering there are several parameters that have not yet been tuned in all models, many possibilities for further research remain. The fact that a relatively simple RNN architecture performed so well, and has no requirement for feature engineering, shows great promise for further applications in gene prediction, and possibly other fields in bioinformatics.

Contents

List of Figures

List of Tables

Abbreviations and Acronyms

XGBoost	Extreme Gradient Boosting
RNN	Recurrent Neural Network
ORF	Open Reading Frame
CDS	Protein Coding Genes (CoDing Sequences)
LORF	Longest ORF
n-LORFs	Negative LORFs
DNA	Deoxyribonucleic acid
A	Adenine
T	Thymine
G	Guanine
C	Cytosine
ANN	Artificial Neural Network
MLP	Multi-layer Perceptron
CNN	Convolutional Neural Network
TP	True Positive
TN	True Negative
FP	False Positive
FN	False Negative
IG	Information Gain
GD	Gradient Descent
ReLU	Rectified Linear Unit
SGD	Stochastic Gradient Descent
RMSProp	Root Mean Square Propagation
ADAM	Adaptive Moment Estimation
NLP	Natural Language Processing
LSTM	Long Short-Term Memory
GRU	Gated Recurrent Units

Chapter 1

Introduction

A genome is the entire set of genetic material found in an organism. Genomics is a field within biology that studies the structure, function and the evolution of genomes [1]. The field aims to characterise and quantify genes, determine how a genome makes up an organism, and how genetic material may be affected by the environment. There are several different types of genomics such as functional genomics, structural genomics, meta-genomics, etc. All these interdisciplinary fields have in common that they produce large amounts of data. This data requires statistical and computational tools and knowledge to be processed, stored and interpreted. The task of handling such data falls within the field of *Bioinformatics*. Bioinformatics can be described as a combination of statistics, biology and computer science. These are skills that are increasingly in demand as a result of the rapid technological development and Big data production that has occurred over the past decade.

Genomic analysis involves preparing DNA samples, sequencing these samples, and then analysing and interpreting the sequenced DNA. Gene prediction takes place during the analysis step, and involves identifying the areas of DNA that encode a gene. A gene product can be either RNA or protein. The genes that code for proteins are referred to as the protein coding genes, while the genes that code fore RNA are non-coding genes. There are also areas of a DNA sequence that are purely regulatory or contain no genes at all. These regions are called the intergenic regions. Identifying and distinguishing these different regions from one another is a challenging task that often requires the use of computational tools. Many of the state-of-the-art gene prediction tools rely on the use of Hidden Markov Models, while the tools developed in more recent years are largely based on machine learning methods.

1.1 Motivation

Gene prediction is a crucial step in most genomics pipelines. By identifying which areas of a genome's DNA that are protein coding genes, one may proceed with decoding them and figure out what proteins they code for. This in turn can give us information about structural and functional properties of an organism. Genomics can be applied in several fields such as medicine, pharmaceutical industry, epidemiology, environmental surveys and anthropology. Advances in gene prediction can, for instance, aid in genetic disorder detection, treatments involving directed therapy, and identifying particular species in metagenome samples. Despite the importance of gene prediction, research regarding computational tools that address this stage in the genomics pipeline has stagnated over the last few years.

The papers written by Silva et al. [2] and Al-Ajlan and El Allali [3] confirm that machine learning and deep learning models are applicable for the purpose of gene prediction. The results from both papers are promising, although the GeneRFinder tool, which makes use of the *Random Forest classifier*, so far shows the best performance. In addition to these two studies there are other published papers and tools that make use of machine learning methods such as Support Vector Machines [4], *Artificial Neural Networks (ANN)* [5], Generalised Linear Models [6] and other ensemble methods. Even though several machine learning methods have been tried already, there are still a number of untested methods and unanswered analytical questions.

1.2 Objectives

The main objective of this thesis has been to evaluate the performance of three machine learning methods, for the prediction of protein coding genes in prokaryotic DNA sequences, when trained on the same datasets. This was done to determine whether the implemented models have potential to perform as well as, or better than, tools such as GeneRFinder or other state-of-the art gene prediction tools.

The first model implemented was a Random Forest classifier, heavily inspired by the GeneRFinder tool detailed in Silva et al. [2]. The incentive for implementing this model was to explore different feature engineering and feature selection techniques and see how the model responded to different training datasets. The second model implemented was the popular ensemble method *Extreme Gradient Boosting (XGBoost)*. The final model implemented is a *Recurrent Neural Network (RNN)*. RNNs are known to be particularly effective when working with sequential data. Despite this, it does not seem to have been explored yet for the purpose of gene prediction.

The objectives of this thesis can be summarised in the form of three questions:

1. What selection of predefined and engineered features yield the highest performance of a Random Forest classifier?
2. How does the XGBoost classifier perform at predicting protein coding genes in prokaryotic DNA?
3. Are RNNs applicable for predicting protein coding genes in prokaryotic DNA, and how do they compare to the ensemble methods Random Forest and XGBoost?

1.3 Structure

The remaining chapters will be arranged in the following manner: Chapter 2 will cover the theory of gene prediction and machine learning models and summarise related work considered particularly relevant. Chapter 3 presents the genome data used to develop the machine learning models, how this data was selected, and a description of the datasets. Chapter 4 details how the models were implemented, trained and tested. Chapter 5 presents and compares the results of each model. Finally, Chapter 6 and 7 discusses and draws a conclusion from the results.

Chapter 2

Theory

This chapter will cover theory on gene prediction, the genetic code, the machine learning models implemented, and a brief review of literature considered particularly relevant for this thesis. A preliminary literature review was done in preparation for this thesis. It involved reviewing literature on state-of-the-art gene prediction methods, identifying their differences and challenges, and reflecting around future directions. The conclusion to the literature review was the starting point for the work on this thesis, and the literature review can therefore be found in Appendix C. Theory regarding DNA sequencing, genes, state-of-the-art gene prediction tools and other structures surrounding the genetic code is explained in greater detail in the preliminary study and will therefore only briefly be explained in this theory chapter.

2.1 Gene Prediction

When referring to *genes* one means sequences of *Deoxyribonucleic acid (DNA)* found within a DNA strand that either encodes protein or RNA. The DNA sequences that code for protein are referred to as *coding DNA* or *protein coding genes (CoDing Sequences - CDS)*. The remaining part of the DNA is referred to as *non-coding DNA*. This includes genes that code for RNA or regulatory regions for instance. Only prokaryotic genomes are used in this thesis, and these genomes are much smaller than eukaryotic genomes. The coding genes in prokaryotic genomes are found close to each other, and often even overlap in a sequence. There are no *introns* in prokaryotic DNA, as opposed to eukaryotic DNA. Introns are large non-coding regions of eukaryotic DNA found between the coding regions that are referred to as *exons*. In this case one is only interested in identifying the protein coding genes in prokaryotic DNA.

The term *gene prediction* refers to the process of identifying CDS in genetic material. Methods used to identify genes in DNA sequences can be divided into *extrinsic* and *intrinsic* approaches. The extrinsic approach, also called *similarity based searches*, is simply an approach that involves comparing sequenced DNA fragments to a database of already annotated genes. By relying only on existing gene databases to make new gene predictions there is a risk of being affected by historic biases. However, this method can be efficient when wishing to identify RNA genes as they tend to be well conserved across organisms. Protein coding genes on the other hand, evolve fast and tend to be quite different from one organism to the other.

The intrinsic approaches, also called *ab-initio* methods, identify genes by signal detection and by using gene structure to create rules and patterns for classification. These methods do not rely on external databases while finding the genes, but they do require a database of already annotated genes to control the accuracy of the gene prediction methods. Hence, the problem of historic bias is hard to avoid.

State-of-the-art gene prediction tools are intrinsic approaches based on methods such as Hidden-Markov models and dynamic programming. Newer tools make use of machine learning models such as Random Forest and Neural Networks. So far the Random Forest model seems to be the strongest competitor to popular state-of-the-art tools such as *Prodigal* [2].

2.1.1 The Genetic Code

A DNA strand is a double helical structure, composed of two intertwined polynucleotide chains, carrying all the genetic material for organisms and many viruses. Each polynucleotide chain is made up of *nucleotides*. A single nucleotide consists of a sugar molecule that is attached to a phosphate group and a *nitrogen-containing base*. These nucleotides are then bound together to form a sugar-phosphate backbone, which forms the polynucleotide chain. There are four different nucleotide bases in DNA: *adenine (A)*, *cytosine (C)*, *guanine (G)* and *thymine (T)*. The polynucleotide chains in DNA are held together by the bonds between the bases. A forms a complementary base pair with T, and G forms a complementary base pair with C. Figure 2.1 shows the structure of a double helix DNA strand, and how the chains are bound together by complementary base pairs.

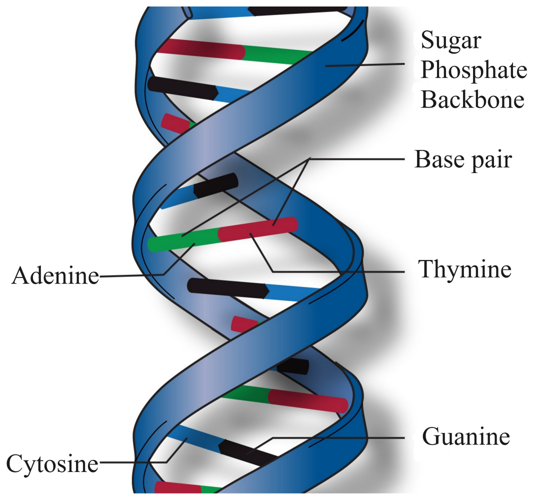


Figure 2.1: Illustration of how two polynucleotide chains are intertwined to form a double helix DNA strand. The complementary base pairing between A-T and G-C are responsible for the coiling of the two chains. The figure is taken from NIH [7].

The genetic code is the term used to explain how sequences of the bases A, T, C and G in genetic material, determine if a DNA sequence codes for proteins or not. During protein synthesis the information given by a DNA sequence is read three bases at a time. A triplet of consecutive base pairs are called a *codon*. Each codon codes for a particular amino acid, which are the building blocks of a protein. However, different codons can correspond to the same amino acid, as seen in Table 2.1. Since there are four bases and three of them make up a codon, there exists 64 unique combinations of these. All 64 codons are found in Table 2.1.

When trying to identify CDS one often starts by identifying the *open reading frames (ORFs)* in a genome. An open reading frame is a DNA sequence that is enclosed by a start and stop codon in the same *reading frame*. A DNA sequence's reading frame is the set of consecutive and non-overlapping codons that follow a given starting point. A start codon can mark the beginning of a gene and a stop codon marks the termination. An ORF can contain multiple start codons, but only one stop codon. A single ORF can contain several shorter ORFs within itself, that all share the same stop codon. A start codon can also simply function as any other codon by coding for an amino acid, without necessarily indicating the start of a new gene. The stop codon on the other hand always codes for termination and not an amino acid. In prokaryotes there are three possible start codons, these are marked in bold in Table 2.1. In

Table 2.1: Table of amino acids, their symbols and the respective codons they are encoded by. The codons highlighted in bold are prokaryotic start codons.

Amino Acid	Symbol	DNA Codons
Alanine	A	GCA GCC GCG GCT
Cystenine	C	TGC TGT
Aspartic Acid	D	GAC GAT
Glutamic Acid	E	GAA GAG
Phenylalanine	F	TTC TTT
Glycine	G	GGA GGC GGG GGT
Histidine	H	CAC CAT
Isoleucine	I	ATA ATC ATT
Lysine	K	AAA AAG
Leucine	L	CTA CTC CTG CTT TTA **TTG**
Methionine	M	**ATG**
Asparagine	N	AAC AAT
Proline	P	CCA CCC CCG CCT
Glutamine	Q	CAA CAG
Arginine	R	AGA AGG CGA CGC CGG CGT
Serine	S	AGC AGT TCA TCC TCG TCT
Threonine	T	ACA ACC ACG ACT
Valine	V	GTA GTC **GTG** GTT
Tryptophan	W	TGG
Tyrosine	Y	TAC TAT
STOP	*	TAA TAG TGA

eukaryotes there is only one: ATG. Since prokaryotic genome data is used in this case, all three start codons are relevant. There does exist rare examples of annotated genes that have alternative start codons to these three [8], but in this thesis however, we will only focus on what we expect to see in most sequences.

An important distinction is that all ORFs found in strands of sequenced DNA are not necessarily genes. In a given ORF there may exist many nested ORFs within the same reading frame. Nested ORFs arise due to the possibility of there being many start-codons to a given stop-codon. The start-codon which is at the most upstream position to the matching stop-codon is considered the *longest ORF (LORF)*. LORFs can either contain coding genes or they can be non-coding. Non-coding LORFs make up the intergenic regions of the genome, and coding LORFs are what one commonly refers to as protein coding genes. In this thesis, non-coding LORFs will be referred to as *negative LORFs (n-LORFs)* and coding LORFs as CDS.

2.2 Machine Learning

Machine learning is a subset of artificial intelligence, and deep learning is a subset of machine learning. In this thesis, two machine learning models and one deep learning model have been implemented. The machine learning models will be presented and explained in this section.

Artificial intelligence is the most general term of the three and refers to intelligence demonstrated by machines that mimic aspects of human intelligence. Machine learning involves the ability of machines to learn from data, then classify and predict outcomes without the use of predefined rules. The algorithms instead use training data to build models by inferring their own rules which are used to make predictions. Machine learning has a wide application basis and can, for example, be used for image analysis, speech recognition, language processing, and for making predictions within the fields of big data analysis and medicine [9].

There are three main categories within machine learning, and these are *supervised learning*, *unsupervised learning*, and *reinforcement learning.* In this thesis, the focus will be on supervised machine learning methods. Supervised learning requires the input data to be labelled to make a classification or numerical estimation. This means a manual job of labelling datasets must be done before feeding the data to a model. Unsupervised learning on the other hand are methods that can discover patterns and distinguishing characteristics without the need of a labelled dataset [10]. Reinforcement learning involves training a model based on rewarding and/or punishing the models behaviours in a given environment. In simpler words, it is a method that learns from its mistakes. It does not rely on labelled training data, but instead makes sequential decisions based on the models ”reinforcement agent”.

Scikit-learn

The scikit-learn version used to implement the machine learning models is **0.23.2** [11]. Scikit-learn is an open source python software library used to implement various machine learning models. Each version can include different parameter updates and default values.

2.2.1 Decision Trees

Since both Random Forest and XGBoost rely heavily on the use of decision trees, it is necessary to have a good understanding of how the decision tree algorithm works. Figure 2.2 is a simple illustration of a decision tree. At each step in the tree one wishes to find a feature that can be used to partition the data in such a way that it minimises the entropy of the data at the next step. This is called a greedy algorithm, as it uses a local optimisation criteria instead of a global one.

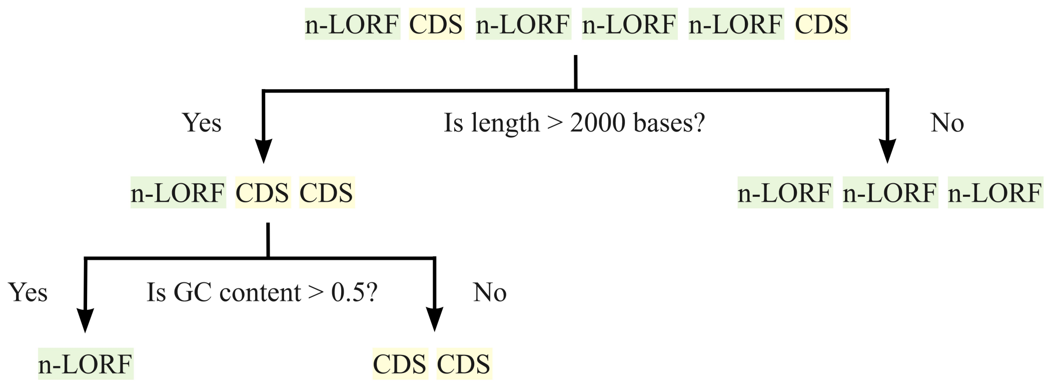


Figure 2.2: Illustration of how a single decision tree works. The goal is to separate CDS from n-LORFs based on the distinguishing characteristics’ ”length” and ”GC-content”.

The decision tree in Figure 2.2 is a simple illustration of the logic behind a decision tree. It attempts to split the input data into pure nodes containing either CDS or n-LORFs, based on distinguishing characteristics. The two features that were chosen to separate the ORFs by are sequence "length" and "GC-content". At the root node, the input data is split into two groups depending on whether or not the sequence has a length greater than 2000 bases. After this split, one of the resulting two nodes is a pure leaf node, containing only n-LORFs. The other resulting node contains two CDS and one n-LORF, so it is not pure yet. Separating the two sequences based on "length" turned out to be quite efficient. The node that is not pure is split again based on a another feature, "GC-content". This time both the resulting nodes are pure leaf nodes, and one has successfully separated the input data into CDS and n-LORFs. When the decision tree is given new data it uses the features (length and GC-content) and thresholds specified during training to classify the data. Real datasets are much more complex, and the decision trees will have many more nodes. As long as the tree has not yet achieved only pure leaf nodes it will continue to generate nodes until it has reached the maximum depth. The maximum depth is a limiting parameters that control how many splits the tree can make before it has to stop. There are other limiting parameters such as minimum sample split and the minimum number of samples required to be at a leaf node, which control the splitting of nodes in a decision tree. These will be described more in Chapter 4. It is important to control these type of parameters in order to avoid overly complex models and overfitting, or too simple models and underfitting.

The goal at each node of a decision tree is to identify a feature that will give a split resulting in two new nodes that are as different from each other as possible, while the members of each node are as similar to each other as possible. Decision trees evaluate different features' ability to obtain this by comparing the potential *information gain (IG)* after a split. Information gain is calculated based on the impurity measures. There are two impurity measures often used in binary decision trees called *Gini impurity* and *entropy.* After calculating the impurity measure for features at a given node, this value can be used to calculate the information gain and determine which feature is more appropriate for making the split.

Gini Impurity

The default impurity measure in a decision tree classifier implemented using scikit-learn, is Gini impurity. Since this was the library used to implement the Random Forest model, the logic behind Gini impurity will be explained briefly. Equation 2.1 shows how one calculates the Gini impurity at an arbitrary node. D denotes the input dataset which has k number of classes. The probability of samples belonging to class i at a particular node is denoted as p_i [12].

$$Gini(D) = 1 - \sum_{i=1}^{k} p_i^2 \tag{2.1}$$

The features that have the smallest Gini impurity are the ones selected for splitting a node. In other words, situations where there are large differences between the probabilities are favoured. Equation 2.2 shows how one calculates the Gini impurity if dataset D is split by feature A into two leaf nodes with the datasets D_1 and D_2, of size n_1 and n_2 respectively. n denotes the size of the parent node dataset D, before the split. In other words, one simply adds the weighted average of the Gini impurity at each new leaf node to get the overall Gini impurity provided by feature A [12].

$$Gini_A(D) = \frac{n_1}{n} Gini(D_1) + \frac{n_2}{n} Gini(D_2) \tag{2.2}$$

Finally Equation 2.3 shows how one calculates information gain for feature A. This is simply done by subtracting the weighted Gini impurities from each leaf node, $Gini_A(D)$, from the Gini impurity of the parent node, $Gini(D)$.

$$IG_A = Gini(D) - Gini_A(D) \tag{2.3}$$

2.2.2 Random Forest

Random forest is a supervised machine learning method used for classification and regression, and is categorised as an ensemble learning method. This means the method combines multiple models or algorithms to solve a computational problem. Random forest combines numerous decision trees and uses bagging and feature randomness when constructing trees during training [13]. The bagging method ensures that one is not using the same data for every tree, but rather a random sample. This helps the model to generalise beyond the training data, and combat overfitting. The random feature selection helps to reduce the correlation between the trees. If every feature was used for each tree, most of the trees would have the same decision nodes and act very similarly. This would increase the variance.

Bagging is also called bootstrap aggregation and involves creating many overfitted models, trained on subsets of the training data. These subsets of data are created from random selections with replacement from the complete training dataset. A final prediction is made by taking an average or choosing the majority of the predictions made, depending on if it is a regression or classification problem. Bagging contributes to reducing variance and redundant information in the dataset, without increasing the bias. Predictions of a single decision tree may be sensitive to redundant information in a dataset, while the average of many trees lowers this sensitivity, as long as the trees are not correlated [14].

By combining these techniques the Random Forest algorithm is known to avoid the common problem of overfitting. It is also a popular algorithm used by data scientist due to the flexibility between regression and classification problems, and the ability to easily determine feature importance. Some disadvantages on the other hand are, that Random Forest models can be time-consuming and require a fair amount of resources when dealing with large datasets. The model is also somewhat complex to interpret compared to single decision trees [15].

Random Forest Classifier

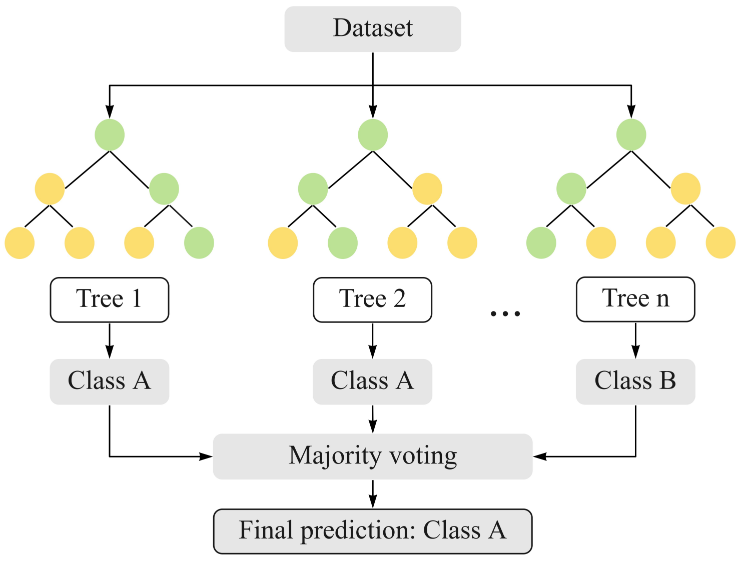


Figure 2.3: Simple illustration of how a Random Forest classifier is built up of multiple decision trees that classify the data as either class A or class B. The final prediction is given by a majority voting. The figure is inspired by Tran [16].

Since gene prediction is a classification problem, the Random Forest classifier is the model implemented. Figure 2.3 summarises the information given above and illustrates how the classifier consists of multiple decision tree classifiers, where each tree makes its own individual class prediction. The class which is represented the highest number of times is the models final prediction. A medium complex classification task can for example require hundreds of decision trees to get a good evaluation-metric score. The default number of trees used by scikit-learn is 100 decision trees. One of the greatest strengths of the Random Forest classifier is its ability to learn non-linear relationships and how separate trees can pick up up local nuances in different subsets of the data. However, the Random Forest classifier is quite complex in its computations, as it computes hundreds of different decision trees. This also makes it expensive in terms of computing resources and training time.

2.2.3 XGBoost

XGBoost is short for eXtreme Gradient Boosted trees, and is also an ensemble method. As the name implies XGBoost is a variant of gradient boosted decision trees, designed for increased speed and improved performance. In order to give a basic understanding of how the XGBoost algorithm works, the concepts of boosting and gradient boosting will be explained. Since the methods used in the XGBoost algorithm are quite complex they will not be explained in great detail in this thesis, due to time limitations.

Boosting is a method that involves combing many *weak learners* iteratively, after which they are combined to get a stronger learner. A weak learner is defined as a model that performs only slightly better than random guessing. Boosting is done in order to reduce bias during training. Gradient boosting takes an ensemble of decision trees and uses a boosting method to improve the samples that led to mis-classification of the previous trees, in order to improve its weaknesses. This is repeated iteratively until one gets a robust and well performing model. Gradient boosting can be explained in terms of its three main components: the *loss function*, a weak learner, and being an additive model. The role of the loss function is to give an estimate of how good the model is at predicting data, given the predicted value in comparison to the true value. The output given by the loss function is often referred to as the loss value of a model. The type of loss function used varies depending on the type of problem one has. Weak learners are defined above and are typically very simple decision trees with few splits and nodes. These decision trees are quite different from the trees used in the Random Forest classifier, which are much more complex with several splits and nodes. Finally, the additive approach refers to how many weak learners are trained and added together one after the other. The goal of each iteration is to have lowered the loss value of the model [17]. New decision trees are therefore added in a sequential manner to improve the error made by the existing model. This is illustrated in Figure 2.4. The reason it is called ”gradient” boosting is because one uses a *Gradient Descent (GD)* algorithm to calculate the gradients such that one can minimise loss when adding new models. The GD algorithm will be explained in more detailed in Section 2.3.

The term *regularisation* refers to different techniques used to combat the problem of overfitting when training a machine learning model (overfitting will be discussed more in the following section). During the training of a model using gradient boosting, one can use various regularisation techniques to control the training and parameter updates of the model. This can for example be as simple as adjusting the number of gradient boosting iterations or adjusting the depth of the decision trees. XGBoost differs from the gradient boosting method in what type of regularisation techniques are used during training. By using the regularisation methods ”L1” and ”L2”, XGBoost has much better generalisation capabilities than more traditional gradient boosting methods. A models ability to generalise refers to its ability to adapt to unseen data.

Other advantages with XGBoost is that it can handle missing values automatically. It is built to do parallel processing, meaning, it can run the trees in parallel on various cores on a CPU making it much more efficient. It can do cross validation at each iteration, which further enables early stopping. Early stopping ends the training of a model early if there is no significant improvement after a given number iterations, which avoids time and resource wastage. XGBoost

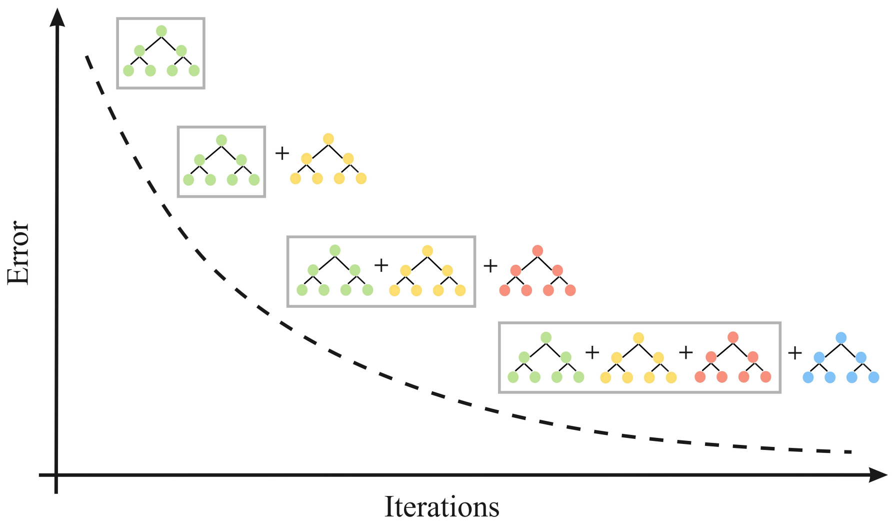


Figure 2.4: Illustrative figure of the logic behind the gradient boosting method. Decision trees are added in an iterative manner, in order to reduce the error of the model.

is also capable of incremental training. This means one can continue training an already trained model, as one gets new data over time, compared to starting training from scratch. Finally, it also has a feature called Tree pruning, which permits the model to simplify trees so that they become deeper and more optimised.

2.2.4 Training and Validation

When implementing a machine learning model, it is common practice to split the dataset into a training dataset and a testing dataset, given that the dataset is large enough. The idea with this is to only train the model on the training data and hold out the testing data to see how well our trained model performs when it makes predictions or classifications on unseen data. One of the dangers when training a machine learning model is overfitting. This can happen if too much attention is paid to random or local variations in the dataset, that do not have much intrinsic value. This further results in a model that does not generalise well. This means it performs well on the training data, giving a high accuracy score, but on unseen data it performs poorly. Splitting the data into training and testing data addresses this issue, as one makes sure that the trained model is tested on unseen data and can confirm that the model is not heavily overfitted. However, overfitting may still go by unnoticed. If our dataset is small, the training and test datasets can by chance turn out to be quite similar. Meaning, the same random variations that occur in the training dataset may occur in the test dataset as well. One way to further protect against overfitting is *cross validation*.

K-fold Cross Validation

K-fold cross calidation involves splitting our training data into K randomly assigned subsets, then repeatedly fitting our model a given number of times, while always training on K-1 folds while validating on the Kth fold. This is repeated until all the K folds have been used once as a validation subset. Figure 2.5 illustrates how the data is split in a 5-fold cross validation, and how a different subset is used to validate during each repetition. K-fold cross validation is integrated as a parameter into scikit-learn classes and models, such as the *GridSearchCV* class for hyperparameter tuning, and the XGBoost model.

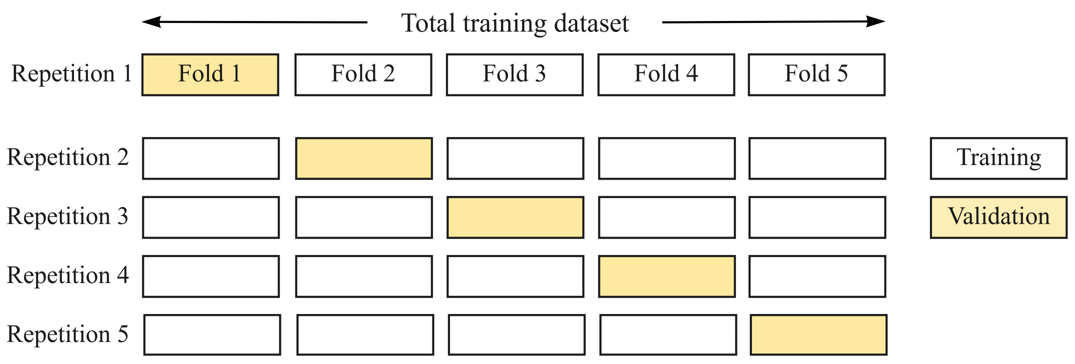


Figure 2.5: Illustration of the splitting of training data in a 5-fold cross validation. The figure is inspired by Koehrsen [18].

2.2.5 Hyperparameter Tuning

When implementing machine learning models one can think of the hyperparameters as design choices affecting the final architecture of the model. A machine learning model has both trainable parameters and hyperparameters. The parameters of the model are settings that can be learned and adjusted during training, for instance the variables and thresholds used to split each node in a decision tree. The hyperparameters on the other hand are settings that must be set before training, such as the number of decision trees in the forest and the number of features considered when splitting a node [18].

2.2.6 Feature Selection

The purpose of feature selection is among other things to reduce training time and improve the performance of a machine learning model. To perform feature selection, the data must have features that are redundant or irrelevant to the model, so that their removal does not lead to information loss [19]. Similar to machine learning algorithms, feature selection methods can be divided into supervised and unsupervised methods. The supervised methods involve the target variables while the unsupervised methods do not. Instead, unsupervised methods use other statistics such as feature correlation, and transformations such as principal component analysis to select features. Supervised methods can further be divided into filter, wrapper and embedded methods. Random forest has its own built in feature selection method that is tested in this thesis, in addition to manual feature selection based on the feature importances computed during training. This will be discussed further in Chapter 4.

2.3 Deep Learning

ANNs form the basis of deep learning and are inspired by the structure of the human brain. Deep learning methods are distinguished from machine learning methods by their use of layered structures of algorithms. These layers enable deep learning models to have even more sophisticated learning abilities, and they can process unstructured data as well as automate the feature engineering process. Deep learning is even more independent from human intervention than other machine learning methods, and can directly take raw data as an input. Training can be supervised, unsupervised or semi-supervised [10].

The deep learning model implemented is a RNN, which is trained in a supervised manner. RNNs are a subtype of ANNs that contain *recurrent layers* in addition to the *dense layers*. These terms will be explained in the following subsections.

RNNs are state-of-the-art algorithms for sequential data. Sequential data is data where the data points have an order to them and are often dependent on one another, such as time series data, financial data or DNA sequences. RNNs can produce predictive results in these type of data, that are more accurate and reliable than other deep learning algorithms, due to their ability to "memorise" the recent past inputs. This feature makes the RNN model unique as it is one of very few models that has the capacity for internal memory. To understand the structure of a RNN model one first needs to understand basic ANN structures such as the *perceptron* and *multi-layer perceptrons (MLP).*

2.3.1 Perceptron

A perceptron can be thought of as a single artificial neuron, and is often referred to as a node in a neural network. Perceptrons are the core processing units of the neural networks. Equation 2.4 is the algebraic representation of how a perceptron produces an output ($O(x)$) [9], and Figure 2.6 illustrates the workflow of a single perceptron. It is given data at one end, denoted by x_1, x_2, to x_n in the figure. Then the data inputs are multiplied with their respective weights w_1, w_2 to w_n, and added together. An additional bias b, is added to the weighted sum. The next step is an *activation function*, denoted by f, which will be explained in more detail in Section 2.3.3. The weights that are multiplied with each data input are used to adjust the impact of each data point on the model as a whole. The bias is used to adjust the initial level sent into the activation function by adding a constant to the weighted sum of the input data. The activation function determines what the output of a single perceptron is, depending on the input. The purpose is to transform the weighted sum of the inputs and bias, to an output that can be used to give a prediction. The activation function is a crucial part of a neural network's ability to learn non-linear patterns from input data. On their own perceptrons are able to classify linearly separable inputs, and perform linear regression.

$$O(x) = f\left(\sum_{i=1}^{m} w_i x_i + b\right) = f\left(w_1 x_1 + w_2 x_2 + \ldots + w_m x_m + b\right) \tag{2.4}$$

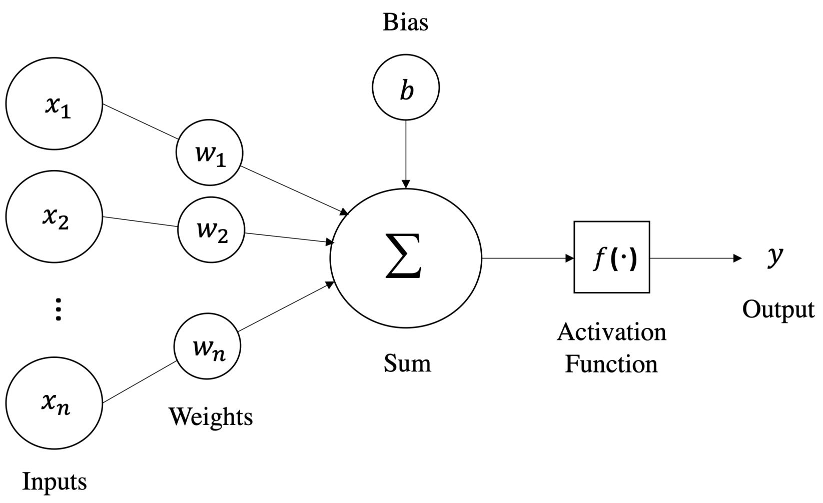


Figure 2.6: Sketch of a perceptron. It receives an input (x), makes a weighted (w) summation of the inputs, adds a bias (b). Finally, the activation function (f) computes an output (y).

2.3.2 Multi-layer Perceptrons

Combining multiple perceptrons gives us a layer of perceptrons. Then combining multiple of these layers gives us multi-layer perceptrons (MLP). Adding a non-linear activation function to the perceptron makes it able to capture some non-linear relationships in the input data. Combining multiple layers of perceptrons together make them able to capture much more complex relations. Figure 2.7 illustrates the architecture of a simple MLP network.

The input layer of the MLP receives the input and the output layer predicts the final output. In between these two layers are the hidden layers which perform complex computations required by the network, before passing their output on to the next layer. This "feed forward flow" of information from one end to the other is called *forward propagation*. Neurons of one layer are fully connected to neurons of the next layer, which are assigned weights. This type of layer is called a dense layer, and is the most basic layer used in ANNs.

The number of layers used in an MLP and the number of neurons in each layer is part of the architecture of the model. Elements such as the loss function, the *optimiser* and the *learning rate* used during training, are hyperparameters that together with the architecture of the neural network play an important role in determining how high performance the model can attain. These concepts will be explained in the following paragraphs.

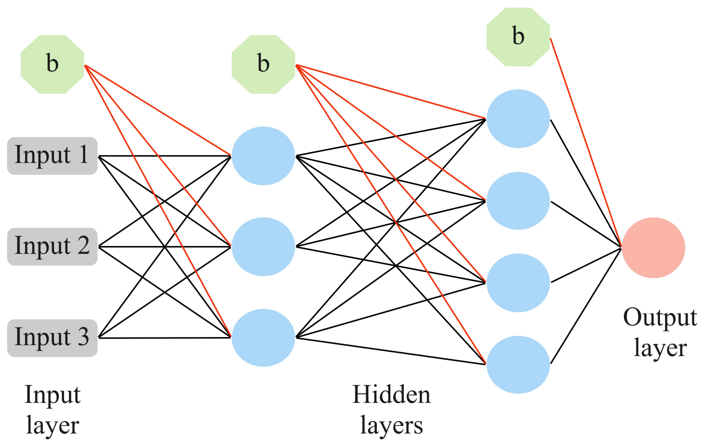


Figure 2.7: Example of a simple MLP consisting of an input layer with three input units plus a bias unit. Two hidden layers with three and four hidden units plus each their bias unit respectively. Finally, there is a single output layer.

2.3.3 Activation Functions

As mentioned, the activation function determines the output of every single node in a layer, depending on its input. Every layer after the input layer in an MLP uses a specific activation function for each of the nodes in that layer. There are several different activation functions, and which activation function is used in a layer depends on the purpose of that particular layer. The choice of activation function has a large impact on the performance and training of the neural network. The hidden layers in an MLP often use the same activation function, while the output layer uses a different activation function depending on the type of prediction that is to be made by the model. The typical activation functions used for hidden layers are *Rectified Linear Unit (ReLU)*, *Logistic (Sigmoid)*, *Hyperbolic Tangent (Tanh)* and *Linear* [20]. These functions are shown in Equation 2.5. There are several more activation functions than the ones mentioned here, including different variants of the ReLU function. Which activation function is used also depends on what type of neural network is being built.

$$\begin{aligned}
Tanh(x) &= \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}} \\
Sigmoid(x) &= \frac{1}{1 + e^{-x}} \\
ReLu(x) &= max(0, x) \\
Linear(x) &= x
\end{aligned} \tag{2.5}$$

Figure 2.8 shows a simple mind map of the most common activation functions for hidden layers, depending on the network type. In this thesis a recurrent neural networks was tested, and therefore the Sigmoid and Tanh activation functions are the most relevant. Figure 2.9 presents the most common activation functions used by the output layer in a neural network, and how they vary depending on the problem type.

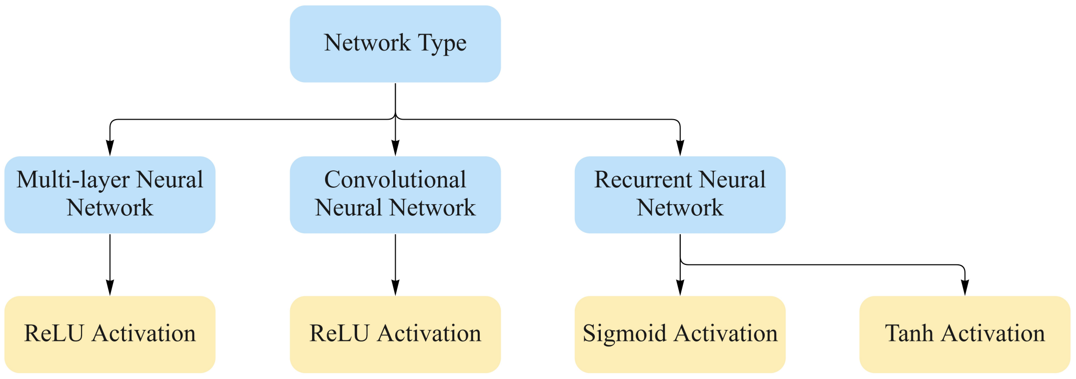


Figure 2.8: Overview of activation functions that are common to use for hidden layers depending on the network type. The figure is inspired by Brownlee [20].

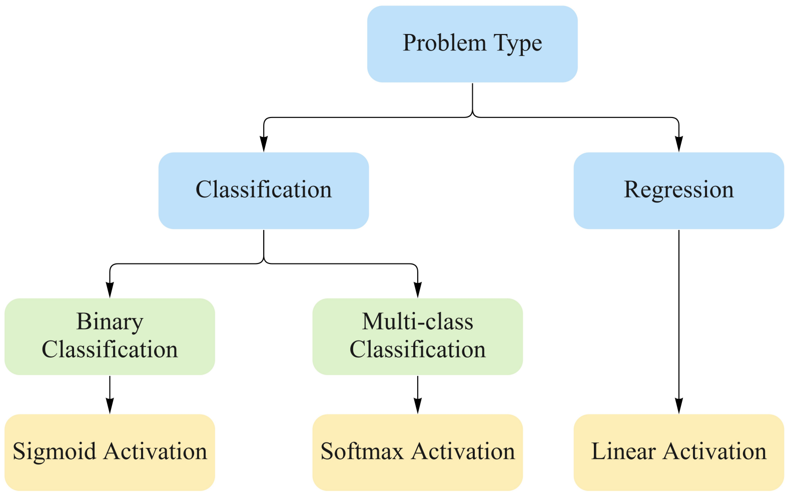


Figure 2.9: Overview of activation functions that are common to use for output layers depending on the problem type. The figure is inspired by Brownlee [20].

2.3.4 Training

During the forward propagation in an MLP, a subset of the training data, called a *batch*, is given as input to the model. This is passed through the different layers in the network, before an output is predicted in the output layer. At this stage the training and learning process of the neural network begins. The predicted output is compared with the actual output, so that one can calculate the error of the prediction. The magnitude and signs of the error indicates how far off from the truth the model is, and in which direction. One then computes the *loss function* which computes the error as a function of the network's trainable parameters. Similar to the activation function, there are different loss functions that can be used depending on what type of model and problem one has. *Binary cross-entropy* is the most common loss function for binary classification problems. Once one has a loss function, this information is used to update the trainable parameters of the network, such as weights and biases. The process of updating these trainable parameters can be thought of as a gradient optimisation problem where the parameters are updated iteratively using a given *optimisation function (optimiser)*. There are several optimisation functions that can be used to train a neural network. Three of the most widely used functions are *Stochastic Gradient Descent (SGD)*, *Root Mean Square Propagation (RMSProp)* and *Adaptive Moment Estimation (ADAM)* optimiser. The latter two are tested in this thesis. These optimisation functions have in common that they use the GD algorithm with *Back-propagation*, and differ in the way their *learning rates* are optimised. These concepts will be explained in the following paragraphs.

Gradient Descent with Back-propagation

The goal of GD is to find the parameter values of the model that minimises the loss function as far as possible. In order for the GD algorithm to update the parameters during each iteration of training, the gradients of the loss function with regard to each individual parameter in the MLP needs to be estimated. This is done by the back-propagation algorithm, which can be thought of as a computationally efficient approach to computing these gradients. It relies on the chain rule and automatic differentiation to calculate the derivatives backward through the layers of the neural network [21]. Automatic differentiation are special techniques developed to solve complex gradient problems efficiently [22]. These gradients are then used by the GD algorithm to give an indication of which direction one should adjust the parameters in order to minimise the loss value.

Optimisers and Learning Rate

In addition to the gradient of the loss function, the GD algorithm needs to know the step size to take towards the local minimum when updating the trainable parameters. This is also called the learning rate of the algorithm. The learning rate determines the amount at which the parameters are adjusted at each iteration during training. If the learning rate is too large one risks overshooting the optimal minimum, and if the learning rate is too small the training of the model may take very long, or get stuck in a local minimum. The gradients calculated during back-propagation affect how the learning rate is adjusted during training, depending on what optimiser is used. Different optimisers have different ways of tackling these problems.

The GD algorithm can be improved by adding what is called "momentum" to the learning process. The momentum is calculated by taking an exponential weighted average of past gradients, which is then used to update the trainable parameters. By adding the momentum term the parameter adjustments are smaller in directions where the past gradients oscillate, and higher in directions where the past gradients have the same direction for a longer period. In other words, one makes use of the momentum accumulated from past gradient calculations when defining the current gradient. This allows the GD algorithm to avoid getting stuck in a local minimum, even when the current gradient is close to zero [23]. By taking smaller steps when the gradient has oscillated a lot in past iterations, one may avoid overshooting the optimal minimum. Similar to GD with momentum, the RMSProp optimiser also uses past gradients. It normalises the current gradient by taking a moving average of past squared gradients, and uses this to adjust the learning rate. This decreases the learning rate for large gradients and increases the learning rate for small gradients, in attempt to avoid the problems of exploding and vanishing gradients. The ADAM optimiser is a combination of GD with momentum and RMSProp. It takes the exponential weighted average of the past gradients such as GD with momentum, and adjusts the learning rate using the moving average of the squares of recent gradients, such as RMSProp.

Different neural network architectures can be more prone to vanishing and exploding gradients. If one has an architecture with many layers for instance, vanishing gradients can occur due to the error gradients becoming smaller and smaller as one moves backward through the layers during back-propagation. Exploding gradients on the other hand occur when the error gradients become larger and larger during back-propagation, causing unreasonable weight and parameter updates. RNNs, which are often used for modelling time-dependent and sequential data, are prone to vanishing and exploding gradients. This can inhibit them from learning from long data sequences. The long term memory introduced by *Long Short-Term Memory (LSTM)* units, can prevent the problem of vanishing and exploding gradients in RNNs. This will be discussed in the Section 2.3.5.

Batches and Epochs

During training, the MLP is repeatedly given batches of training data. This is repeated until the MLP has been trained on all the training data once. The model has then been trained for one *epoch*. MLP's are normally trained for multiple epochs, and the repetitive cycles of forward and backward propagations is how the model learns. The chosen batch size determines how often the weights are updated during one epoch, and the number of epochs determines how many iterations the model will be trained for on the entire training dataset. If the number of observations in the training data is not divisible by the batch size the last observations that do not make up a complete batch will be discarded during that epoch.

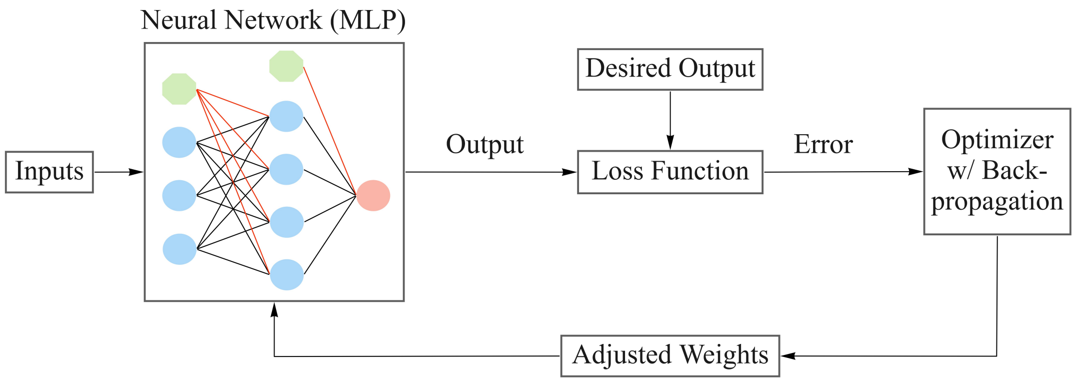


Figure 2.10: Illustration of the training process for a multi-layer perceptron.

In addition to setting the batch size, and the number of epochs during training, one also sets a validation split size. This determines the percentage of the training data that will be held aside for validating the model performance after one epoch. The remaining data is split into batches of the given batch size. It is said that the number of epochs used to train a model should be increased until the validation score starts to decrease even though the training accuracy increases [24]. Figure 2.10 illustrates the different stages of training that have been mentioned in this section.

2.3.5 Recurrent Layers

Dense layers such as the one presented in the simple MLP in Figure 2.7, consider only the current input received at each node. They have no memory of previous inputs which will make it difficult for such a model to predict what is coming next in a sequence for instance. A recurrent layer on the other hand, consists of special perceptrons that have the ability to loop back the output from a node such that it considers both current inputs as well as past inputs simultaneously. A comparison between a simple MLP and RNN is presented in Figure 2.11.

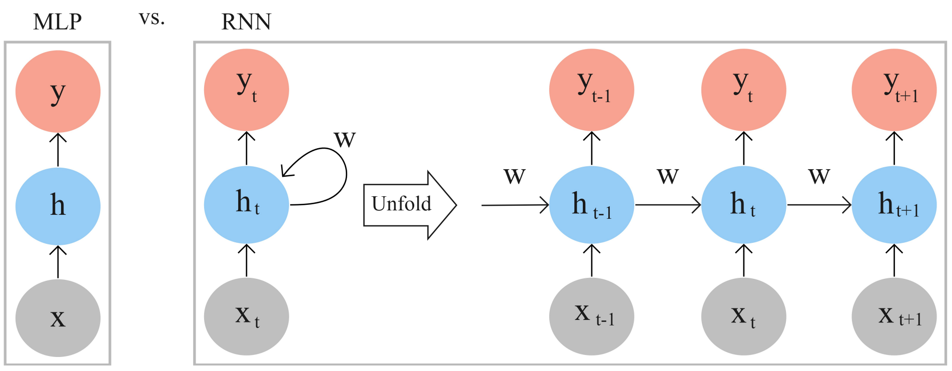


Figure 2.11: Comparison of a simple MLP and RNN. The networks consist of an input layer x, a single hidden layer h and an output layer y. The t in the RNN denotes the time step of each input, and the unfolded version of the RNN illustrates how input is received from both the input at the current time step as well as from the hidden layer of the previous time step.

Both the networks in the figure have only one hidden layer, denoted as h and h_t respectively. The nodes in each of the layers are not displayed in the figure, but one can assume that both the input layers (x), output layers (y) and hidden layers have many nodes. The t in the RNN denotes the time step from one input to the next. As mentioned above RNNs have the ability to receive input from both the input layer at the current time step, as well as from the hidden layer of the past time step [22]. This is more clearly presented in the unfolded version of the RNN in Figure 2.11. The figure should not be confused with a RNN containing many layers, but rather an illustration of the time steps of a RNN. The looping of information also includes a weight (W) that is applied to the output that is to be memorised. This determines the extent to which previous output affects the current input.

The fact that RNNs are able to memorise previous inputs make them particularly efficient for the purpose of sequence modelling. RNNs are commonly used in sentiment analysis, which for example involves taking a text as input and predicting a class label as output (many-to-one). It can be used for image captioning, which involves getting a single image as input and a phrase describing the image as output (one-to-many). Or, it can be used for translation, taking an English phrase as input for example and giving the translated phrase as an output (many-to-many) [22].

A recurrent layer's ability of short-term memory is both its strength and its weakness. When presented with long sequences the RNNs may struggle to remember enough of the past information to make correct predictions. In addition, these long sequences may cause vanishing or exploding gradient problems. LSTM layers are a popular solution to this problem.

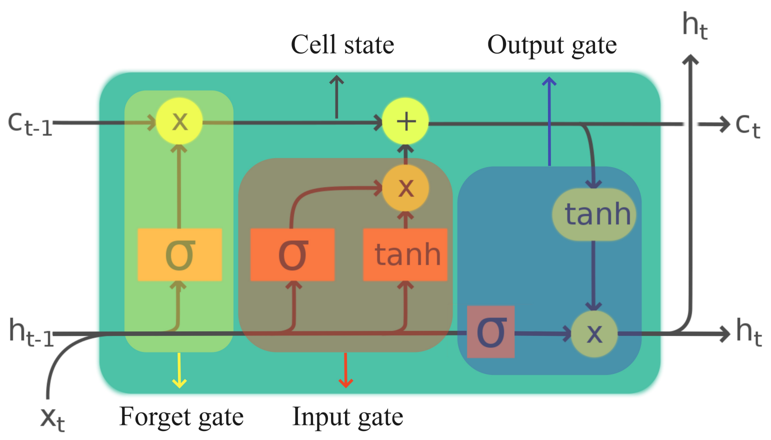


Figure 2.12: Illustration of an LSTM layer and how it is controlled by a memory cell, a forget gate, a input gate, a output gate and the respective activation functions. The input from the current time step to the cell is denoted by x_t. c_{t-1} is the cell state from past time steps, h_{t-1} is the input from past time steps, c_t is the cell state output, and h_t is the layers output. The figure is inspired by Graves, Mohamed, and Hinton [25].

LSTM layers are advanced recurrent layers, and were developed to handle chronological sequences and long-term dependencies in a better manner than the vanilla-RNN. An LSTM layer can store information for a longer period of time, as well as control the extent at which past inputs are memorised or forgotten. When implementing a simple LSTM architecture, the LSTM layer replaces the hidden layer of the RNN. Imagine the h_t of the RNN in Figure 2.11 being replaced by what one calls an *LSTM cell*. Figure 2.12 is an illustration of an LSTM cell. In addition to h_t, which in this case is the output of a hidden layer from a given time step, an additional connection between every layer is also added in LSTM layers, called the cell state c_t. Each LSTM layer is composed of a memory cell and three gates, as shown in Figure 2.12. The cell memorises information over arbitrary time intervals, and the three gates control the flow of information through the cell [26]. The input gate controls whether the memory cell is updated, the forget gate determines if information is to be discarded or if it is should impact the output, and the output gate controls whether the information of the current cell state is made visible. The combinations of the Tanh and Sigmoid activation functions used in the LSTM cell help distribute the gradients, which in turn helps solve the problem of vanishing or exploding gradients during back-propagation.

2.4 Evaluation Metrics

The metrics *accuracy*, *precision* and *recall* are used to evaluate the performance of the three machine learning models. They are all based on the true positive (TP), false positive (FP), true negative (TN) and false negative (FN) values derived from the model predictions. True positives are CDS that have been correctly classified as CDS by the model, and false positives are n-LORFs that have been incorrectly classified as CDS. True negative are correctly classified n-LORFs, and false negatives are CDS that have been incorrectly classified as n-LORFs. Accuracy is a measure of the share of correct predictions considering the total number of classifications. Precision is a measure of the number of correct positive predictions given the total number of positive predictions made. Recall on the other hand measures the amount of correct positive predictions given the total number of positive instances in the data. The equations for each of the three metrics are given in Equation 2.6.

$$\begin{aligned} Accuracy &= \frac{TP+TN}{TP+TN+FN+FP} \\ Recall &= \frac{TP}{TP+FN} \\ Precision &= \frac{TP}{TP+FP} \end{aligned} \tag{2.6}$$

2.5 Related Work

The articles written by Silva et al. [2] and Al-Ajlan and El Allali [3] have inspired much of the work done in this thesis. In this section details regarding the methods used in both papers will be explained, how the approaches differ and which method proved to be more effective. Both articles aim to identify protein coding genes in prokaryotic genomes, using different machine learning techniques and datasets. When using multiple genomic samples for gene prediction, the mixing of genetic information occurs and the read fragments are often short and incomplete. This can impact the quality of state-of-the-art gene prediction tools.

2.5.1 GeneRFinder: Random Forest Classifier

Silva et al. [2] present an ab-initio gene prediction tool, called GeneRFinder, which aims to tackle the difficulties regarding gene prediction in genomic complexities and identify both CDS and intergenic regions in such DNA sequences. The tool uses the Random Forest classifier, and this method was chosen considering it was the best performing machine learning model when compared to other classification methods. GeneRFinder is a tool based on ORFs extracted from genome samples. It makes predictions based on signals captured from these ORFs [2]. 11 sets of features are used in the final version of the GeneRFinder. Four of which correspond to the GC-content of the different reading frames in the ORFs as well as the overall GC-content. Five features correspond to k-mer frequencies from 2-mers to 6-mers. One feature aims to take the codon bias of synonymous codons into account. This is implemented by counting the codons in the reading frame of each ORF. The last feature is the length of each ORF.

GeneRFinder was trained and tested on complete genomes, with their respective gene annotations from the NCBI database. After the ORFs were extracted from the genome data, they were labelled as either a positive instance or negative instance depending on whether or not the ORF was a coding gene. The tool was trained on 129 complete genomes, of which 11 were archaea and 118 were bacteria. Around 700,000 sequences were extracted from the genomes in total, of which approximately half were positive instances and the remaining half were negative instances. The tool was then tested on a new set of 12 public genomes, of which three were archaea and nine bacteria. From these genomes 50,000 ORFs were extracted in total, of which approximately 30,000 were positive and 20,000 were negative instances. GeneRFinder outperformed the state-of-the-art gene prediction tools Prodigal [27] and FragGeneScan [28], and consequently received higher evaluation-metric scores.

The method used to derive the GeneRFinder tool has largely inspired the Random Forest classifier implemented this thesis. When building the Random Forest classifier presented in Chapter 4, a similar approach was followed as for the GeneRFinder tool, but using different feature engineering and selection techniques. The incentive of this was to see how the alternative Random Forest model would compare to the results of the GeneRFinder tool, as well as to the XGBoost and Recurrent Neural Network models. These are also presented in Chapter 4.

2.5.2 CNN-MGP: Convolutional Neural Network Classifier

As opposed to the more common approaches that rely on feature extraction and ensemble methods, Al-Ajlan and El Allali [3] present the tool CNN-MGP, which uses a *Convolutional Neural Network (CNN)* for genomic gene prediction. This approach automates the process of gene prediction further by avoiding the need for feature engineering and feature selection before feeding the ORFs to the machine learning model. The CNN-MGP simply takes raw DNA sequences as input, and extracts the necessary information needed to classify an ORF as either a coding or non-coding gene. An interesting aspect to the method used by Al-Ajlan and El Allali [3] is that they chose to train 10 different CNN models on 10 different datasets, that are arranged according to their GC-content. GC-content is known to be quite varying across genomes. The CNN-MGP addresses this variation in a different manner than other gene prediction tools. Different versions of the CNN-MGP are used on various genome samples depending on the samples' GC-content.

Before the raw DNA sequences are inputted to the appropriate CNN-MGP, the ORFs are extracted and numerically encoded by character-level one-hot encoding. 131 prokaryotic genomes from the NCBI GeneBank, and their respective annotations, were used to create the training dataset. The length of the genome fragments were limited to 700 base pairs, of which seven million ORFs were extracted. These ORFs were then divided into 10 mutually exclusive datasets based on their GC-content, which were then used to train the 10 different CNN-MGP models. The test dataset consisted of three archaeal and eight bacterial genomes, that were also limited in length to to 700 base pairs. The fact that the sequences are limited to this length means one excludes all CDS that may be longer than this, which is a large limitation. The architecture used in the CNN-MGP consists of four alternating convolutional and max-pooling layers, followed by a flattening layer before a fully connected layer, and finally the output layer which produces the gene probability. Since convolutional layers are not used in this thesis, an explanation of what they are falls outside the scope of this thesis.

The CNN-MGP model proves that deep learning methods can be used for gene prediction in prokaryotic DNA sequences, and its performance is better than or comparable to state-of-the art gene prediction tools such as Orphelia [29] and Prodigal. Considering the CNN architecture used by Al-Ajlan and El Allali [3] was fairly simple, but yet successful, it makes one wonder how other deep learning models would perform, such as Recurrent Neural Networks.

Chapter 3

Data Exploration

In this chapter the sequenced genome data used to develop the gene prediction models are presented and explored in detail. This includes methods and reasoning used during data selection and pre-processing, as well as assumptions and limitations made in order to make the data amounts manageable and limit the scope of the thesis.

The data exploration and pre-processing was done in R Studio [30], relying largely on the R-packages **seqinR** [31], **microseq** [32] and **microclass** [33]. The purpose and versions of each package used can be found in Table A.1 in Appendix A. GitHub-links to the R-scripts created to process the genomes and make the datasets in this chapter can be found in Table B.1 in Appendix B.

3.1 Selecting Genomes

As mentioned in Chapter 2, Section 2.5, this thesis is largely based on the work done by Silva et al. [2] and Al-Ajlan and El Allali [3]. When the *GeneRFinder* and *CNN-MGP* tools were developed, a combination of fully sequenced bacterial and archaeal genomes were used to build their datasets, extracted from the *NCBI GeneBank.* The genomes selected to build the datasets in this thesis are 15 bacterial *reference genomes* extracted from the NCBI *Reference Sequence Database (RefSeq).* In comparison to the NCBI GeneBank database, the RefSeq database is said to be more selective as only the more verified and complete genomes are represented there. A reference genome is a representation of the complete set of genes in an individual organism of a species [34]. Even though these genomes are considered the most sequenced and well-annotated genomes, the reality is that there is no true way of knowing that these annotations are correct or complete. However, since they are the closest thing to the truth that is available, one can assume that they are correct and complete for the purpose of this investigation. The limiting factor when selecting genomes from the Refseq database was therefore, that only genomes that are considered reference genomes were to be selected. After filtering out all genomes that were not reference genomes, only 15 bacterial genomes were remaining. No archaeal genomes were considered reference genomes in the database.

3.2 Description of the Data

In Table 3.1 the 15 bacterial genomes are presented. The table includes the latin name for the species of each genome, the number of CDS, the number of n-LORFs (which neither are, or include CDS), the total number of LORFs in each genome, as well as the percentages of the two classes of LORFs compared to total number of LORFs extracted from each genome.

When exploring the 15 bacterial reference genomes a number of observations were made. The two most obvious differences when looking at Table 3.1 is that the genomes vary greatly in the number of total LORFs extracted from each genome. Secondly, there are far more CDS than n-LORFs in each genome. During data prepossessing other characteristics and differences between CDS and n-LORFs were also taken in account, which will be presented in the following section. In the table one sees how all the genomes are from different species except for the two *E. coli* genomes, that are different strains of the same species. CDS and n-LORFs make up on average 5.54% and 94.47% respectively, of the total number of LORFs extracted from each genome on average. The number of CDS extracted compared to the number of n-LORFs is significantly smaller. However, it is important to keep in mind that these percentages are not equivalent to showing how much of a genome that consist of CDS or n-LORFs. For this one also needs to take the length of the extracted CDS and n-LORFs sequences into account. In the table one can see that most of the genomes deviate only around 1% over or under the average for both CDS and n-LORFS, except for the genomes *S. aureus* and *E. coli str. Sakai.* These deviations are marked in bold in Table 3.1 and they are noticeably higher for CDS and lower for n-LORFs compared to the respective mean values.

Table 3.1: Descriptive table of the 15 genomes used in this thesis. Including their latin name, number of CDS, number of n-LORFs, number of total LORFs extracted from the genomes, and the percentages of CDS and n-LORFs. The percentages are simply the number of CDS or n-LORFs divided by the total number of LORFs found in each genome.

Latin name	CDS	CDS%	n-LORFs	n-LORFs%	Total LORFs
Mycobacterium tuberculosis	4324	4.87%	84479	95.17%	88762
Staphylococcus aureus	5573	**8.09%**	63334	**91.91%**	68906
Salmonella enterica	4313	4.60%	89525	95.40%	93838
Listeria monocytogenes	4549	5.13%	84172	94.88%	88717
Campylobacter jejuni	1833	4.71%	37106	95.29%	38939
Shigella flexneri	888	4.70%	18015	95.31%	18901
Pseudomonas aeruginosa	5156	4.83%	101582	95.17%	106736
Bacillus subtilis	4240	5.03%	80127	94.98%	84363
Coxiella burnetii	1579	6.03%	24629	94.00%	26201
Chlamydia trachomatis	2727	5.25%	49209	94,69%	51967
Escherichia coli str. Sakai	3886	**8.01%**	44658	**92.00%**	48543
Escherichia coli str. K-12	3599	4.68%	73262	95.32%	76861
Caulobacter vibrioides	3906	5.71%	64518	94.29%	68424
Acinetobacter pittii	2867	5.15%	52780	94.85%	55645
Klebsiella pneumoniae	5779	6.28%	86260	93.72%	92039
MEAN	3681	5.54%	63577	94.47%	67256

3.3 Data Preprocessing

When pre-processing the datasets, each genome was explored separately due to their different composition with regards to CDS and n-LORFs. For the sake of simplicity, CDS will be referred to as positive instances and the n-LORFs as negative instances. The pre-processing started with extracting the positive instances from the genome, and then the negative. When extracting the negative instances, a limitation was made regarding the length of the n-LORFs. Only n-LORFs that were longer than 45 base pairs were extracted. From Table 3.1 one can recall that the percentage of n-LORFs extracted from the genomes was significantly larger than CDS. In addition, a large number of the n-LORFs in each genome are very short. Hence, the filtering was done in order to limit the number of short n-LORFs and also the total number of n-LORFs extracted. This reason for choosing a length of exactly 45 base pairs was somewhat arbitrary, but also inspired by the article by Silva et al. [2], where the minimum length was set to 90 base pairs.

Table 3.1 shows that the number of positive and negative instances in each genome is very unbalanced. After selecting both the positive and negative instances one would ideally just remove the number of negative instances that are excessive, so that one is left with an equal number of positive and negative instances in the final dataset. However, after studying the length-distribution of the instances in both classes, it was apparent that there was a large difference in this distribution which needed to be taken into account.

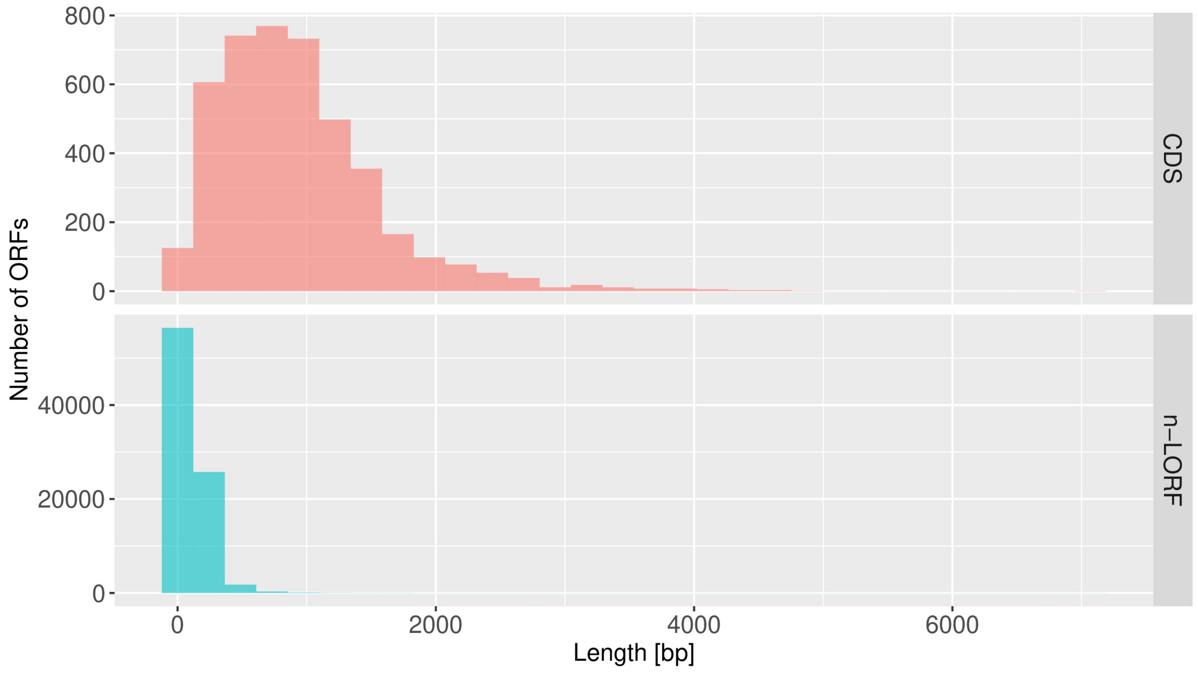


Figure 3.1: Histogram plots featuring the length and number distribution of all CDS and n-LORFs extracted from the genome of species *M. tuberculosis*. The length is measured in number of base pairs [bp]. Notice that the y-axis is not the same in both plots.

In Figure 3.1 one can see a histogram plot of the length-distribution of the positive and negative instances in the genome of species in *M. tuberculosis*. Notice that the x-axis is the same in both plots but the y-axis differs. The length of an ORFs is measured by the number of base pairs it consists of. From this figure one can first and foremost see that there is a large difference in the length distribution between the two classes. The positive instances are fewer in number but have a much larger number of long instances than the negative class. This is easy to see in Table 3.2. The majority of the instances in the negative class in this genome are quite short compared to the positive class. This is the general trend in all of the genomes. In order to make balanced datasets for each genome with regards to both the number of positive and negative instances as well as the individual length-distributions of each class, the instances in both classes were categorised based on their lengths.

In Table 3.2 an example of how the positive instances were categorised based on their base

Table 3.2: CDS arranged according to the given length categories.

Category	Count
0-100	6
100-200	93
200-300	387
300-500	839
500-800	938
800-1100	879
1100-1400	548
≥1400	623

Table 3.3: n-LORFs arranged according to the given length categories.

Category	Count
0-100	50142
100-200	27660
200-300	7358
300-500	3184
500-800	820
800-1100	205
1100-1400	84
≥1400	72

pair lengths is presented. Table 3.3 presents a similar table with the categorisation of the negative instances. Based on the counts of the positive instances in each length category, the same number of instances were sampled from the same length category of the negative instances, in order to create a balanced dataset. However, there was a limiting factor when doing so. The negative instances in most of the genomes had fewer long instances than the positive instances, as one can see when comparing Table 3.2 and 3.3. This made it impossible to get an identical length distribution while simultaneously keeping the number of positive and negative instances equal. At least without removing CDS.

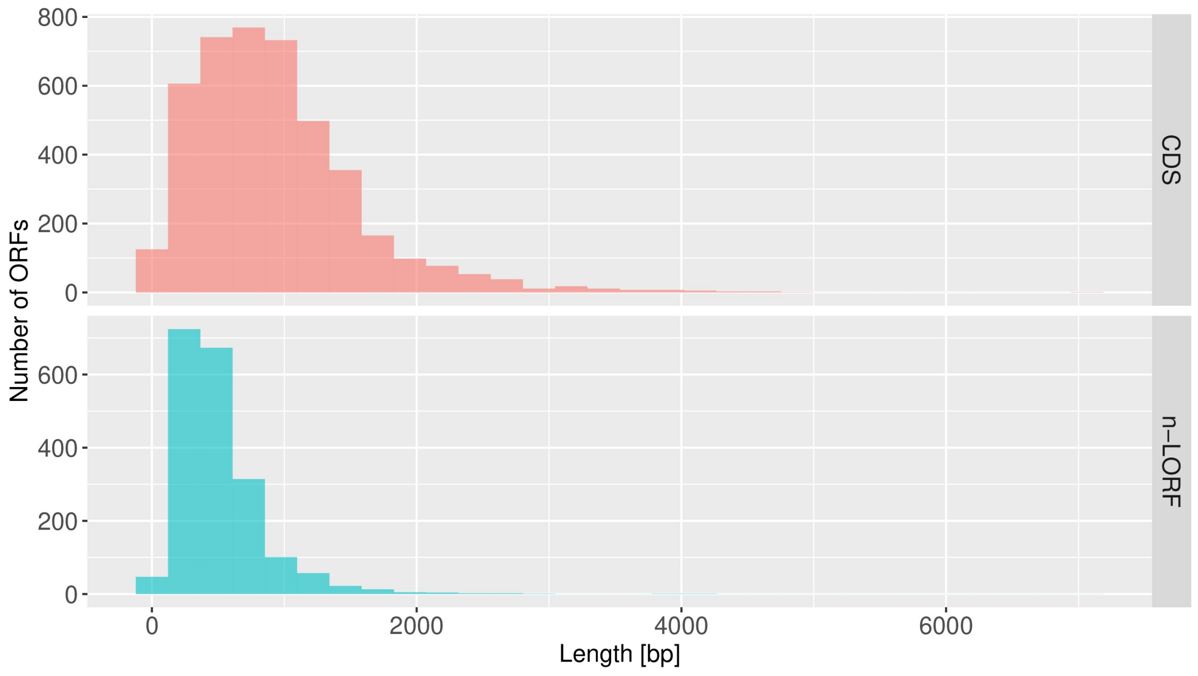


Figure 3.2: Histogram plot featuring the length and number distribution of CDS and n-LORFs in dataset 1, from the genome of species *M. tuberculosis*. The length is measured in number of base pairs [bp].

The challenge with identifying CDS among the LORFs in a genome is quite different for long and short LORFs. If one makes a random guess that a LORF that is longer than 1400 bases is a CDS one can be almost 90% certain that this is correct. Among the short LORFs in a genome, on the other hand, the ratio of CDS is about 1:10 000. For that reason two datasets for each genome was created. The first dataset (dataset 1) attempts to have an as balanced as possible length-distribution between the positive and negative instances, which therefore also means the numbers of positive and negative instances will be unequal in most genomes. While the second dataset (dataset 2) is primarily the same as the first, but with additional negative instances so that the number of positive and negative instances becomes equal. In Figure 3.2 and 3.3 one can see how the length and number-distribution between the negative and positive instances varies depending on the datasets described above. The aim with creating these two datasets for each genome is to see if the model gets better at classifying CDS when trained on a dataset that is more balanced in terms of the length of the sequences.

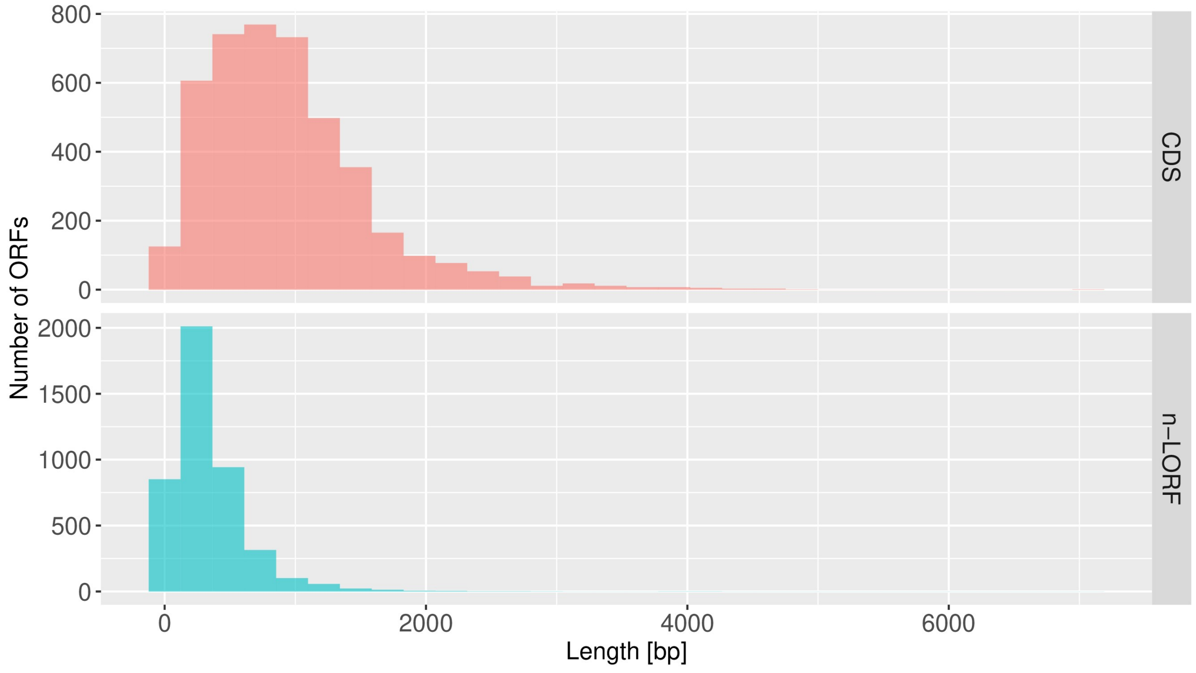


Figure 3.3: Histogram plot featuring the length and number distribution of CDS and n-LORFs in dataset 2, from the genome of species *M. tuberculosis*. The length is measured in number of base pairs [bp].

Chapter 4

Methods

The implementation of the three machine learning methods will be presented in this chapter. The models have been trained tuned, and tested using two different training datasets, and using single genomes as well as combined genomes. The methods and reasoning involved in the selection and engineering of features will also be discussed. Feature engineering was done in R Studio, while the implementation of all three models was done in Kaggle using Python programming language. The Python libraries used can be found in Table A.2 in Appendix A, and GitHub-links to the source code can be found in Table B.1 in Appendix B.

4.1 Feature Engineering

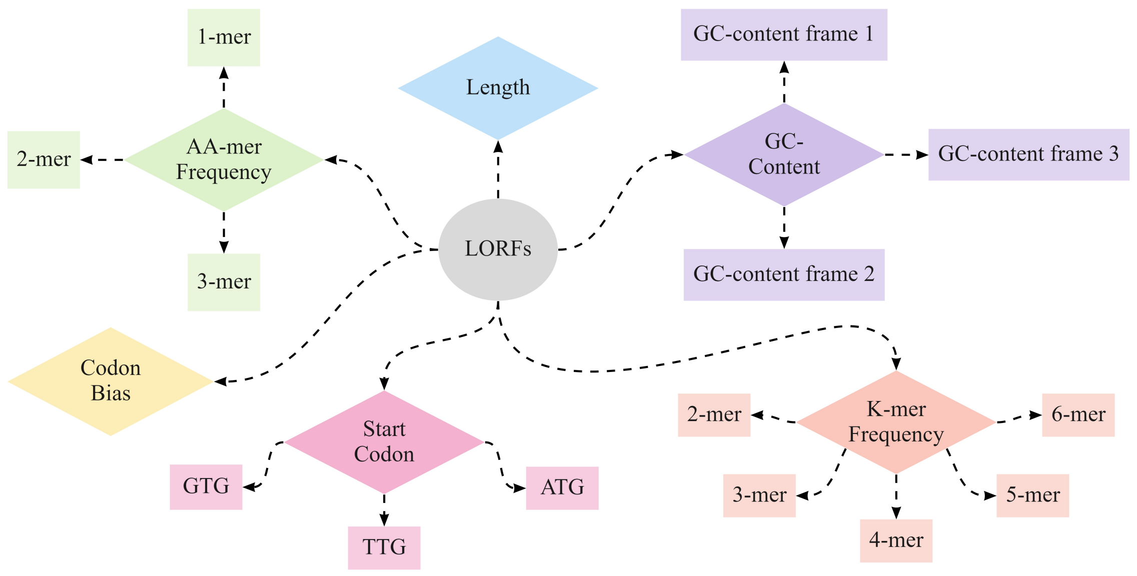


Figure 4.1: Features engineered from LORFs in 15 different genome datasets, which are to be used as input in the machine learning models Random Forest and XGBoost.

Feature engineering is a preprocessing step where one extracts the most relevant information from the raw dataset, which then can be used as input in a machine learning model. This process often involves selection, creation and transformation. By extracting features that describe different aspects of the data, one increases the amount of data used as input to the model, in hopes of increasing model performance. In this case, the raw datasets are sequences of CDS and n-LORFs, as described in Chapter 3. The feature engineering process involves identifying properties of sequences that can be used as features, transform them into numerical vectors, which are then to be learned by both the Random Forest and XGBoost model.

The features created for the Random Forest and XGBoost model were largely inspired by the features used in the the article by Silva et al. [2]. The same features were created, however using a different approach in cases such as the codon bias feature. Additional features were also created such as amino acid usage. In Figure 4.1 the 17 sets of features extracted from each genome dataset are presented. In the subsections below a short description of each feature set is given and an explanation as to why it was chosen.

GC-content

The GC-content of an ORF is the percentage of G and C nucleotide base pairs found in the ORF. As stated by Al-Ajlan and El Allali [35], GC-content is a well-known feature that has been used by several machine learning based gene prediction tools such as Orphelia [29] and Metagenomic Gene Caller (MGC) [36]. In this thesis GC-content values for each ORF were extracted using the "G+C Content" function in the "SeqinR" R-package. This function computes the overall GC-content of an ORF as well as the GC-content of all the base pairs in the first, second and third position reading frames of the same ORF. Hence, in total this function can return 4 different GC-content values.

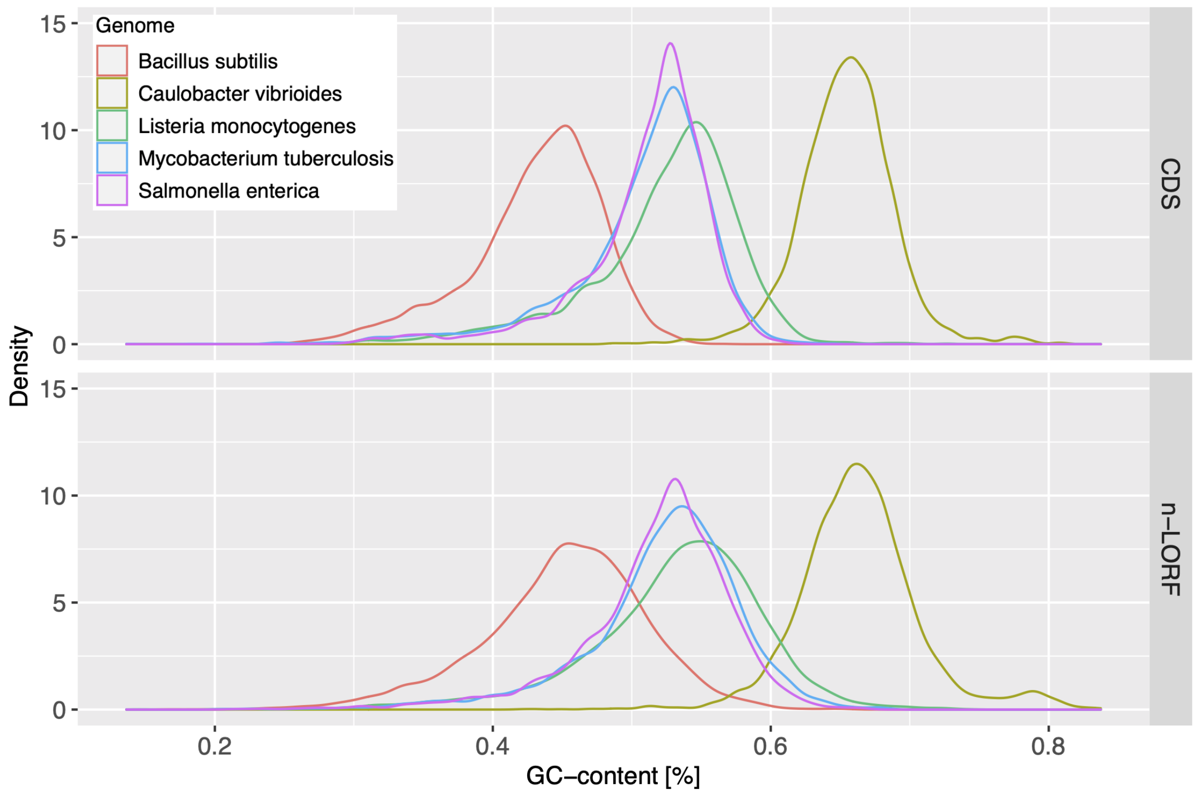


Figure 4.2: Density plot showing the difference in the density distribution between CDS and n-LORFs in 5 genomes given their overall GC-content which includes all frames.

GC-content is an interesting feature to include because it both varies within the same genome and also between genomes. This is clearly seen in Figure 4.2. When comparing the two subplots one can see that the density distributions of ORFs with different GC-content varies within the same genome depending on if one is looking at the coding gene (CDS) or the n-LORFs of a particular genome. All the genomes in the subplot showing the CDS have a slimmer and taller density distribution than the genomes in subplot showing n-LORFs. This indicates that the CDS have a greater number of sequences with high GC-content than n-LORFs. By including GC-content as a feature one may be able to determine if the GC-content can play a role in distinguishing between CDS and n-LORFs.

Some argue that there is a correlation between GC-content and the use of synonymous codons (codon bias) [37]. Other argue that there is a relationship between the GC-content of an ORF and the CDS lengths [38]. Pozzoli et al. [39] state that there is a directly proportional relationship between GC-content and CDS length due to the fact that stop-codons have a bias towards A and T nucleotides. Thereby, the shorter the ORF is, the higher the AT bias. The three stop-codons can be found in Table 2.1 in Chapter 2.

Length

The ORF-length is simply the number of bases that makes up the ORF. In resemblance with GC-content, the ORF-length is also a well-known feature, often used by state-of-the-art gene predictions tools with good reason. There is clearly a strong correlation between length and coding genes, which is visible in Figure 4.3. Looking at the two subplots it is evident that the length distribution in CDS across different genomes is much more stable than the length distribution in n-LORFs across the same genomes. Chapter 3 explains how a sufficient amount of data preprocessing was done on each genome dataset due to the differences in the length distributions between CDS and n-LORFs. Before the data preprocessing the differences in length distributions between CDS and n-LORFs were even greater.

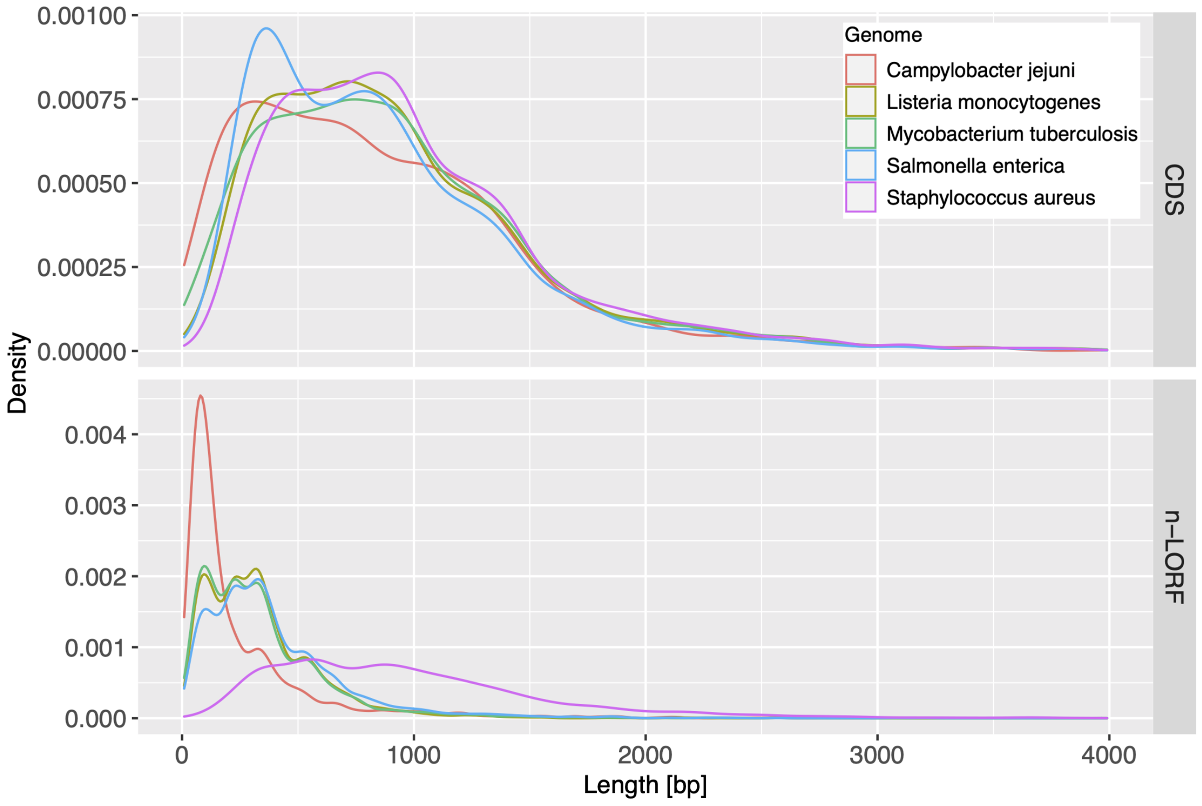


Figure 4.3: Density plot showing the length distribution of CDS and n-LORFs in 5 different genomes, after data preprocessing. The length is measured in number of base pairs [bp].

K-mer Frequency

As presented in Figure 4.1 the k-mer frequencies computed were: 2-mer, 3-mer, 4-mer, 5-mer and 6-mer. A k-mer is simply a substring of nucleotides with a length of k. When one calculates a 2-mer frequency for instance, one simply sums up the number of times a given 2-mer is observed in an ORF. Since there exist only 4 different bases, A, T, C and G, the number of possible k-mers for a given k would be: 4^k. Al-Ajlan and El Allali [35] compute only the monocodon usage and the dicodon usage of each ORF, but Silva et al. [2] compute all k-mers from 2 to 6. In this thesis the latter option was chosen during feature engineering. Including k-mers as features means one can determine whether there are certain k-mers that are more represented in CDS and if they are useful in the prediction of CDS.

Amino Acid Usage

The amino acid usage follows the same principles as k-mer frequencies. One simply translates the ORF and counts the frequencies of the k-amino acid occurrences in the ORF. Since there exist 20 amino acids and one stop codon, the number of possible k-amino acids for a given k would be: 21^k. 1-, 2- and 3-amino-acid-mers were computed for each ORF in a genome. The amino acid-mer features could potentially have better predictive power than the k-mer features as they essentially contain the same information, but without as much redundancy. In other words, the redundancy in the genetic code is eliminated by using amino acid sequences, making the sequences stable and more closely related to proteins. By including amino acid-mers as features one could determine whether the particular sequence of amino acids in a sequence is a better indicator of the presence of coding genes than the codon sequence.

Codon Bias

Codon bias, also referred to as codon weight by some, is defined differently in many articles. In Silva et al. [2] the codon bias is simply the monocodon frequencies in the ORF. In this paper however, another approach has been used to estimate the codon weight. First the monocodon frequencies of each ORF is found as well as the respective amino acid frequencies of the translated ORF. Then each codon is given a weight by dividing the frequency of a specific codon by the total number of times the corresponding amino acid is found in that particular sequence. For each sequence, every codons is given its respective weight. These set of features then indicates if certain synonymous codons are preferred over others in a given genome.

Start Codon Usage

As mentioned in the Chapter 2 there are three possible start codons for ORFs in prokaryotic genomes. These are ATG, TTG, and GTG. ATG is the most common start codon. The start codon feature simply indicates which start codon is used in a given ORF with 1 or 0 if the start codon has not been used. By including this feature one could potentially get an indication if start codon usage can be used for the purpose of gene prediction.

4.2 Random Forest Classifier

The first model that was tested on the genome-data was the Random Forest classifier. This model was implemented using the machine learning library **scikit-learn**. Theory about Random Forest is given in Chapter 2, Section 2.2.2.

Implementing a Random Forest classifier involves several stages: feature engineering, training and testing, tuning and feature selection. Figure 4.4 is a rough visualisation of the stages in the prediction pipeline, from having raw sequencing data and until classification. The datasets presented in Chapter 3 were used as training and test data for the Random Forest classifier presented in this section. The Random Forest classifier makes predictions based on signals captured from the ORFs, which can be used as input in the model. These signals are the features presented above in Section 4.1.

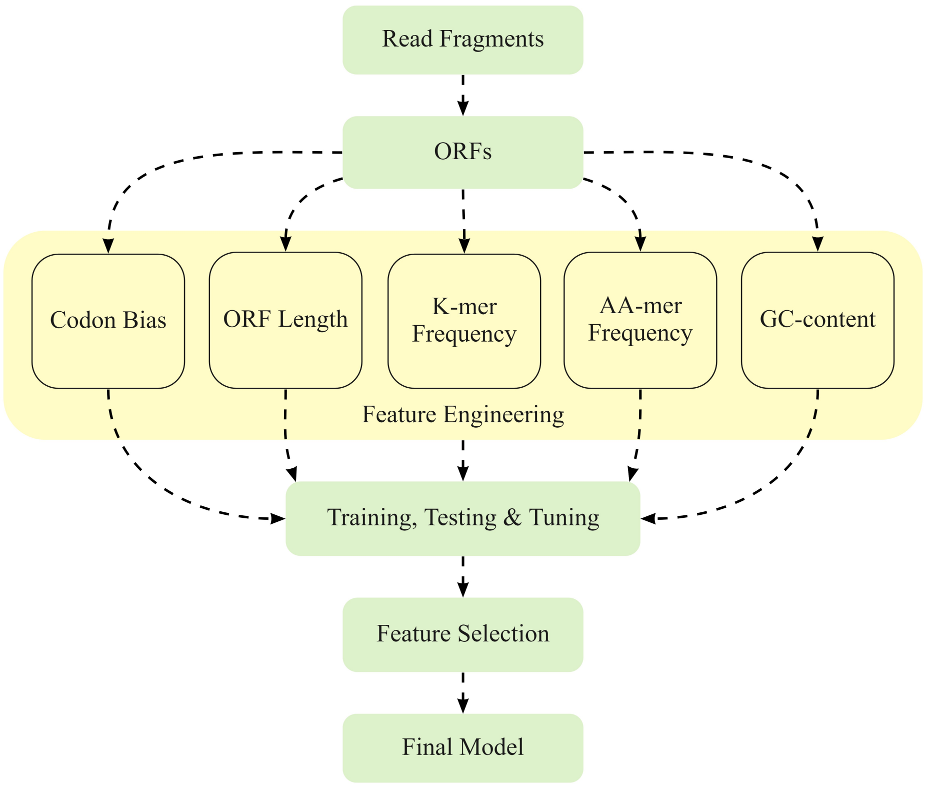


Figure 4.4: Illustration of the stages during the implementation of a Random Forest classifier, from raw sequence data and until a final model. The figure is inspired by *Fig.1* by Al-Ajlan and El Allali [35].

4.2.1 The Input Datasets

The Random Forest classifier was trained and tested multiple times with various input datasets which will be presented in this subsection. In Chapter 3 two datasets for each genome are presented. The first dataset is attempted balanced in terms of length distribution between the two classes. The other dataset is attempted balanced in terms of length distribution, while also remaining balanced in the number of CDS and n-LORFs. The Random Forest classifier was trained and tested on both the datasets for all 15 genomes individually. This was done to determine if the difference in balance between the datasets affects the performance of the Random Forest classifier. After determining this, further exploration of the classifier continued using only the dataset variation that gave the best performance. The training of the classifier, on all genomes individually, was also done in order to see how performance varies depending on the genome. Following this, the classifier was trained on a combination of genomes. This will be explained in more detail in Section 4.5. The Random Forest classifier was the only classifier that was trained on pairs of genomes. This was done to see if there may be an advantage to training genomes that are difficult to classify together with genomes that are easier to classify, as opposed to training them separately.

4.2.2 Hyperparameter Tuning

Since scikit-learn was used to implement this model one could simply run the model with the default parameters and get reasonable results. However, one could potentially improve these results further by tweaking the hyperparameters according to the dataset. Koehrsen [18] state that the most important parameters to tune for a Random Forest classifier are the number of trees in the forest and the number of features that are considered at each node. The hyperparameters that were tuned are listed below. These were inspired by Koehrsen [18].

- **n_estimators**: number of trees in the forest
- **max_features**: maximum number of features to consider when looking for the best split
- **max_depth**: the maximum number of levels of a tree
- **min_samples_split**: minimum number of samples required to split an internal node
- **min_samples_leaf**: minimum number of samples required to be at a leaf node
- **bootstrap**: method for sampling datapoints [40].

There are various approaches to hyperparameter tuning, in this case, scikit-learn's GridSearchCV and RandomizedSearchCV were both tested. The difference between these two methods is that GridSearchCV systematically tests all the combinations of hyperparameters defined in a grid, while RandomizedSearchCV only selects a combination at random. Koehrsen [18] recommends starting with a RandomizedSearchCV approach and then narrow down the search further with a GridSearchCV afterwards. RandomizedSearchCV tends to be much faster, and is therefore often preferable when dealing with large datasets. Since this was the case with the datasets used for this model, this approach quickly became the favoured one. Both approaches were initially tested on the first genome from species *M. tuberculosis.* After yielding similar results with both approaches only the RandomizedSearchCV class was used when tuning hyperparameters with other genome datasets.

Hyperparameter	Values tested	Result
Number of estimators	min: 10, max: 1000, num: 10	200
Maximum features	auto, sqrt	auto
Maximum depth	min: 10, max: 1000, num: 10	20
Minimum samples for split	2, 5, 10	2
Minimum samples for leafs	1, 2, 4	1
Bootstrap	True, False	False

Table 4.1: Representation of the hyperparameters tuned using RandomizedSearchCV, the span of values tested for each parameter, and the optimal hyperparameters (results) found during tuning.

RandomizedSearchCV

When hyperparameter tuning with the RandomizedSearchCV one first defines a grid or distribution of values for the parameters that one wishes to tune. During training, different combinations of parameters are randomly sampled from the grid, while performing a K-fold cross validation for each combination. When running a machine learning model, it is common practice to hold out a part of the dataset for testing in order to avoid overfitting the model. When training a model multiple times with different combinations of parameters, the same problem of overfitting can occur. For this reason one implements cross-validation methods while tuning. K-fold cross validation is used during a randomised search, and how it works is explained in Chapter 2. During the randomised search performed in this thesis a 3-fold cross validation was used with 10-50 iterations.

Table 4.1 presents the six hyperparameters that were tuned, the span of values tested for each parameter, and the last column shows the optimal hyperparameters selected for the final Random Forest classifier.

4.2.3 Feature Selection

After tuning the Random Forest classifier the next attempt to further improve the accuracy of the model was feature selection. As mentioned in Chapter 2 there are different methods for feature selection, depending on what type of model one is building. The Random Forest classifier has its own built in feature selection method, which falls under the category of embedded methods [41]. The selection of features is based on the feature importance computed by the decisions trees created when building the model. In Chapter 2 it is stated how each decision tree in the Random Forest classifier is built on a selection of arbitrary features. At each decision node in a tree, the data is divided into two groups based on their similarity to each other, given a particular feature. The importance of each feature is thereafter determined by how pure each leaf node is, after such a split. Gini impurity or information gain are the functions used to measure the quality of a split in classification problems. A pure leaf node is an end node that only contains data from one of two target classes in a classification problem. In other words, the importance of each feature is the ability of a given feature to separate a subset of ORFs into CDS and n-LORFs. All the feature importance scores are normalised so that they sum up to 1 for each dataset. An advantage of using the feature importances computed during training of the Random Forest model, is that the importance are computed simultaneously with model training. A disadvantage on the other hand, is that the decision trees have a tendency to favour features with high cardinality [42]. By high cardinality we mean features that have a large number of distinct values. The decision trees also have a tendency to give correlated features similar scores that are lower than what they would be if the model was trained without correlated counterparts. In the sections below, the two different approaches chosen for feature selection will be discussed. Both are based on the feature importance's computed by the tuned Random Forest classifier.

Scikit-learn SelectFromModel

Scikit-learn has a class called SelectFromModel which automatically extracts the best features from a model based on their feature importance weights, measured against a given threshold. This is a straightforward approach where one simply provides the estimator one wishes to use, which in this case was the Random Forest classifier, including its tuned parameters, and the threshold at which one wishes to discard features. Examples of thresholds can be the median value of all the feature importance's, the mean, or the median times a scaling factor for instance. In this model the mean feature importance was used as a threshold. This means features having a feature importance below the mean value were discarded. The feature selection method was trained and tested on all genomes, and then compared in terms of the accuracy scores with the feature selection method presented below.

Manual Feature Selection

In addition to the method above, manual feature selection was also attempted. This simply involved training the tuned Random Forest classifier with a range of features, that were selected based on their feature importance. The testing range spanned from 100 - 1400 features, with an increment of 100. In other words, all the features were arranged in descending order of feature importance, after which the given number of features were selected, always starting with the features with highest importance. After running this on multiple datasets the metric scores were compared. Finally a conclusion regarding the best number of features was made, based on both the best metric score and also on the degree of complexity. The greater the number of features in a model, the more time consuming and resource-intensive the model is to run. After a conclusion was made the tuned Random Forest classifier was trained and tested again on all genomes with only the selected number of best features.

4.3 XGBoost

After attempting various combinations of datasets, genomes and feature selection methods with the Random Forest classifier, the ensemble method XGBoost was next in turn. Theory regarding the XGBoost method is found in Chapter 2, Section 2.2.3. The incentive for implementing an XGBoost classifier was to see how it performed in comparison to the Random Forest classifier and the RNN model. The implementation was through the scikit-learn library, and no hyperparameter tuning was done, only the default XGBoost classifier was used. The same set of features used for the Random Forest classifier were used for the XGBoost classifier, presented in Section 4.1. Cross-validation and early stopping was also used for some datasets, without significant improvement in performance. A manual feature selection was done based on feature importance, identical to the manual feature selection process done for the Random Forest classifier. Then a new model was trained and tested for each genome with the selected features. The XGBoost classifier was also trained and tested on both the two types of datasets. These results were very similar to the Random Forest classifier, and the results will therefore not be presented in Chapter 5. Finally, the model was tested on a combination of genomes. This will be discussed further in section 4.5.

4.4 Recurrent Neural Network

The RNN model was implemented using the **Keras** [43] library through **TensorFlow** [44]. Theory regarding the general structure of RNNs is found in Chapter 2, Section 2.3. In this section the preprocessing of input data, architecture and tuning process executed during implementation of the RNN will be presented.

4.4.1 Preprocessing

The same datasets as presented in Chapter 3 are used for the RNN model. The first dataset is balanced in terms of the length distribution of the positive and negative instances. The second dataset is balanced in terms of the length distribution as well as having a balanced number of positive and negative instances. As with the ensemble methods, the RNN model was tested on both these datasets for all 15 genomes. When using these datasets as input to the RNN model they required much less preprocessing and no feature engineering as opposed to the ensemble methods. This is because the sequences themselves are used as input directly. Figure 4.5 a), demonstrates what a single raw input sequence looks like. In order to feed these sequences to the RNN they must be made numeric. This was done using one-hot-encoding for the target variables and tokenisation followed by a padding of zeros for the predictor variables. The padding needed to be done since there is a large variation in the lengths of the ORFs in each dataset. Each sequence was padded until its length was equivalent to the longest ORF found in that particular dataset.

a) ATGAAAAATAGTGTCGCTGAGCACTAA

b) MKNSVAEH*

Figure 4.5: a) A single raw base input sequence. b) The same input sequence, but translated to its respective amino acids.

A translated version of the nucleotide sequences used as input was also used as input to the RNN model. By translating the sequences to their respective amino acids one wishes to minimise the redundancy in the sequences. Translation also makes the sequences considerably shorter and thereby potentially makes it easier and faster for the RNN model to perform a classification. However, even though the dimension of the vector becomes much shorter in one direction, the number of tokens during tokenistaion increases from 4 to 21 as there are 21 different amino acids and only four different bases. This in turn makes the sequences longer in another direction. Figure 4.5 b), demonstrates what the translated input sequence looks like.

4.4.2 Architecture

Figure 4.6 is a simple illustration of the architecture of the RNN model. The first layer is an *embedding layer*. Embedding layers are commonly used in *Natural Language Processing (NLP)* as a method to deal with textual data. According to Saxena [45] it can be thought of as an alternative to one-hot-encoding along with dimensionality reduction. In this case, the purpose of the embedding layer is simply dimensionalty reduction, since the sequences already have been tokenised. The input dimension in the embedding layer is 22, since there are 21 tokenised amino acid symbols, plus a zero due to padding.

The second layer is the recurrent layer. The activation functions used in this layer are the default functions in keras LSTM layers. That is the ”tanh” function for feed forward activation and the ”sigmoid” function for recurrent activation, meaning the output that is memorized.

The third layer is the *dropout layer*. Dropout layers are used in neural networks to prevent overfitting. This is done by randomly setting the output of certain nodes to zero at a given rate, during each training iteration. This in turn makes the loss function more sensitive to the remaining nodes which affects the way the weights are updated during back-propagation [46]. The last layer is a dense layer consisting of only one node, which is also called the output layer. Since the problem at hand is a classification problem, the activation function used in the output layer is the ”sigmoid” function. This was explained in more detail in Section 2.3.3.

4.4.3 Tuning

Before starting to tune the model, both nucleotide sequences and amino acid sequences were used as input to the RNN model and compared in terms of training time and performance metrics, using a dataset from species *M. tuberculosis*. In addition, different maximum lengths of the sequences and different number of epochs for each type of input data were also compared. When using the amino acid sequences the RNN model was much faster and achieved better performance scores. The tuning of the hyperparameters in the model was therefore done using only the translated sequence data.

The maximum length of each sequence used as input to the RNN model was manually tuned to see how it affected the speed of the model training as well as if it had an effect on the performance. The maximum length parameter is used in the function that ”pads” the sequences before they are passed as input to the RNN. By *padding* one refers the process of extending sequences that are shorter than a given length by adding zeros to the beginning or end of the

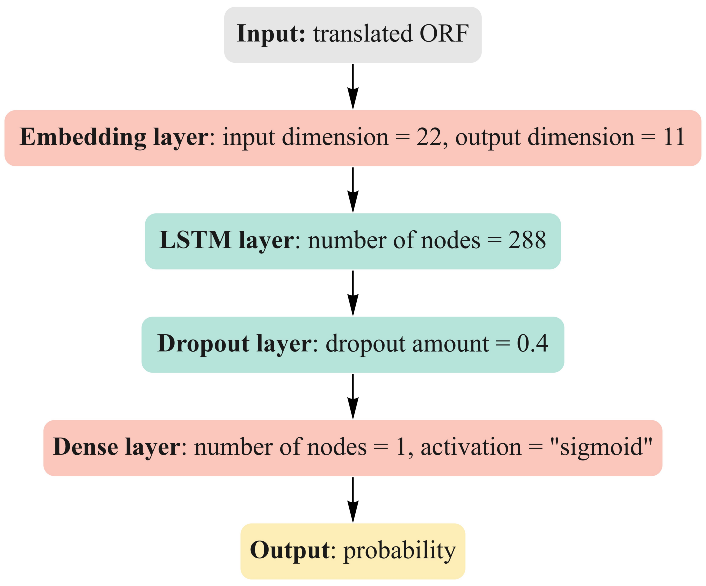

Input: translated ORF
Embedding layer: input dimension = 22, output dimension = 11
LSTM layer: number of nodes = 288
Dropout layer: dropout amount = 0.4
Dense layer: number of nodes = 1, activation = "sigmoid"
Output: probability

Figure 4.6: Simple illustration of the layers in the final RNN model. Layers coloured in red (embedding layer and dense layer) contain parameters that have not been tuned. Layers coloured in green (LSTM layer and dropout layer) contain parameters that have been tuned.

sequences. This needs to be done in order for the RNN to be able to take the sequences as input. Another parameter in this function determines whether the sequence is to be padded from the beginning or the end of the sequence. In this case, the sequences were padded from the start of the sequences. For the raw sequence data a span of approximately 1000 to 8000 maximum lengths were tested, and for the amino acid sequences a span of approximately 200 to 2000 maximum lengths were tested. Each length was tested three times at the same number of epochs and the final result was the average of the three replications. Since the training of the RNN model was quite fast when using the amino acid sequences, the maximum length of the sequences was finally decided to be set the same length as the longest ORF in a given dataset. This also gives a closer depiction of the length distribution of ORFs in an unaltered genome.

The number of epochs used during training was also manually tuned for each type of input data. The preferred number of epochs used varied depending on if one used the nucleotide sequence data or if one used the translated sequence data. A span of 15 to 50 epochs was tested when using the translated sequence data. The number of epochs that gave the best performance-time trade-off was 15 epochs. For the raw sequence data, a span of 15 to 500 epochs were tested, of which 500 epochs gave the best performance-time trade-off. The different number of epochs were tested three times for both models, using the same maximum length of sequences. The average of the three replications was used as the final result. The manual tuning of both epochs and maximum lengths was also done using genome data from species *M. tuberculosis*. The results from the comparisons of different maximum lengths and the number of epochs, when using different types of sequence inputs will be presented in Chapter 5.
After selecting the best type of data input, as well as manually tuning the maximum length and epochs, the hyperparameters of the RNN model were tuned using the **Keras Tuner** library. The complete source code for the tuning process is found on GitHub [47]. Table 4.2 shows the four hyperparameters that were tuned, as well as the span of values that were tested for each parameter. The number of nodes in the LSTM layer was tuned in the span of 32 - 512 with a step size of 32. The dropout rate in the dropout layer was tuned in the span 0 - 0.9. Three different loss functions were tested, binary cross entropy, hinge, and squared

Hyperparameter	Values tested	Result
Number of nodes	min: 32, max: 512, step: 32	288
Dropout rate	min: 0, max: 0.9, step: 0.1	0.4
Learning rate	$10^{-1}, 10^{-2}, 10^{-3}, 10^{-4}, 10^{-5}$	10^{-4}
Loss function	binary cross entropy, hinge, squared hinge	binary cross entropy

Table 4.2: Representation of the hyperparameters tuned by Keras Tuner and the span of values tested for each parameter. The optimal hyperparameters chosen for the final model are presented in the last column labelled as ”results”.

hinge. According to Brownlee [48] these loss functions are appropriate for binary classification problems. Finally, the optimiser learning rate was tuned using values in the span 0.1 - 0.00001. Two optimiser functions were considered during tuning, ADAM and RMSprop. They were both tested and tuned separately, before a comparison was made which resulted in the selection of RMSprop as the preferred optimiser. Even though the learning rate is initially set at a certain value, and treated as a hyperparameter, the two optimisers used will update the learning rate under training, as mentioned in Chapter 2. Therefore, the learning rate is regarded as both a hyperparameter and a trainable parameter.

The RNN was tuned on datasets from three species, *M. turberculosis*, *C. burnetii* and *E. coli str. K-12*. The datasets from these exact genomes were chosen due to their dissimilarities in CDS and n-LORF percentage. The optimal hyperparameters selected by the tuner varied greatly depending on the species. Therefore, the hyperparameters for the final model were chosen to be an average of the three tuning results. The optimal hyperparameters chosen for the final model after tuning are shown in the ”Results” column of Table 4.2.

When fitting the final RNN model the batch size was set to 32. The batch size parameter limits the number of samples that are used for training the model at a time. 32 and 64 are common batch sizes. The number of epochs was set to 20 and the validation split parameter was set at 0.2. This means that 20% of the input data is used to validate the model during training. This model was then trained separately and used to classify the datasets from all the 15 genomes individually, as well as on combinations of genome datasets. This will be explained in more detail in the following section, and the results will be presented in the following chapter.

4.5 Combining Genomes

The four different combinations of genomes that the models were tested on were a combination of 10 genomes, a combination of 5 genomes, a combination of the 5 best performing genomes and a combination of the 5 worst performing genomes. The combination of the genomes were made based on how the models performed on the genomes individually. One of the challenges when training and testing the models on combinations of genomes was that when combining more than two genomes, the datasets became very large and difficult to handle. This was solved by taking a random sample from each genome dataset, without replacement, and then combining them. When combining 10 genomes a random sample of 2000 ORFs were taken from each of the 10 datasets, and when combining 5 genomes a random sample of 4000 ORFs were taken from each dataset. The genome datasets that contained less than 2000 ORFs, such as the genome dataset of species *S. flexneri*, were used as they were. The performance results of all the three models on the different combinations will be presented in the following chapter.

Chapter 5

Results

This chapter will present and compare the results obtained by the three models tested, while addressing the three objectives introduced in Chapter 1. Chapter 3 and Chapter 4 discuss why two differently balanced datasets were created to train the three models. The dataset that is balanced only in terms of the length distribution of positive and negative instances will be referred to as *dataset 1*. The dataset that is balanced both in terms of the length distribution as well as the number of positive and negative instances will be referred to as *dataset 2*.

5.1 Random Forest Classifier

As presented in Section 4.2, implementing a Random Forest classifier involves several stages. The results obtained when testing different datasets, from hyperparameter tuning, feature engineering and feature selection will be presented in this section. This includes feature importance and the difference between feature selection techniques. The performance of the classifier will also be compared when using different genomes as input, as well as combined genomes.

5.1.1 Comparing Datasets

The results presented below were created using scikit-learn's default Random Forest classifier. With "default", one means that no hyperparameter tuning was done at this stage, only the parameter values set by scikit-learn version 0.23.2 were used. This default model was used to train and test dataset 1 and dataset 2 for all 15 genomes. Figure 5.1 compares how the three evaluation metrics vary when using dataset 1 as opposed to dataset 2 as input to the Random Forest model, for all genomes. The difference in accuracy score between the two datasets is very minor. The lines in the two plots follow an almost identical path. In contrast to the accuracy score, the precision and recall metrics are clearly affected by which datasets that is used as input. Some genomes are also more affected by the different datasets than others. The genomes from species *S. enterica*, *L. monocytogenes*, *C. jejuni*, *S. flexneri*, *P. aeruginosa* and *C. burnetti* have significantly lower precision and recall scores when using dataset 1 compared to when using dataset 2. Other genomes from species such as *S. aureus* and the *E. coli*, show little to no difference in all evaluation metric scores. From the two plots one can also see that there is a large difference in performance metrics from one genome to another. The metrics used to evaluate model performances in this thesis are explained in Chapter 2, Section 2.4.

The confusion matrices shown in figure 5.2 presents the results when using dataset 1 and dataset 2 from the genome of species *L. monocytogenes*, as input to the default Random Forest classifier. The confusion matrices of this specific genome were of particulate interest as the precision and recall scores were considerably different in the two plots for this species. In the matrix for dataset 1 we can see that there is a considerable difference in the distribution of true positive and true negative values compared to the matrix for dataset 2. There is also a difference in the distribution of false positives and false negatives in the two datasets. In the following results, only dataset 2 is used unless something else is specified.

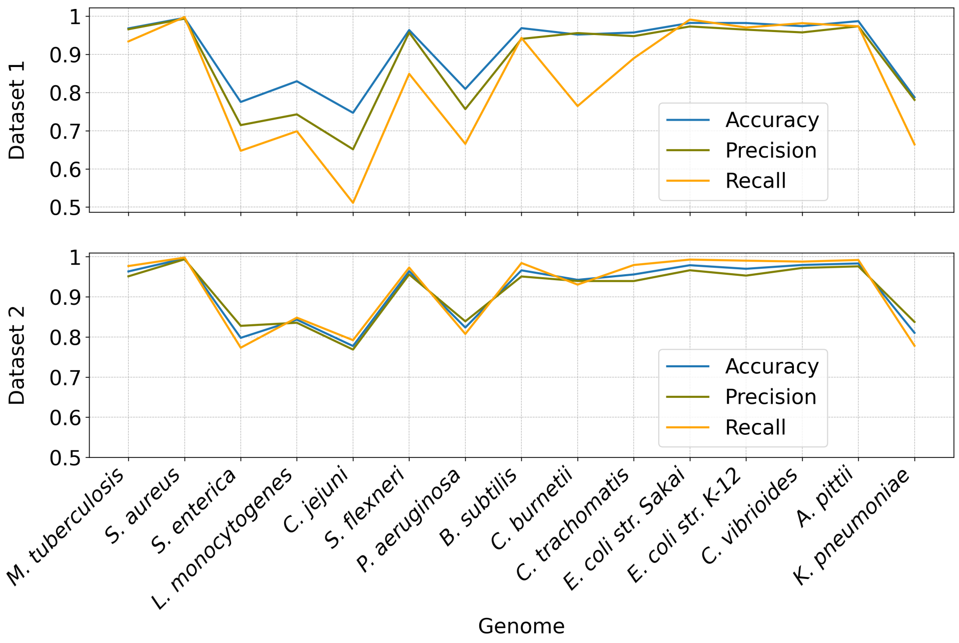


Figure 5.1: Comparison of the accuracy, precision and recall score for the Random Forest classifier given dataset 1 and dataset 2 as input for each of the 15 genomes. The x-axis for both plots are the same genomes, in the same order.

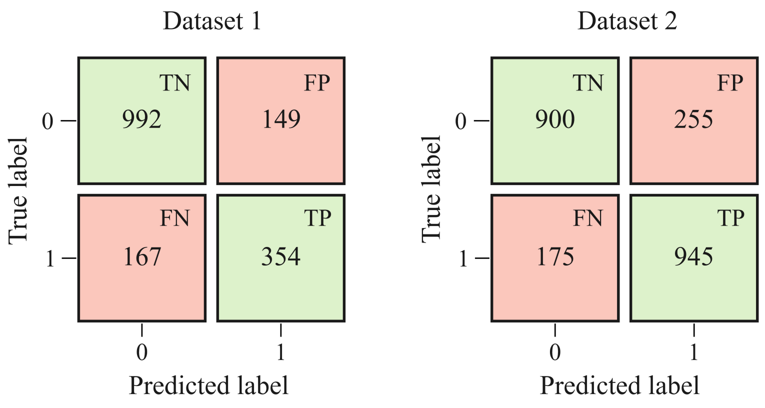


Figure 5.2: The confusion matrices present the results when using dataset 1 and dataset 2 from the genome of species *L. monocytogenes*, as input to the default Random Forest classifier. "0" corresponds to predicting a n-LORF, and "1" corresponds to predicting a CDS.

5.1.2 Default vs. Tuned Random Forest Classifier

In this section the performance improvement after hyperparameter tuning will be presented. The parameters that were tuned using RandomizedGridSearch are presented in Section 4.2.2. Figure 5.3 compares the accuracy score before and after tuning of the Random Forest classifier. The tuning improves the performance only marginally. The genomes with lower accuracy scores are affected more by the tuning than the features with a higher score. Figure 5.4 presents the results from tuning of the number of trees in the forest, as well as how the number of trees chosen affects the training time. After reaching 100 tress the increase in validation accuracy score stops. The training time on the other hand continues to increase steadily as the number of trees increases. The fewer the number of trees one needs to get a good validation score, the faster it takes to train the model.

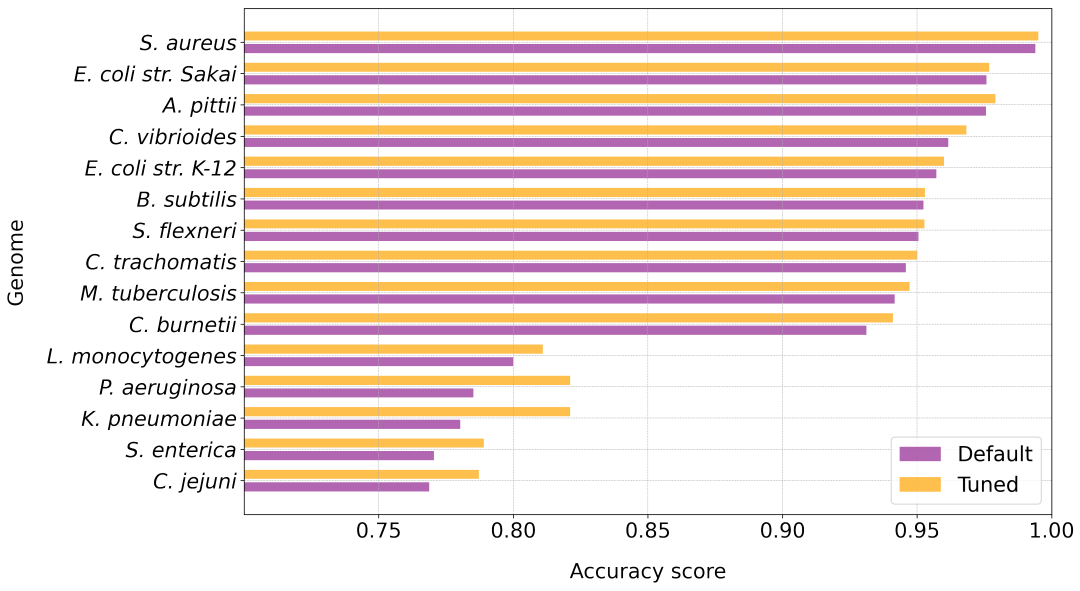


Figure 5.3: Plot presenting the accuracy score of the default Random Forest classifier compared to the tuned model, trained and tested on all genomes, one at a time.

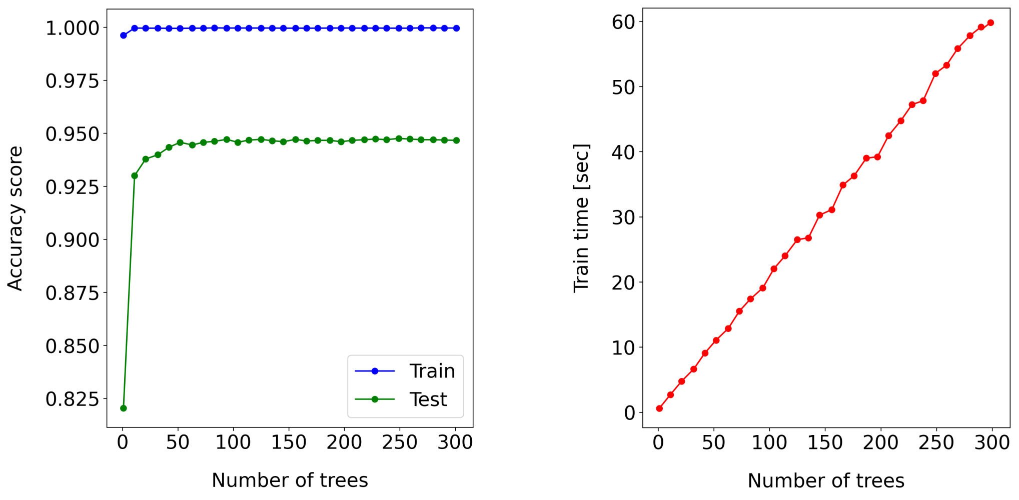
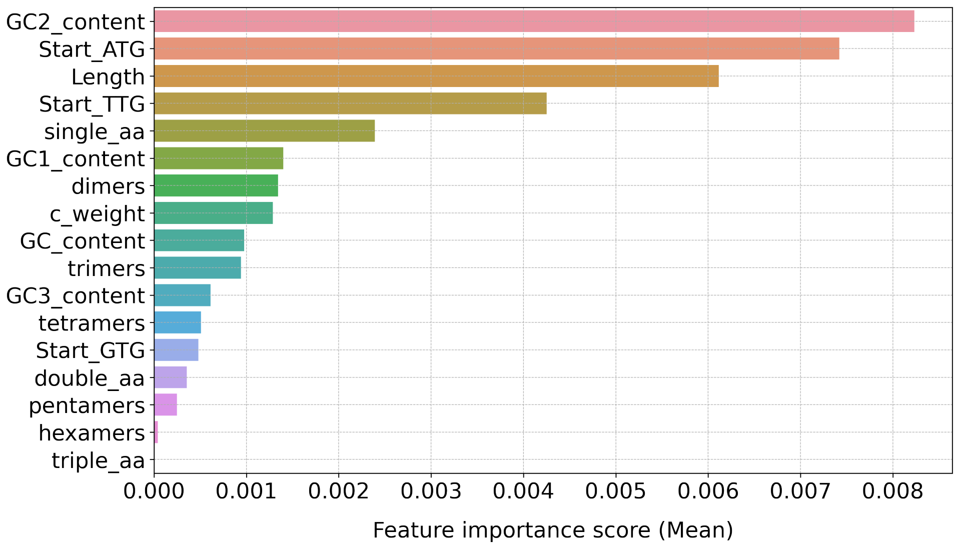


Figure 5.4: Plots showing how the number of trees in a Random Forest classifier affects the accuracy (left plot) and the training time (right plot) of the Random Forest classifier. The blue points in the left plot is the accuracy score of the training data and the green points are the accuracy scores for the validation data.

5.1.3 Feature Importance

As mentioned in Chapter 4, 17 feature sets were engineered to be used by the Random Forest and XGBoost classifiers. Feature importance is the score that the Random Forest classifier



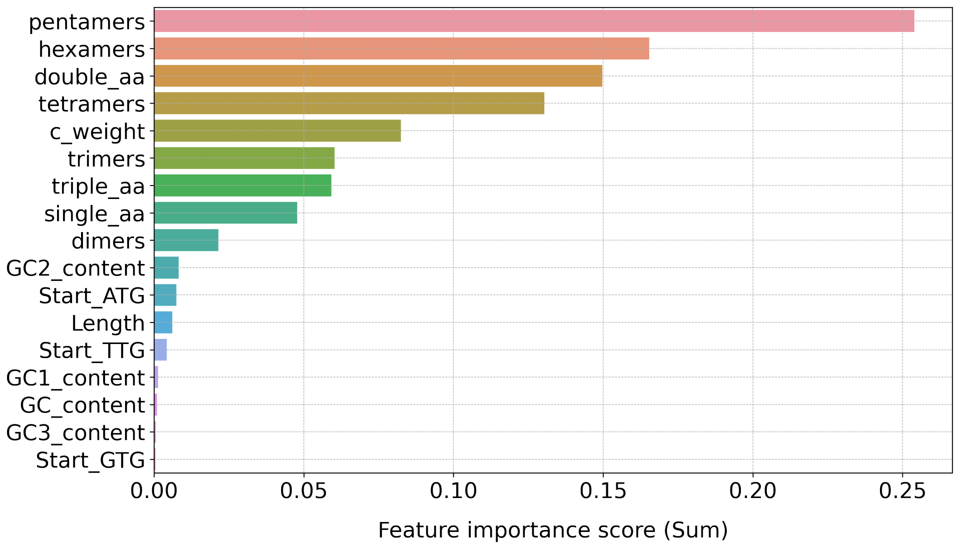


Figure 5.5: Plots showing the feature importance scores for all 17 feature sets, computed during training of the Random Forest classifier on the genome of species *M. tuberculosis*. The first plot computes the mean score of all features in a set, while the lower plot takes a sum.

gives each feature during training. This score gives an indication of how important a given feature is to the model when making gene predictions, and the scores for all these features sum to 1. Since 9 of the 17 features sets consist of many feature computations, such as the k-mer count features and codon bias, it is difficult to present the feature importances of all the feature sets in their condensed form. Figure 5.5 presents two alternative methods of presenting the feature importance for the genome of species *M. turberculosis*. The first plot is simply created by computing the mean of all feature importances in each feature set. However, this means the feature sets that only consist of one computation, such as the ”GC-content” features, ”length” and the ”start codon” features, get an unfair advantage. Despite this, some k-mer features such as the ”single-aa” feature which is a count of the single amino acid usage, still receives a relatively high score. This goes for ”dimers” (2-kmer) and the ”c_weight” (codon bias) features as well.

The second plot in Figure 5.5 presents each feature set’s importance score as a summation of all importance values within that feature set. This approach, as opposed to the ”mean score”

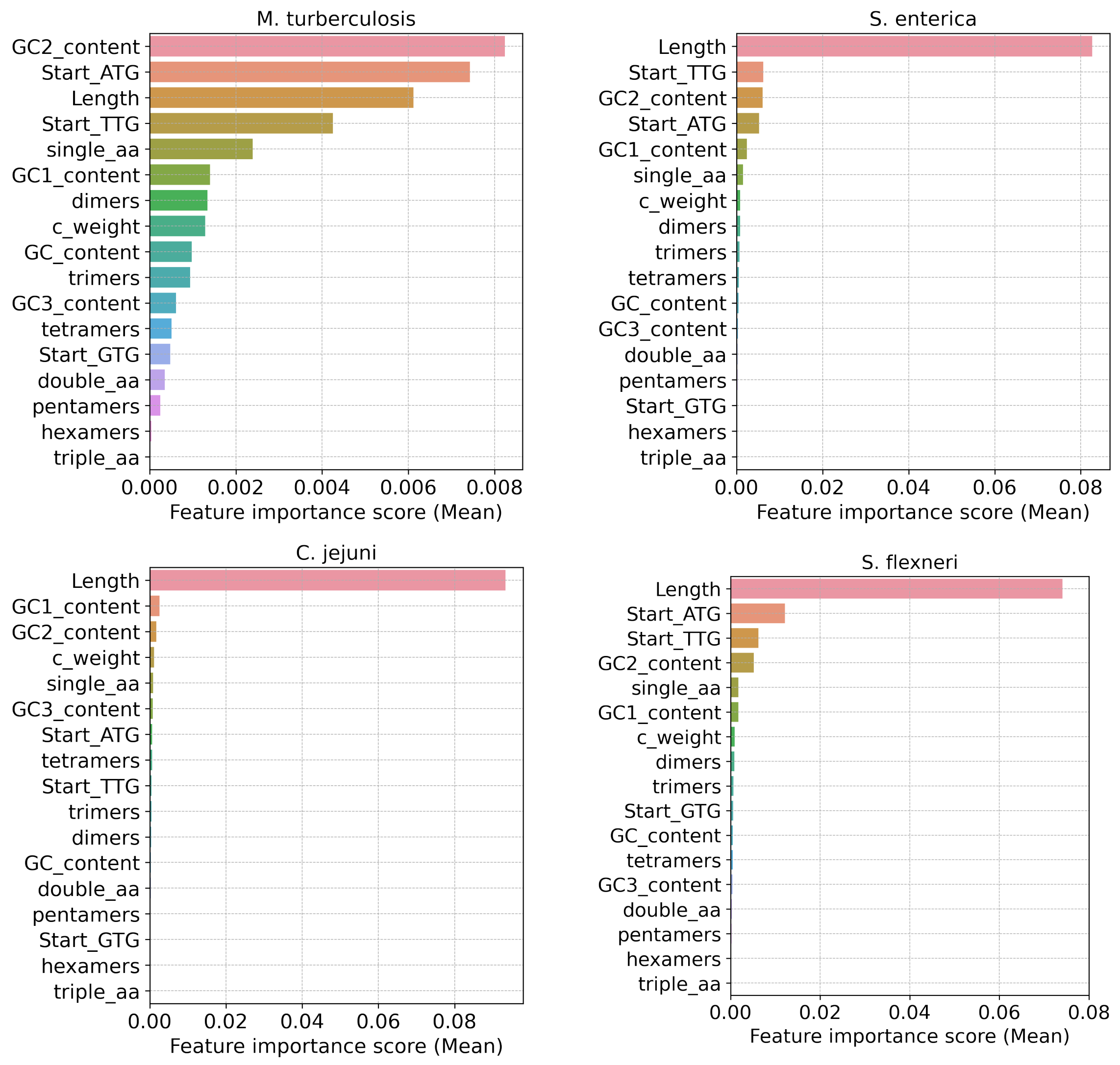


Figure 5.6: Plots presenting feature importance scores for all 17 feature sets computed using the "mean score" approach, for the four genomes of species *M. tuberculosis*, *S. enterica*, *C. jejuni*, and *S. flexneri*.

presented above, gives an unfair advantage to the feature sets that consist of multiple feature computations. All the features that have multiple computations score considerably better than the ones with only one. Neither of the two approaches are optimal, but the "mean score" approach seems like the most fair approach of the two.

Figure 5.6 presents the mean score feature importance for four selected genomes. These genomes vary in size and the Random Forest model classified each of them with varying performance success. *M. tuberculosis* is a large genome and the Random Forest classifier scored well with this genome as input. *S. enterica* is also a large genome, but got one of the lowest accuracy scores out of the 15 genomes when classified with the Random Forest model. *C. jejuni* is a small genome and was classified well by the Random Forest model. *S. flexneri* is also a small genome and was classified poorly by the Random Forest classifier. Feature importance scores clearly vary greatly depending on the genome used as input to the Random Forest classifier. The top five feature sets in each of the four plots are different. *M. tuberculosis* is the only genome out of the four that does not have the "length" feature as its most important feature. In total, 11 out of the 15 genomes had the "length" feature as its most important feature in the "mean score" feature importance plots. In addition, the magnitude of the feature importance score of the ”length” feature was often a lot greater than the remaining feature sets, as seen in the plots of *S. enterica*, *C. jejuni* and *S. flexneri* in Figure 5.6.

5.1.4 Feature Selection Techniques

Based on the feature importance scores computed by the Random Forest classifier during training, a selection of features was made using both scikit-learn's ”SelectFromModel” estimator as well as a manual feature selection technique.

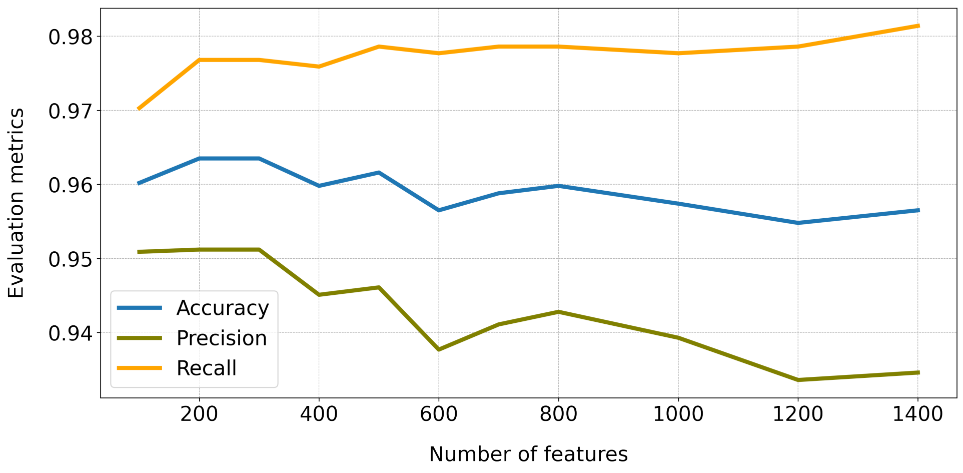


Figure 5.7: Plot presenting how increasing the number of features affect the three evaluation metrics. The features are selected based on the best feature importance score.

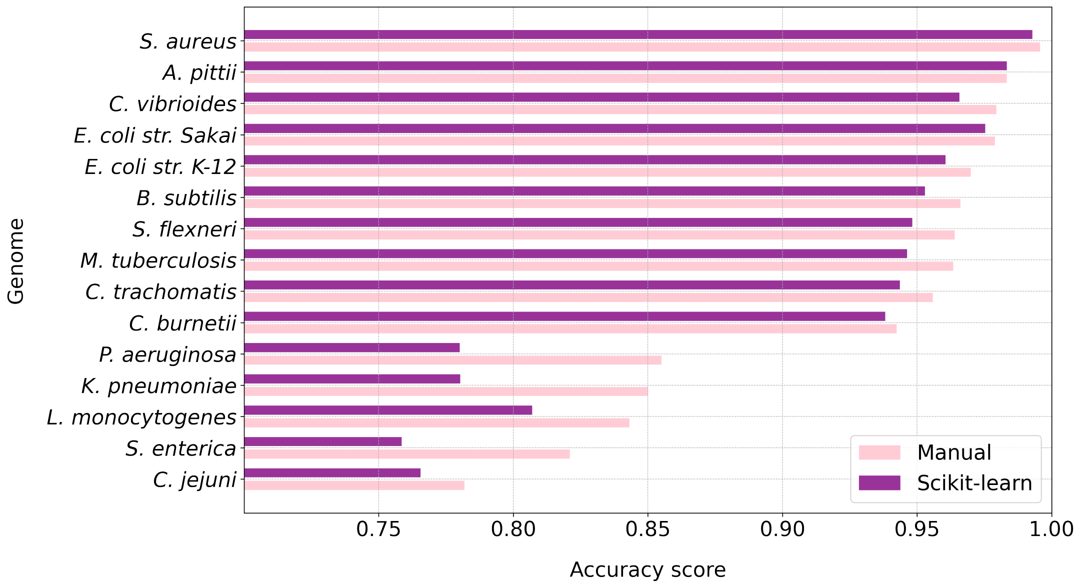


Figure 5.8: Plot presenting the accuracy score of the tuned Random Forest classifier with features selected using the scikit-learn technique, compared to manual feature selection.

The manual selection technique is shown in Figure 5.7, and one can see that the three evaluation metrics varies depending on the number of features that are used for the Random Forest classifier. The plot shows that the accuracy and precision scores are the highest between 200 to

300 features after which they start decreasing as more features are included. The recall score on the other hand increases gradually as the number of features included increases.

The difference in accuracy score when using scikit-learns ”SelectFromModel” feature selection technique compared to the manual feature selection technique is shown in Figure 5.8. From the plot one can see that the manual feature selection technique quite clearly yields better results than the scikit-learn technique.

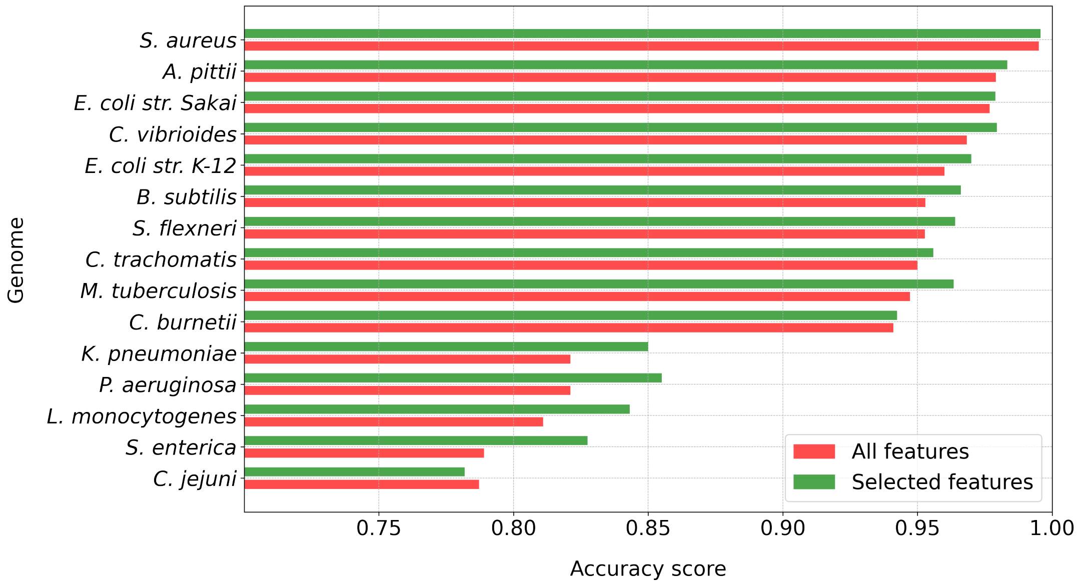


Figure 5.9: Plot presenting the accuracy score of the tuned Random Forest classifier with all features, compared to the tuned Random Forest classifier with only the selected best features.

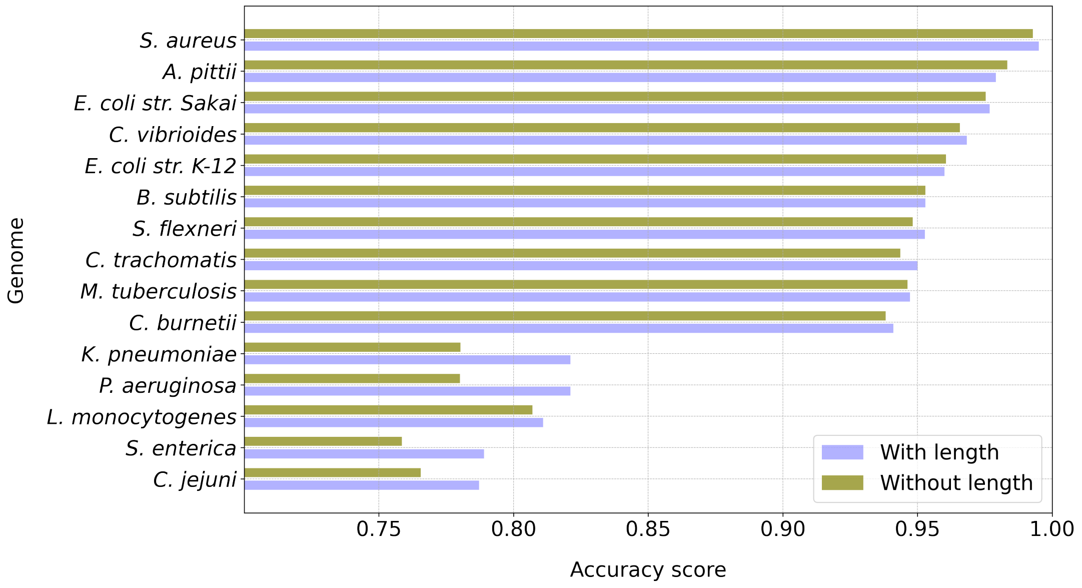


Figure 5.10: Plot presenting the accuracy score of the tuned Random Forest classifier with all features compared to a tuned model that is trained and tested without the feature ”length”.

A comparison of the accuracy score of the tuned Random Forest classifier with all features, and the tuned Random Forest classifier with only the 200 most important features is shown in

Figure 5.9. As one can see from the plot, the model with selected features performs better than the model with all features for almost all the 15 genomes.

The feature "Length" is historically known to have much importance in gene prediction [49]. One has also learnt from the feature importance plots above that the feature is given high feature importance by the Random Forest classifier for most of the 15 genomes classified. Figure 5.10 shows how the performance of the Random Forest classifier is affected if one removes only the feature "length". From the plot one can see how the absence of this feature affects the accuracy score for some genomes and not for others. The four genomes from species *S. enterica*, *C. jejuni*, *P. aeruginosa* and *K. pneumoniae* show a distinct drop in accuracy, while other genomes are affected significantly less. These results correlate well with the feature importance graphs shown in Section 5.1.3.

5.2 XGBoost

A selection of the results from the methods described in Chapter 4, Section 4.3 are presented in this section. The XGBoost classifier was tested on dataset 1 and 2 for all genomes, manual feature selection was attempted, and it was tested on different combinations of genomes. The comparison of dataset 1 and 2 will not be presented, as the results were very similar to the Random Forest classifier. The main objective of implementing the XGBoost classifier was to see how its performance is comparable with the Random Forest classifier and the RNN.

5.2.1 Feature Selection

Since manually selecting features based on feature importance for the Random Forest classifier yielded better accuracy scores, the same method was attempted with XGBoost. In Figure 5.11 the accuracy scores of a default XGBoost model with all features compared to a default XGBoost model with only the 200 features with best feature importance is presented. As one can see there is close to no improvement in accuracy after manual feature selection. For the *S. enterica* genome the accuracy is in fact higher for the XGBoost model using all features.

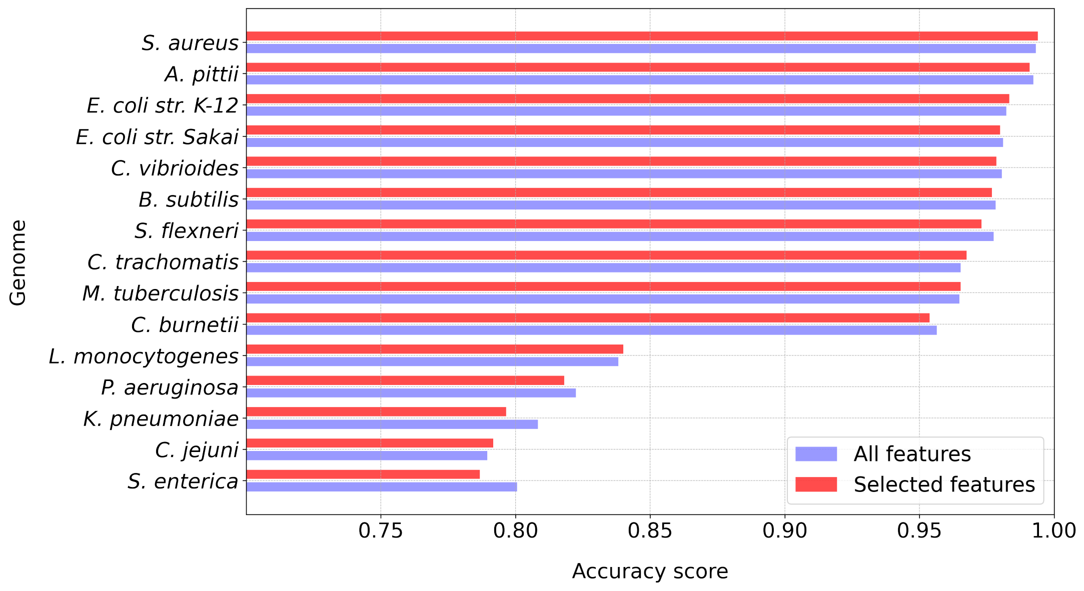


Figure 5.11: Plot presenting the accuracy score of the default XGBoost classifier with all features, compared to the default XGBoost classifier with only a selection of features.

5.2.2 Default XGBoost vs. Default Random Forest Classifier

Since no parameter tuning was done on the XGBoost model, and feature selection did not yield significant improvement in accuracy the default XGBoost classifier is considered the best version of the XGBoost model. By default one means an XGBoost classifier using the default parameter settings in the scikit-learn implementation. Figure 5.12 compares the default XGBoost model with the default Random Forest classifier, before any tuning or feature selection was performed. The plot shows that the default XGBoost classifier performs significantly better than the default Random Forest classifier.

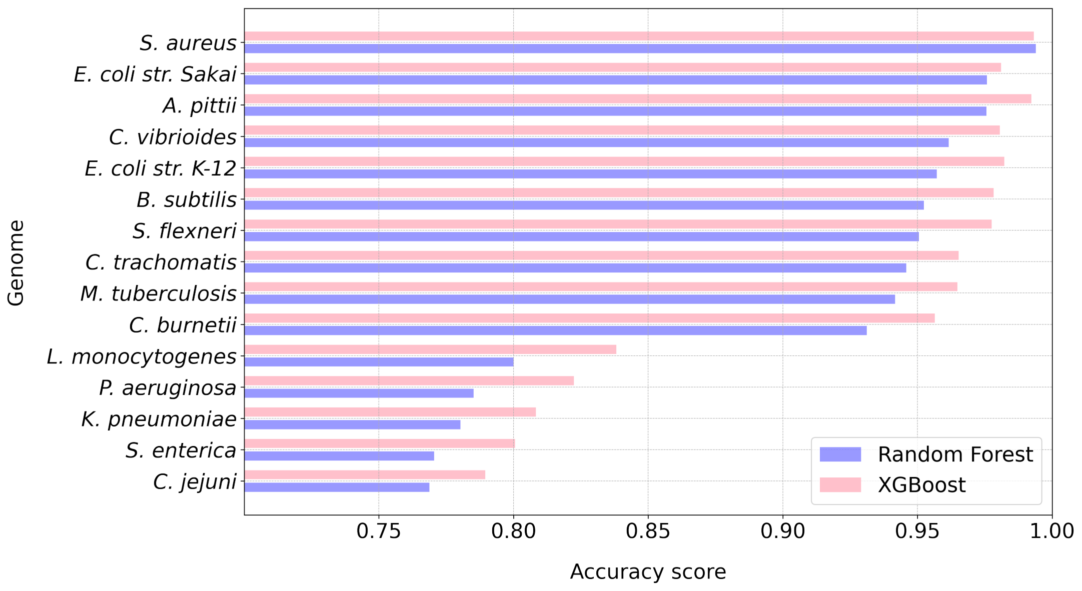


Figure 5.12: Plot comparing the accuracy score of the default Random Forest classifier with the default XGBoost classifier.

5.3 Recurrent Neural Network

As outlined in Chapter 4, Section 4.4 the RNN was tuned manually in terms of the maximum length used for the input sequences in the neural network, and the number of epochs used. In addition, both raw nucleotide sequences as well as translated amino acid sequences were tested as input. Finally, hyperparameter tuning was done using the Keras tuner. These results will be presented in this section, in addition to a comparison between dataset 1 and 2 and how the RNN performs on a combination of genomes.

5.3.1 Nucleotide Sequence Data vs. Amino Acid Sequence Data

Two things were established early while getting started with the RNN model. First, it was decide whether to use the raw nucleotide sequences as input or amino acid sequences. Second, it was decide which maximum sequence length yields the best performance-time trade-off. A description of the maximum length parameter is given in Chapter 4, Section 4.4.3. The longer the sequences, are the longer the training time is. Table 5.1 and Table 5.2 shows the difference in number of epochs, accuracy score and the time used by the RNN model to train the same dataset when using raw nucleotide sequences compared to amino acid sequences. The results when using amino acid sequences are clearly better, as one achieves a better accuracy score, fewer epochs are required and the training time is shorter. These results were computed with the maximum length set to approximately half of the length of the longest sequence in the genome. This was a length of 4000 bases for the raw nucleotide sequences and 1000 amino acids for the amino acid sequence, when using the genome from species *M. tuberculosis*.

Table 5.1: Metrics for the RNN model using nucleotide sequences as input and different number of training epochs.

Epochs	Accuracy	Time
15	0.6237	3 min 59 sec
50	0.7786	12 min 24 sec
100	0.8399	23 min 46 sec
500	0.9069	1 hour 44 min

Table 5.2: Metrics for the RNN model using amino acid sequences as input and different number of training epochs.

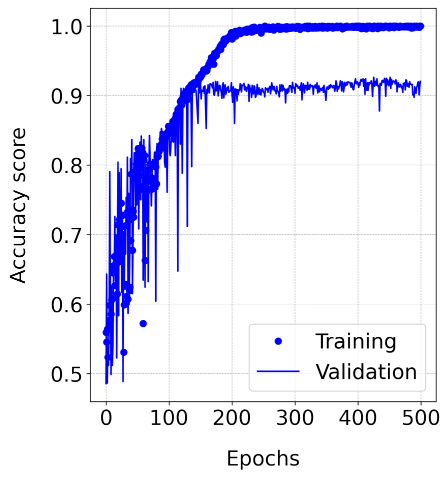
Epochs	Accuracy	Time
15	0.9393	1 min 30 sec
20	0.9422	2 min 02 sec
30	0.9410	3 min 04 sec
50	0.9353	4 min 07 sec

Table 5.3: Metrics for the RNN model using nucleotide sequences as input and different number of maximum lengths.

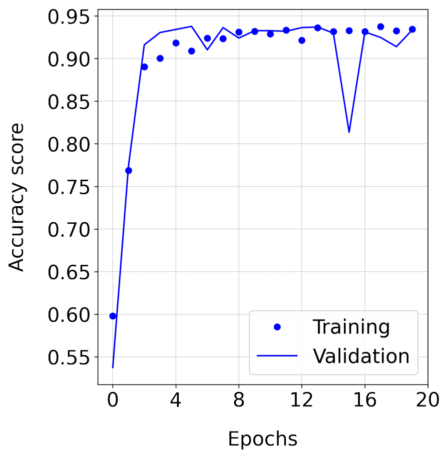
Max length	Accuracy	Time
1000	0.8745	11 min 20 sec
2000	0.8566	11 min 20 sec
3000	0.8364	16 min 40 sec
4000	0.8399	22 min 49 sec
5000	0.8516	27 min 34 sec
6000	0.8310	32 min 31 sec
full	0.8235	37 min 55 sec

Table 5.4: Metrics for the RNN model using amino acid sequences as input and different number of maximum lengths.

Max length	Accuracy	Time
200	0.9242	1 min 02 sec
400	0.9395	1 min 10 sec
800	0.9376	1 min 40 sec
1200	0.9291	2 min 01 sec
1600	0.9241	2 min 05 sec
2000	0.9204	2 min 40 sec
full	0.9237	3 min 05 sec



(a) Nucleotide sequences.



(b) Amino acid sequences.

Figure 5.13: Training and validation accuracy for an RNN model using nucleotide sequences as input (left plot), compared to a model using amino acid sequences as input (right plot).

Table 5.3 and 5.4 present the results when testing various maximum lengths for the raw nucleotide sequences and for the amino acid sequences respectively. The different maximum lengths for the raw nucleotide sequences were all trained at 100 epochs and the maximum lengths tested for the amino acid sequences were trained 15 epochs. The genome of species *M. tuberculosis* was used also in this case. The longest nucleotide sequence in this genome dataset is 8649 nucleotides, which corresponds to 2359 amino acids when translated. In the last row in both tables the maximum length of the sequences for each model is set to ”full”. This means it is set to the length of the longest nucleotide or amino acid sequence in the genome. Again one can see that using amino acid sequences as input results in much faster training time than using nucleotide sequences. There seems to be a slight downward trend in accuracy as the maximum length is increased, especially for the model using amino acid sequences as input.

The training and validation accuracy score when training two RNN models with each sequence input type is shown in Figure 5.13. Figure 5.13a is from the training of the model with raw nucleotide sequences as input. It was trained for 500 epochs, with a maximum sequence length of 4000 nucleotides. Figure 5.13b is from the training of the model using amino acid sequences as input. It was trained for 20 epochs and with a maximum sequence length of 1000 amino acids. In Figure 5.13a one can see that the model only stabilises around 200 epochs. In Figure 5.13b on the other hand, the model stabilises quite quickly, despite the fact that considerably fewer epochs were used. The time used and final accuracy achieved by both models is presented in Table 5.1 and Table 5.2.

5.3.2 Comparing Datasets

A comparison of the accuracy score of the RNN model when using dataset 1 and dataset 2 as input is shown in Figure 5.14. In Figure 5.1 we saw that there was not much difference in the accuracy score when comparing the two datasets with the Random Forest classifier. In Figure 5.14 on the other hand, we can observe that there is a significant reduction in accuracy for some genomes when using dataset 1 as opposed dataset 2. The genomes from the species *C. jejuni* and *S. flexneri* score significantly worse with dataset 1.

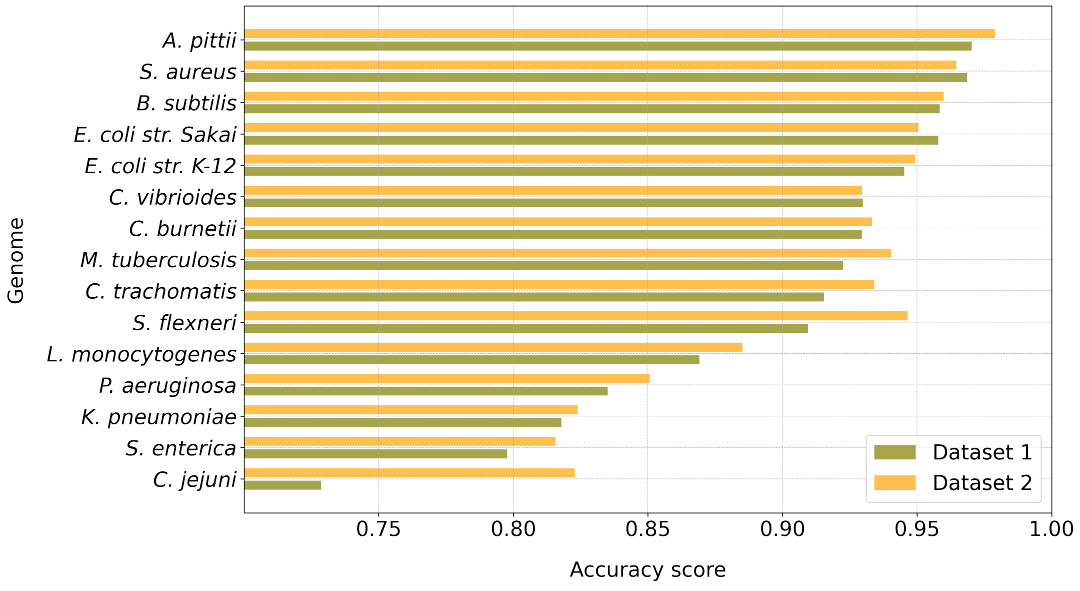


Figure 5.14: Plot comparing the accuracy score of the RNN model given dataset 1 and dataset 2 as input for each of the 15 genomes.

5.4 Comparing the Methods

Figure 5.15 presents a comparison of all three models that have been discussed and tested in this thesis. The best performing versions of the models have been used in this comparison.

When comparing the default version of the Random Forest classifier with XGBoost in Figure 5.12, one could see that the XGBoost model yielded the better results. In 5.15, the best performing version of the Random Forest classifier is used, with tuned hyperparameters and selected features, as opposed to the default version. The difference in performance between the Random Forest classifier and XGBoost in Figure 5.15 is considerably less prominent than in Figure 5.12. In fact, the two ensemble methods seem to perform quite equally. The RNN model on the other hand shows a slightly different pattern than the two ensemble methods. It scores worse for genomes where the ensemble methods score very well and slightly better for the genomes where the ensemble methods score poorly.

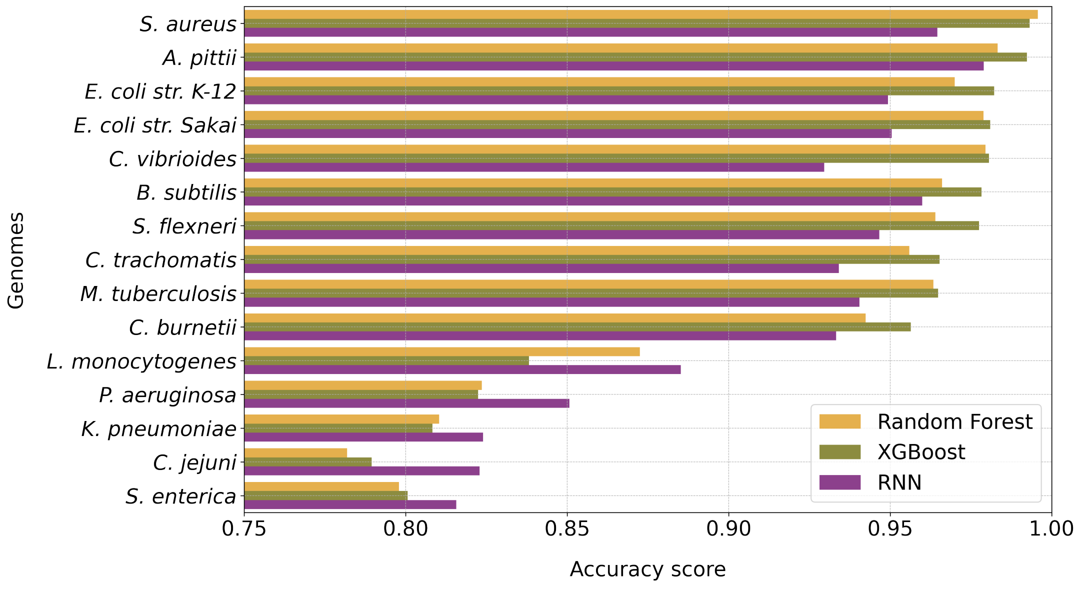


Figure 5.15: Plot presenting the difference in accuracy score between the best performing Random Forest, XGBoost and RNN model. The models were trained and tested on all 15 genomes.

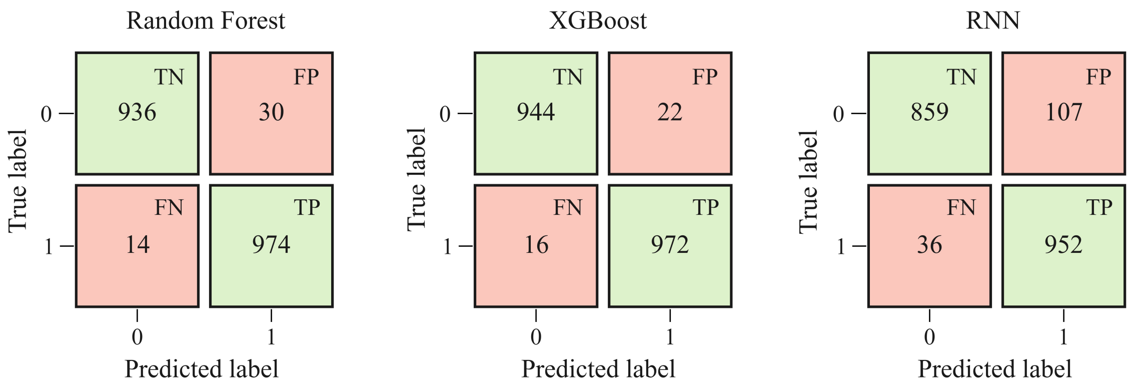


Figure 5.16: The confusion matrices display the results from the Random Forest, XGBoost and RNN model, after classifying data from the genome of species *C. vibriodes*. "0" corresponds to predicting a n-LORF, and "1" corresponds to predicting a CDS.

In Figure 5.16 one can see three confusion matrices presenting the results of all three models when classifying the genome data from species *C. vibriodes*. The confusion matrices shows how the three models compare in the number of predictions that were false positive, false negative, true positive and true negative. This particular species was chosen due to it being a genome where the ensemble methods performed better compared to the RNN, which can be confirmed in Figure 5.15. The RNN model shows a significantly higher number of false positives than the ensemble methods in addition to a lower amount of true negatives. It also has more than double the amount of false negative predictions compared to the ensemble methods.

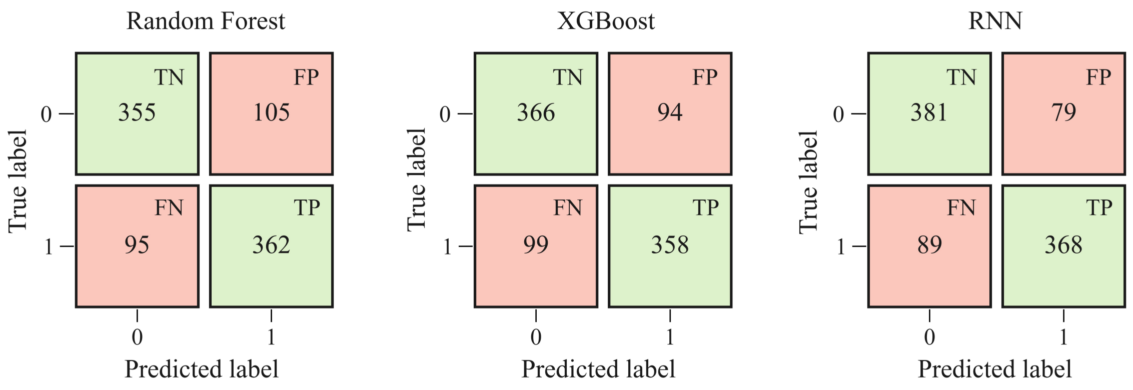


Figure 5.17: Confusion matrices for each of the three models when classifying data from the genome of species *C. jejuni.* "0" corresponds to predicting a n-LORF, and "1" corresponds to predicting a CDS.

The confusion matrices for the three models when classifying the genome data from species *C. jejuni* is presented in Figure 5.17. For this species the RNN model performed better in terms of accuracy score than both the ensemble methods, as seen in Figure 5.15. In this case the ensemble methods have slightly lower true positive and true negative predictions than the RNN, in addition to larger false positive and false negative predictions. The Random Forest classifier has a significantly higher number of false positive predictions compared to the other two models.

5.4.1 Combined Genomes

Until now we have trained and tested all models on data from one genome at the time. As mentioned in Chapter 4, Section 4.5, all the models were also trained and tested on a combination of genomes. The different combinations were chosen based on how the models performed on each genome. As one can see in Figure 5.15 all three models get a lower accuracy score on the same 5 genome datasets. These are the genome datasets of species *S. enterica*, *L. monocytogenes*, *C. jejuni*, *P. aeruginosa* and *K. pneumoniae.* The combination of the 10 genomes included four of these genomes in addition to six genomes where all the models received a good accuracy score. The combination of the five genomes had three of the poorly classified genomes and two of the better classified genomes. Finally, all three models were trained and tested on the 5 genomes with the best scores, and the 5 genomes with the worst accuracy scores. The results of the models performance on all four combinations of genomes is presented in Figure 5.18. The figure is a two way comparison, as the solid coloured bars show how the three models perform compared to each other for the different combinations of genomes. The bars that are lighter in colour, on the other hand, present the performance average of the same genomes that are combined in each of the four combinations. This average is calculated after the models have been individually trained and tested on the different genomes.

In the figure one can see how the ensemble methods (yellow and green bars) always perform slightly better than the RNN when classifying the various combinations of the genomes. At least the Random Forest classifier. The accuracy score for each combination of genomes coincides quite well with the value one gets when taking an average of the scores of each genome

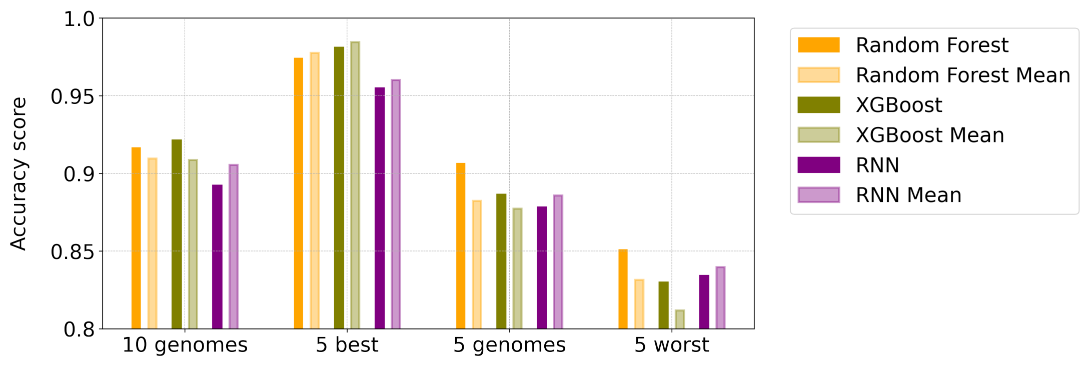


Figure 5.18: Plot presenting a to-way comparison of the accuracy score for the Random Forest classifier, XGBoost classifier and RNN model when trained and tested on four different combinations of genomes (fully coloured bars). As well as their average score when trained on the same genomes individually (lightly coloured bars).

when classified individually (lightly coloured bars). The ensemble methods consistently seem to perform better when trained on a combination of genomes, as opposed to individually. Except for when they are trained on the combination of the five best genomes. The RNN on the other hand performs worse when trained on all the combinations of genomes compared to when it is trained individually. Even though the genome combination consisting of 10 different genomes has double the number of genomes as the combination consisting of only 5 different genomes, all three models still get a better accuracy score when classifying the combination of 10 genomes.

The Random Forest classifier was also trained on pairwise genomes. The genomes that were combined were *M. tuberculosis* and *S. enterica*. These exact genomes were chosen because the Random Forest classifier gets a good accuracy score when classifying *M. tuberculosis* and a significantly worse score when classifying *S. enterica*, as we can see in Figure 5.15. The two genomes are similar in size and in CDS and n-LORF percentage, as seen in Table 3.1. Table 5.5 shows how training the genomes together compares to training them individually in terms of their individual accuracy scores. Training the genomes together seems to improve the performance of the Random Forest classifier when classifying genome data from species *S. enterica*. The performance when classifying *M. tuberculosis* is not affected greatly by being trained together with genome data from speceis *S. eneterica*.

Trained	M. tuberculosis	S. enterica
Combination	0.9621	0.8396
Individually	0.9634	0.8275

Table 5.5: Presents the performance metrics for the Random Forest classifier when classifying genome data from species *M. tuberculosis* and *S. enterica*. The column ”Trained” indicates whether the Random Forest classifier was trained on a combination of the genome data from the two species, or individually.

Chapter 6

Discussion

The findings from the previous chapter will be discussed in this chapter, with the objectives presented in Chapter 1 in mind. Possible explanations to why these results may have occurred will also be presented, as well as some of the limitations with the models tested and possible future directions of work.

6.1 Random Forest Classifier

The first objective presented in Chapter 1 addresses the question of which subset of predefined and engineered features yield the highest performance of a Random Forest classifier. These results, among other things, are discussed in the following subsections, in the same order as they were presented in Chapter 5.

6.1.1 Comparing Datasets

Figure 5.1 presents the results of the three performance metrics for the Random Forest classifier, after using dataset 1 and dataset 2 to classify all 15 genomes. The precision and recall metrics both drop significantly when dataset 1 is used as opposed to dataset 2. From Chapter 3 one can recall that dataset 1 is balanced in terms of length distribution of the CDS and n-LORFs only. Since the majority of the CDS sequences are long and the majority of the n-LORF sequences are short, dataset 1 contains considerably fewer n-LORFs than CDS. This means the dataset is imbalanced in terms of the number of positive and negative instances. The reason why this type of imbalanced dataset was tested, is due of the fact that most sequences over a certain length are much more likely to be CDS than n-LORFs. Far more of the longer LORFs found in a genome are CDS than n-LORFs, and far more of the shorter LORFs are n-LORFs. Therefore, creating a dataset that is more balanced in terms of the length distribution between the two classes is an attempt to make it less likely that the longer sequences in all the dataset are CDS. In this case however, it unfortunately means that one gets an imbalanced dataset in terms of the number of instances of each class, which entails its own disadvantages. When handling imbalanced datasets the evaluation metrics may react differently depending how the true positive, true negative, false negative and false positive values are affected by the imbalance. This is seen in Figure 5.2. There are much fewer true positive classifications when using dataset 1 as opposed to dataset 2. In addition, there are more false negatives than false positives when using dataset 1. This difference clearly does not affect the overall fraction of classifications that our model classified correctly (accuracy score), but it does, on the other hand affect the metrics' precision and recall. This can be explained by looking at the definitions of the metrics precision and recall presented in Section 2.4. To summarise, when evaluating models using imbalanced datasets one may need to look at different metrics to measure the performance of a classification model.

Even though the accuracy score conveys that there is not much difference in the performance when using dataset 1 or dataset 2 as input, the other evaluation metrics say otherwise.

Even though using dataset 2 gave the best results between the two datasets, it does not necessarily mean training our model on this dataset will give the best results if the trained model was given a new dataset that was balanced differently. Due to time limitations, neither of the three models implemented in this thesis were tested on a dataset that had neither been balanced in terms of the length distribution or the number of each class. This could potentially have given an impression of how the models perform on a more realistic length and class distribution. A different strategy could also have been to train the models on dataset 1 and 2, as well as on all the ORFs in a genome, and then test which model gives the best results when given a new dataset with a realistic length and class balance.

6.1.2 Feature Importance

Since 9 of the 17 feature sets used for the ensemble methods consisted of multiple features (k-mers, aa-mers and the codon biases), it was difficult to present the feature importances for complete feature sets without bias. The method chosen to compare the feature importances in this thesis was by taking a mean score of the importances of the feature sets that consisted of multiple features. As mentioned in the Chapter 5 this gives an unfair advantage to the feature sets that only consist of one feature. The second method involved taking a sum of the feature sets consisting of multiple features, which on the other hand gives an unfair advantage to the feature sets consisting of many features. An alternative to these two approaches could be to use the *Jaccard index*. The Jaccard index is a similarity index that allows one to measure the similarity of the members in two sets of data. In this case, it would involve extracting the 100 features with highest importance after training separate models for each genome. These are then compared and one computes a Jaccard index based on how many of the 100 features each genome has in common.

As may be recalled from Chapter 2, Section 4.2.3 the feature importance computed by the Random Forest classifier has certain bias towards features with high cardinality or high correlation. In this case it would mean that features such as "length", "GC-content" and "codon bias" with high cardinality have an advantage over features such as "start codon usage" which has low cardinality. In addition, features such as "aa-mers", "k-mers" and "GC-content" are likely to be highly correlated which also gives them a disadvantage in the Random Forests feature importance computations. Features with high importance scores and low cardinality should therefore be regarded as more trustworthy. An alternative method to computing feature importance is the feature permutation importance method. This gives an importance value to each feature based on its impact on the error of the model. The importance values are computed by testing a trained model with different permutated datasets iteratively. Each dataset has one missing or inactivated feature. The error scores from each iteration are compared with a baseline score where all features were included in the dataset. If the model error is unchanged without a certain feature, this feature is not regarded as important. On the other hand, if the error increases the feature is regarded as important. The disadvantage with this method is that it is slightly more time consuming if there are many features, considering that the number of predictions made with the model must equal the number of features. An alternative to this method is simply dropping one feature from the dataset and training and testing the model without the given feature. This implies training the model and making predictions as many times as one has features, which is even more time-consuming. Despite this it is still the gold-standard for computing trustworthy feature importances.

6.1.3 Feature Selection

During the manual feature selection process we learned that the accuracy and precision scores are the highest between 200 to 300 features for the Random Forest classifier. This was shown in Figure 5.7 in Chapter 5. The recall score on the other hand increases constantly as the number of features included increases. This makes sense considering recall is a measure of the number of correctly identified CDS in comparison to all existing CDS in the dataset. It seems as if the more features one has, the higher the probability is of correctly identifying a CDS. When picking the optimal number of features to include in the Random Forest classifier, one must remember there is a trade-off between the best performance metrics and model complexity. An overly complex model not only increases training time but also makes the model more prone to overfitting, and more difficult to interpret. From Figure 5.7 the optimal number of features seemed to be the first 200 considering this trade-off. By making a selection of features to be used for training, one gets a more parsimonious model, as the remaining features get more value and predictive power.

The greatest improvement in performance of the Random Forest classifier was established after feature selection. This turned out to yield better performance results than than hyperparameter tuning. Determining which features that are the most important for gene prediction across all genomes turned out to be a more complex task than first assumed. Certain features were regarded more critical, depending on what genome data was used as input to the Random Forest classifier. An alternative approach to the forward feature selection used in this thesis could be a backward approach. It would involve gradually discarding the least important features and then retraining the model to see how this affects the remaining features and the model performance.

In Figure 5.6 we see that the feature ”length” was consistently regarded as one of the most critical features, and the same went for a majority of the other genomes. This is not surprising, considering most of the longest ORFs found in the genomes are CDS, and most of the shorter ORFs are n-LORFs. In Figure 5.10 we also saw examples of the Random Forest model performing equally good for certain genomes without the ”length” feature as it did including it. It makes one wonder if the model without the ”length” feature becomes more robust by being forced to employ other distinguishing characters and not just the length feature. Perhaps it can make more accurate classifications when given sequences of similar length, or shorter. Unfortunately, this was not explored in this thesis but could be a target for further investigation.

6.2 XGBoost

The results from Figure 5.12 shows how the default XGBoost classifier clearly outperforms a default Random Forest classifier. However, the default XGBoost classifier and the best performing Random Forest model, perform quite similarly as seen in 5.15. Implementing a Random Forest classifier for the purpose of gene prediction turned out to be quite time consuming. The process of feature engineering was the most time consuming part, followed by tuning and feature selection. Implementing an XGBoost classifier, on the other hand, was slightly less time-consuming as tuning and feature selection did not yield better results than the default model. That being said, the XGBoost also required feature engineering beforehand, which was the most time consuming step in the implementation. Since the XGBoost classifier yields the best performance when using all features, its training time is longer than the best performing Random Forest classifier, which only uses the 200 most important features to yield its highest scores.

To summarise and answer the objective regarding XGBoost presented in Chapter 1; the XGBoost classifier performs on par with the established Random Forest classifier, and is a straightforward model to implement.

6.3 Recurrent Neural Network

The main objective of implementing an RNN model was to determine if it is applicable for gene prediction and how the model compares to the ensemble methods. Even though the performance results for the RNN model were both better and worse than the ensemble methods, one can definitely establish that the RNN model is applicable for the purpose of gene prediction. In the following subsections, the results from the RNN model will be discussed, in the same order as they were presented in Chapter 5.

6.3.1 Maximum Sequence Lengths

The difference between using nucleotide sequences and amino acid sequences as input to the RNN model came across quite clearly in Tables 5.1 and 5.2. By translating the nucleotide sequences one considerably reduced the training time of the model, and a lot of the irrelevant variations in the sequences were removed, which resulted in better accuracy scores as well.

While simultaneously comparing the difference between using nucleotide sequences or amino acid sequences as input to the RNN model, different maximum lengths of the sequences were also tested. Tables 5.3 and 5.4 show how the accuracy decreased when longer maximum lengths were used for both sequence types. When the maximum length is large, the shorter sequences become more padded. The larger the share of 0's due to padding becomes, the lesser the share of relevant sequence information becomes. This makes it more difficult for the model to extract the information needed from the sequence to make the correct classifications. If a sequence is longer than the given maximum length they have to be truncated. As with padding, this can also be done from either the start or the end of the sequence. In this case, the sequences that were longer than the given maximum length were truncated from the beginning of the sequence. The beginning of the sequences often contain more irrelevant information than the end of the sequences. The coding part of the sequence does not need to start from the beginning, it can start from anywhere in the LORF. Therefore, when one truncates the start of the sequences, which one needs to do when one sets the maximum length to a small value, the probability of removing irrelevant information is often high. The remaining part of the sequence then contains a higher percentage of the coding region, which explains why the shorter maximum lengths yield higher performance scores. Since there is such a large span in the lengths of sequences for most genomes, the percentage of padding in the shorter amino acid sequences becomes extremely large. A possible way to avoid this excessive padding could be to group the sequences based on their length and then train separate RNN models for each length category. This is a similar approach to what Al-Ajlan and El Allali [3] did with the CNN-MGP tool, but instead of grouping in terms of GC-content one groups in terms of sequence length.

6.3.2 Comparing Datasets

In Figure 5.14 we see a drop in the accuracy score for certain genomes when dataset 1 was used as input as opposed to dataset 2. This drop was not seen when testing the different datasets with the Random Forest classifier. However, the drop in accuracy score that we see in Figure 5.14, is visible in Figure 5.1 for the precision and recall scores. Specific genomes used as input in both the RNN model and the Random Forest model result in different performance metrics, depending on whether dataset 1 or dataset 2 is used. Clearly, the accuracy score for the RNN model is also affected by the imbalance in dataset 1, as opposed to the Random Forest model where only the precision and recall scores were affected.

6.4 Comparing Machine Learning Methods

From Figure 5.15 we see that the ensemble methods are more similar in performance compared to the RNN model. However, all three methods follow the same general pattern. They perform well for the same genomes and worse for the same genomes. The RNN model, however, performs slightly better with the genomes that the ensemble models find difficult to classify and slightly worse with the genomes that the ensemble models score well with. It means there is a bit less of a difference between the performance scores for the different genomes when using the RNN model as opposed to the ensemble models. The fact that the RNN model fluctuates somewhat less in its performance scores is a promising results, as one ultimately wishes to have a model that predicts good scores consistently, independent of the genome. There are numerous ways one could continue tuning the RNN model to potentially get a model that is better performing overall. One could for instance try different loss functions and optimisation functions, tune the model to prioritise either true positives, true negatives, false negatives, or false positives, reuse samples that are difficult to classify, etc. Another approach could be to create an ensemble model utilising the strengths of all the three models tested. This would then involve using all models for each classification, and then using a majority voting method to select the final result based on what the majority of the models achieved.

In the confusion matrix in Figure 5.16 all three models perform relatively well with this genome, but the ensemble methods get a slightly better result than the RNN model. The largest difference between the confusion matrices for the ensemble methods and the RNN model is that the number of false positive classifications is much higher for the RNN model. In the confusion matrices in Figure 5.17 it is the ensemble methods that have the lower true positive and true negative values and slightly higher false positive and false negative values. It is difficult to say which error is worse between false positives and false negatives in gene prediction. It depends on the problem at hand. In gene mapping it is not problematic to have many positive classifications. When one wishes to compare sequences across genomes, on the other hand, it is a problem if one gets large amounts of false positives. Ideally one would like to have a model that can be tuned to be sensitive to either false positives or false negatives, given the situation.

Most likely many of the false positive classifications made by gene prediction tools are not false, but are genes that have not yet been annotated. Examples of these are *Pseudo-genes*. These are genes that greatly resemble CDS structurally, but are in fact not coding genes. Often they have been coding genes at one point but then have lost their coding ability. These ”genes” are difficult to define, as one cannot be sure whether it is a pseudo-gene or not. Unfortunately, no ground truth is available to tell which genes are correctly annotated or not, which makes it difficult for new classifications and research not to be affected by historical bias. With historical bias, one means the bias that follows already annotated genes and the fact that these genes are used as truth when classifying new sequences. All new classifications are therefore judged and affected by these historically annotated genes, which in reality one cannot be certain if have been annotated correctly or not.

When it comes to training time, the ensemble methods were quicker than the RNN model. The gradient computations that occur during back-propagation in the RNN model are computationally heavy and to speed up the training process parallel processing from a GPU was used. This makes the RNN model both slower in terms of training time and more in need of extra computational power than the ensemble methods. However, one of the great advantages with RNNs is that no feature engineering is required. This is one the most time-consuming steps in implementing ensemble methods. Selecting which features to use, and finding the most accurate way to extract them, was a challenging tasks. The time and resources saved by not needing to extract ”smart” features manually definitely exceeds the extra training time of the model.

6.4.1 Combined genomes

As one can see in several of the plots presented in Chapter 5, the accuracy performance varies greatly depending on which genome is classified. All models struggle with the same genomes and perform well on the same genomes. They all have scored reasonably well on many of the genomes. However, this is of no advantage if all new genome data requires creation of a new model. In addition, most sequence data nowadays is metagenome data. This means there is a growing need for gene prediction tools that can generalise well when given new genomes or a mix of genomes. Figure 4.5 compares four different combinations of the genomes, and the ensemble methods seem to perform the best on all combinations. The most promising result with the combined genomes was that *S. enterica* got a better individual classification score when trained together with *M. tuberculosis*, as presented in table 5.5. As mentioned earlier, *M. tuberculosis* is straightforward to classify as the Random Forest classifier gets a good performance score for this genome. *S. enterica*, on the other hand, is trickier as the Random Forest classifier gets a worse performance score for its genome. These results indicate that by combining "tricky" genomes with "straightforward" genomes and training them together, one can potentially improve the score for "tricky" genomes.

6.5 Other Future Work

Trying RNN for the purpose of gene prediction was a natural next step, after discovering CNNs have been tried and proven to be sufficient [3]. In addition, RNNs are known to be particularly good with handling sequential data, and have so far shown promising results in the work done in this thesis. However, there is still a lot more left to explore with RNNs. The architecture used for the RNN model in this thesis was a very simple one, and the parameters chosen have a large impact on the results of the model. This means a lot more tuning could be done to improve the model, including tuning the number of LSTM layers. It is said that the width of the neural network, which refers to the number of nodes in each layer, extracts more features, while the depth, which refers to the number of layers in the neural network, extracts richer features. There are definitely many potential features to extract in DNA sequences, and one should therefore ideally try many different combinations of widths for the various layers in the model.

There are also several other types of recurrent layers and deep learning methods that could be tested, such as *Gated Recurrent Units (GRU)*, bidirectional LSTM, and transformer models. GRUs are somewhat similar to LSTM units. Both use gating mechanisms to memorise information and by this, minimising the problem of vanishing gradients. GRUs however, only have two gates, the reset gate and the update gate, which makes them less complex than the LSTM units. The fact that LSTMs are more sophisticated due to their extra gates, can be both an advantage and a disadvantage. It makes them more capable of remembering long term dependencies in long sequences, but it also makes them more complex with many more parameters to tune. Therefore, GRUs are known to be faster and simpler to modify than LSTMs. There is no rule or guarantee that one will work better than the other; one simply has to try both and compare their performances. Bidirectional LSTMs involve two LSTM models being trained, one learns from one direction of the sequences, and the other learns from the reverse direction of the sequence. The output from the two LSTM models then needs to be merged, and this can be done in several ways such as summation, multiplication, averaging or concatenation. This makes bidirectional LSTMs particularly good for NLP problems, as they are known to be more successful in capturing the context of the input sequences. The disadvantage to this is that they need much more training time and are therefore slower than normal LSTMs. Finally, a transformer model could also be explored for gene prediction. A transformer model has the ability of self-attention, which means it learns by enhancing or diminishing certain parts of a sequence depending on what it regards as more or less important. This is done to determine the context of each part of the input data. It does not need to process or memorise input sequence chronologically but rather learns by parallel computing, which makes it a faster option than other RNNs.

Only hard classifications were used for the models in this thesis. Instead of using only hard classifications one could use predicted probabilities to plot the outputs. Such plots would show the distribution of the classifications made by the models, i.e., indicate the fractions of the classifications that the models are very certain of, and respectively less certain of. These probabilities are particularly useful in cases where certainty about the classifications is essential, or if it is desirable to identify as many CDS as possible. One simply adjusts the probability cutoff. Another use case for probability outputs are, if there are many overlapping ORFs, of which some have been classified as CDS and some not, one can assume that the ORF with the highest probability is the actual CDS.

Due to time limitations, feature importance across genomes was not explored further in this thesis. However, it would be interesting to get a better idea of which features, other than "length", are important for the majority of genomes. In general, more data exploration of different genomes could allow one to discover characteristics that explain why some genomes are more difficult to classify than others. One could for instance use these characteristics to create separate gene prediction models that are trained to tackle these differences, such as length distribution, different GC-content, codon biases, etc. As stated by Al-Ajlan and El Allali [3] "previous research has shown that building multiple models based on GC-content is better than building a single model [50], because fragments with similar GC-content have closer features such as codon usage [50]." The CNN-MGP has 10 different versions, each for a specific GC-content group. This, among other things, is something that could be tested for all three of the models in this thesis. In a metagenomic setting, it would be useful to know if there are certain characteristics that one should filter the metagenomic data by that determines what custom-made model one should use for each of the different subsets of data.

One of the incentives for implementing a Random Forest classifier, despite the fact that it has already been tested by others successfully, was to implement different features. The features introduced in this thesis were inspired by the features used by Silva et al. [2]. Some were engineered differently (codon usage) and some additional features were added such as "aa-mers" and "start codon usage". The addition of both these feature sets seem to have been a good idea, considering both got relatively good feature importance scores as seen in Figure 5.6. The RNN model shows the benefits of using the amino acid sequences as opposed to the nucleotide sequences, which perhaps also could be related to the use of k-mers and aa-mers. These feature sets essentially both contain the same information, and it may be redundant to use both feature sets in the ensemble methods. It would be interesting to see how the models perform when given only either the aa-mer or k-mer feature sets.

Even though a couple new feature sets were introduced during the implementation of the ensemble models in this thesis, there are still plenty more potential features that could be tested, such as upstream activating sequences for instance. These sequences play a part in the expression of protein coding genes and are found upstream of the start codon of an ORF. Other potential features could be the relative position of the ORF in a genome, given that the whole genome has been sequenced and that one has long consecutive sequences. Or, whether or not coding genes have other neighbouring ORFs that also are coding genes. Finally, one could consider whether the functionality of the coding genes, given by the amino acid sequences, can indicate how one more readily can predict these coding genes in a sequence.

Chapter 7

Conclusion

The main objective of this thesis has been to evaluate the performance of three machine learning models to classify CDS and n-LORFs in prokaryotic DNA sequences. The models tested are a Random Forest classifier, an XGBoost classifier, and an RNN model. The performances of the three models have been compared to each other when trained, tuned, and tested on the same datasets, consisting of individual genomes, as well as combinations of genomes. Several factors have motivated the choice of the sub-objectives presented in Section 1.2. Firstly, the Random Forest classifier is an established method in the field of gene prediction, and the article written by Silva et al. [2] showed promising results when using this method. The implementation of the Random Forest classifier in this thesis has therefore attempted to follow a similar approach while introducing new features and trying different feature engineering and selection techniques. The XGBoost classifier is a popular ensemble method, as well as a known competitor of the Random Forest classifier, and it has in this thesis proven to perform on par with the established Random Forest classifier. Since Al-Ajlan and El Allali [3] proved to be successful with a CNN model for gene prediction, and RNNs are known to be good at handling sequential data, the method seemed like an appropriate next candidate. Even though a simple RNN model was implemented and minimal tuning was done, the model showed satisfactory results.

The three models were trained and tested on 15 prokaryotic genomes, of which all LORFs were extracted and prepared to be used as input to the models. One of the challenges with preparing these datasets is that the length distribution of CDS and n-LORFs is very different. A majority of all long sequences are CDS, and in return a majority of all short sequences are n-LORFs. In addition, there are fewer CDS than n-LORFs. This was solved by creating two different datasets, one focused on having an as balanced length distribution as possible and the other focused on having a balanced number of CDS and LORFs. The three models were tested on both datasets, and they all performed better when trained on the dataset with a balanced number of CDS and LORFs.

Another aspect of the data preparation that was time consuming when implementing the ensemble methods was feature engineering. One of the great advantages with RNNs is that feature engineering is avoided entirely. The sequence data can be used as input directly without manual feature extraction and selection.

The overall consensus from the results is that all three models are applicable for gene prediction in prokaryotic DNA. Considering the data used to train these three models was significantly less than the data used for GeneRFinder and CNN-MGP, their performance is that much more impressive. In addition, all three models have an abundance of parameters and tuning potential that has not yet been tested, and the experiences made in this thesis can serve as a building block for further exploration of the models. The fact that the RNN model yielded as good results as it did, despite limited tuning, is perhaps the most promising result achieved. This creates a magnitude of opportunities not only for gene prediction but also for other fields within bioinformatics.

Bibliography

[1] *Genomics.* Mar. 2022. URL: https://en.wikipedia.org/wiki/Genomics#:~:text=Genomics.

[2] Raíssa Silva et al. "geneRFinder: gene finding in distinct metagenomic data complexities". In: *BMC Bioinformatics* 22.87 (2020). ISSN: 1471-2105. URL: https://doi.org/10.1186/s12859-021-03997-w.

[3] Amani Al-Ajlan and Achraf El Allali. "CNN-MGP: Convolutional Neural Networks for Metagenomics Gene Prediction". In: *Interdisciplinary Sciences: Computational Life Sciences* 11.4 (2019), pp. 628–635. URL: https://doi.org/10.1007/s12539-018-0313-4.

[4] Tulio L. Campos et al. "An Evaluation of Machine Learning Approaches for the Prediction of Essential Genes in Eukaryotes Using Protein Sequence-Derived Features". In: *Computational and Structural Biotechnology Journal* 17 (2019), pp. 785–796. ISSN: 2001-0370. URL: https://doi.org/10.1016/j.csbj.2019.05.008.

[5] Amani Al-Ajlan and Achraf El Allali. "The Effect of Machine Learning Algorithms on Metagenomics Gene Prediction". In: (Mar. 2019). DOI: 10.1145/3309129.3309136.

[6] Le Duc Hau, Nguyen Hoai, and Yung-Keun Kwon. "A Comparative Study of Classification-Based Machine Learning Methods for Novel Disease Gene Prediction". In: *Advances in Intelligent Systems and Computing* 326 (Jan. 2015), pp. 577–588. DOI: 10.1007/978-3-319-11680-8_46.

[7] National Human Genome Research Institute NIH. *Double helix.* URL: https://www.genome.gov/genetics-glossary/Double-Helix.

[8] Frida Belinky, Igor B. Rogozin, and Eugene V. Koonin. "Selection on start codons in prokaryotes and potential compensatory nucleotide substitutions". In: *Nature News* (Sept. 2017). URL: https://www.nature.com/articles/s41598-017-12619-6.

[9] *AI vs. Machine Learning vs. Deep Learning vs. neural networks: What's the difference?* URL: https://www.ibm.com/cloud/blog/ai-vs-machine-learning-vs-deep-learning-vs-neural-networks.

[10] IBM Cloud Education. *What is deep learning?* URL: https://www.ibm.com/cloud/learn/deep-learning.

[11] *Scikit-Learn.* URL: https://scikit-learn.org/stable/.

[12] Fatih Karabiber. *Gini impurity.* URL: https://www.learndatasci.com/glossary/gini-impurity/.

[13] Tony Yiu. *Understanding random forest.* Sept. 2021. URL: https://towardsdatascience.com/understanding-random-forest-58381e0602d2#:~:text=The.

[14] *Random Forest.* Mar. 2022. URL: https://en.wikipedia.org/wiki/Random_forest.

[15] IBM Cloud Education. *What is Random Forest?* URL: https://www.ibm.com/cloud/learn/random-forest?mhsrc=ibmsearch_a&amp;mhq=random+forest.

[16] Hieu Tran. "Survey of Machine Learning and Data Mining Techniques used in Multimedia System". In: *BioData Mining* 11 (Sept. 2019). URL: https://doi.org/10.13140/RG.2.2.20395.49446/1.

[17] Vihar Kurama. *Gradient boosting for classification.* Apr. 2021. URL: https://blog.paperspace.com/gradient-boosting-for-classification/.

[18] Will Koehrsen. *Hyperparameter tuning the random forest in python.* Jan. 2018. URL: https://towardsdatascience.com/hyperparameter-tuning-the-random-forest-in-python-using-scikit-learn-28d2aa77dd74.

[19] *Feature selection.* Mar. 2022. URL: https://en.wikipedia.org/wiki/Feature_selection.

[20] Jason Brownlee. *How to choose an activation function for deep learning.* Jan. 2021. URL: https://machinelearningmastery.com/choose-an-activation-function-for-deep-learning/.

[21] Seb. *Understanding backpropagation with gradient descent.* Oct. 2021. URL: https://programmathically.com/understanding-backpropagation-with-gradient-descent/.

[22] Sebastian Raschka and Vahid Mirjalili. *Python Machine Learning, 3rd Ed.* 3rd ed. Birmingham, UK: Packt Publishing, 2019. ISBN: 978-1789955750.

[23] Rauf Bhat. *Gradient descent with momentum.* Oct. 2020. URL: https://towardsdatascience.com/gradient-descent-with-momentum-59420f626c8f.

[24] Kuldeep Chowdhury. *10 hyperparameters to keep an eye on for your LSTM model and other tips.* May 2021. URL: https://medium.com/geekculture/10-hyperparameters-to-keep-an-eye-on-for-your-lstm-model-and-other-tips-f0ff5b63fcd4.

[25] Alex Graves, Abdel-rahman Mohamed, and Geoffrey Hinton. "Speech recognition with deep recurrent neural networks." In: *Speech and Signal Processing (ICASSP)* (Jan. 2013), pp. 6645–6649. URL: https://en.wikipedia.org/wiki/Long_short-term_memory#/media/File:Peephole_Long_Short-Term_Memory.svg.

[26] *Long short-term memory.* Apr. 2022. URL: https://en.wikipedia.org/wiki/Long_short-term_memory.

[27] Doug Hyatt et al. "Prodigal: prokaryotic gene recognition and translation initiation site identification." In: *BMC Bioinformatics* 11.9 (2010). URL: https://doi.org/10.1186/1471-2105-11-119.

[28] Rho M;Tang H;Ye Y; "FragGeneScan: Predicting genes in short and error-prone reads". In: *Nucleic acids research* (). URL: https://pubmed.ncbi.nlm.nih.gov/20805240/.

[29] Katharina J Hoff et al. "Orphelia: Predicting genes in metagenomic sequencing reads". In: *Nucleic acids research* (July 2009). URL: https://www.ncbi.nlm.nih.gov/pmc/articles/PMC2703946/.

[30] *Professional software for data science teams: R Studio.* Apr. 2022. URL: https://www.rstudio.com/.

[31] *Biological Sequences Retrieval and analysis.* June 2021. URL: https://cran.r-project.org/web/packages/seqinr/index.html.

[32] Kristian Liland and Lars Snipen. *Package microseq.* URL: https://cran.r-project.org/web/packages/microseq/index.html.

[33] Kristian Liland and Lars Snipen. *Package microclass.* URL: https://cran.r-project.org/web/packages/microclass/index.html.

[34] *Reference genome*. Jan. 2022. URL: https://en.wikipedia.org/wiki/Reference_genome#:~:text=A.

[35] Amani Al-Ajlan and Achraf El Allali. “Feature selection for gene prediction in metagenomic fragments.” In: *BioData Mining* 11.9 (2018). URL: https://doi.org/10.1186/s13040-018-0170-z.

[36] Achraf El Allali and John R Rose. *MGC: A metagenomic gene caller - BMC bioinformatics*. June 2013. URL: https://bmcbioinformatics.biomedcentral.com/articles/10.1186/1471-2105-14-S9-S6#citeas.

[37] Kajetan Bentele et al. “Efficient translation initiation dictates codon usage at gene start.” In: *Molecular systems biology* 9.675 (2013). URL: https://www.ncbi.nlm.nih.gov/pmc/articles/PMC3964316/.

[38] J L Oliver and A Marín. “A relationship between GC content and coding-sequence length.” In: *Journal of molecular evolution* 43.3 (1996), pp. 216–223. URL: https://pubmed.ncbi.nlm.nih.gov/8703087/.

[39] Uberto Pozzoli et al. “Both selective and neutral processes drive GC content evolution in the human genome.” In: *BMC evolutionary biology* 8.99 (2008). URL: https://pubmed.ncbi.nlm.nih.gov/18371205/.

[40] *Sklearn.ensemble.randomforestclassifier*. URL: https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestClassifier.html.

[41] Akash Dubey. *Feature selection using Random Forest*. Dec. 2018. URL: https://towardsdatascience.com/feature-selection-using-random-forest-26d7b747597f.

[42] Piotr Płoński. *Random Forest feature importance computed in 3 ways with python*. June 2020. URL: https://mljar.com/blog/feature-importance-in-random-forest/#:~:text=Random.

[43] Keras Team. *Keras*. URL: https://keras.io/.

[44] *Machine learning education: tensorflow*. URL: https://www.tensorflow.org/resources/learn-ml.

[45] Sawan Saxena. *Understanding embedding layer in Keras*. Feb. 2021. URL: https://medium.com/analytics-vidhya/understanding-embedding-layer-in-keras-bbe3ff1327ce.

[46] Cory Maklin. *Dropout neural network layer in Keras explained*. June 2019. URL: https://towardsdatascience.com/machine-learning-part-20-dropout-keras-layers-explained-8c9f6dc4c9ab.

[47] Yva Sandvik. *Data Science Thesis*. 2022. URL: https://github.com/yvasandvik/data-science-thesis.

[48] Jason Brownlee. *How to choose loss functions when Training Deep Learning Neural Networks*. Aug. 2020. URL: https://machinelearningmastery.com/how-to-choose-loss-functions-when-training-deep-learning-neural-networks/.

[49] *Open reading frame*. URL: https://www.genome.gov/genetics-glossary/Open-Reading-Frame#:~:text=So.

[50] Achraf El Allali and John R Rose. “MGC: A metagenomic gene caller - BMC bioinformatics”. In: *BioMed Central* (June 2013). URL: https://bmcbioinformatics.biomedcentral.com/articles/10.1186/1471-2105-14-S9-S6.

Appendices

Appendix A: R-packages and Python Libraries Used

R-package name	Version	Purpose of use
seqinr	4.2.8	Mainly for the "G+C Content" function
microclass	1.2	Mainly for the "KmerCount" function
ggplot2	3.3.5	Plotting graphs
dplyr	1.0.7	Data frame manipulations
microseq	2.1.5	Fetch and process data, find ORFs, and for feature engineering

Table A.1: R-packages used during data exploration, data processing, feature selection and feature engineering, the respective version used as well as the purpose of use.

Python libraries	Version	Purpose of use
pandas	1.3.5	Pre-processing of data
numpy	1.20.3	Array manipulations
matplotlib	3.5.1	Plotting
seaborn	0.11.2	Plotting
sklearn	0.23.2	Implementation of the Random Forest and XGBoost models
tensorflow	2.6.2	Implementation of the RNN model
keras	2.6.0	Implementation of the RNN model
keras_tuner	1.1.0	Tuning of the RNN model

Table A.2: Python libraries used during the implementation of the Random Forest, XGBoost, and RNN models, as well as the version used, and a short description of the purpose of use.

Appendix B: Source Code Links for GitHub

File name	Link	Commit hash
script_data_exploration.Rmd	GitHub:data_expo	d1b0df7
script_creating_features.Rmd	GitHub:creating_features	de97615
rf_model_master.ipynb	GitHub:random_forest	382bbb1
xgb_model_master.ipynb	GitHub:xgboost	112fccf
rnn_model_master.ipynb	GitHub:rnn	0687a2a

Table B.1: Links and commit hashes for some of the source code created during data exploration, feature engineering and model implementation. All source code created during the work on this thesis can be found on GitHub[47].

Appendix C: Preliminary Work - Data Science Seminar

Methods used for Prediction of Protein Coding Genes in DNA Sequences: A Literature Review

Written by
Yva Sandvik

Supervised by
Kristian Hovde Liland

Norwegian University of Life Sciences

Master of Science
Faculty of Science and Technology
December 5, 2021

Abstract

Gene prediction plays a crucial role in understanding the genome of a species once it has been sequenced. It has a broad application basis and it is regarded as one of the most promising topics within bioinformatics. Several gene prediction tools have been developed and improved over the years. The computational methods applied by these tools vary, as well as their performance. This literature review aims to compare state-of-the-art gene prediction tools with an emphasis on recent solutions, outline the existing challenges, and present possible future efforts.

First, background knowledge required to understand the application and evaluation of gene prediction tools is introduced. Next, current gene prediction tools that apply different computational methods such as Hidden Markov Models, Dynamic Programming and Machine Learning, are summarized and compared.

Despite the many efforts that have improved the performance of gene prediction tools in recent years, one can see that there still remains many important topics for future research.

1 Introduction

Following the technological advances that have been made over the past decade the amount of DNA sequencing data that is produced has increased rapidly. In accordance with this increase, the demand for analysis, interpretation and learning from said data has never been greater. The work that goes into tackling this task falls within the field of bioinformatics.

Bioinformatics can broadly be defined as an intersection between the disciplines biology, computer science, statistics and medical science. It is essential for the management and analysis of biological data, and it makes use of computational tools and mathematical models to give insight into fields such as genetics, pharmacology and microbiology. For instance, through the prediction and annotation of undiscovered gene products, the field of bioinformatics has contributed to the functional understanding of the human genome, enhanced the discovery of drug targets and made advances in directed therapy [1]. These are just a few examples of the many benefits from getting increased insight into genomic data.

Gene prediction, also called gene finding, is the process of identifying an area of genomic DNA that encodes a gene. This may include both protein coding genes, RNA genes as well as regulatory regions [2]. In this paper the focus will be on explaining and comparing existing computational methods used to predict primarily protein coding genes in DNA sequences.

1.1 Motivation

Today's gene prediction methods have high error rates, and there has not been much development in this area of research. This paper aims to get acquainted with the current methods used in gene prediction, and investigate machine learning models used for the purpose of predicting protein coding genes. The motivation for this is the possibility of improving the current gene prediction techniques, and finding new tools that can attain lower error rates.

1.2 Objective

The literature search yielded plenty of papers giving comprehensive explanations about specific gene prediction methods. However, in recent years there has not been written any literature reviews that take on state-of-the-art methods as well as up and coming machine learning methods. Therefore, this review aims to compare the current most cutting-edge methods for gene prediction, and present the findings along with the current challenges. This leaves us with three objectives which are presented below in the form of questions.

Objectives

1. What tools are currently being used for prediction of protein coding genes in prokaryotes?
2. Are machine learning methods applicable for predicting protein coding genes in prokaryotic DNA?
3. What are the current challenges with the prediction of protein coding genes?

1.3 Structure of Review

The rest of the review is organised as follows:

- In Section 2, the method used to find literature and the screening criteria for articles are presented.
- In Section 3, essential terms and theory required to understand the process of gene prediction are explained.
- In Section 4, the objectives presented above are addressed by categorizing and explaining the existing gene prediction methods.
- In Section 5, challenges with existing gene prediction methods, and areas for further exploration, are presented.
- In Section 6, the review is concluded.

2 Method

The search engines Oria and Google Scholar were utilized to find relevant literature on the subjects of interest, using the keyword searches found in Table 1. The searches were limited to peer reviewed papers, with no limitations on publication dates. This yielded a total of 18 articles of which 6 were selected after screening.

Nr.	Title terms	General terms
1	gene $\times$ prediction* $\times$ review	None
2	gene $\times$ prediction* $\times$ (method* $\wedge$ technique*)	machine $\times$ learning
3	gene $\times$ prediction* $\times$ (method* $\wedge$ technique*) $\times$ review	None

Table 1: Search terms used in the Oria searches. The columns contain terms required to be in the title, or generally in the article. The "$\times$" represent the AND operator between two terms, and "$\wedge$" represent the OR operator. The "*" means that the search will include any word beginning with the word before the star. The table is based on Table 2.1 in Sandvik [3].

A number of broad searches were also made on the websites Nature, NCBI, PubMed and BMC Bioinformatics with terms such as machine learning, deep learning, gene finding, gene prediction, Prodigal, GeneMarK, etc. These searches yielded another 22 relevant articles of which 10 were selected after screening. The limitations and screening done to select the final 16 articles used for this review is explained in Subsection 2.1 below.

2.1 Limitations and Screening

To limit the scope of this review only the papers using intrinsic approaches to gene prediction were selected. In addition, as protein-coding gene prediction in prokaryotic genomes is considered a much simpler task than for intron-containing eukaryotic genomes, the papers regarding prokaryotic gene prediction tools were prioritized. Studies based on gene function prediction or annotation were discarded.

3 Theory

This section will briefly explain what is meant by a genome, DNA, DNA sequencing, genes, and other structures in the genetic code. These terms are necessary to understand the context and applications of gene prediction.

3.1 The Genetic Code

Deoxyribonucleic acid (DNA) is the genetic material used to store an organism's complete set of genetic information. This material is also responsible for the translation of the information encoded within the genetic code and into proteins. *The genetic code* is a term used to explain the sequence of nucleotide bases in a genome. The four different DNA nucleotides are adenine (A), thymine (T), guanine (G) and cytosine(C), and they are regarded as the building blocks of DNA. In Figure 1 we present a nucleotide which consists of a sugar molecule, phosphate group, and a distinguishing nitrogen-containing base.

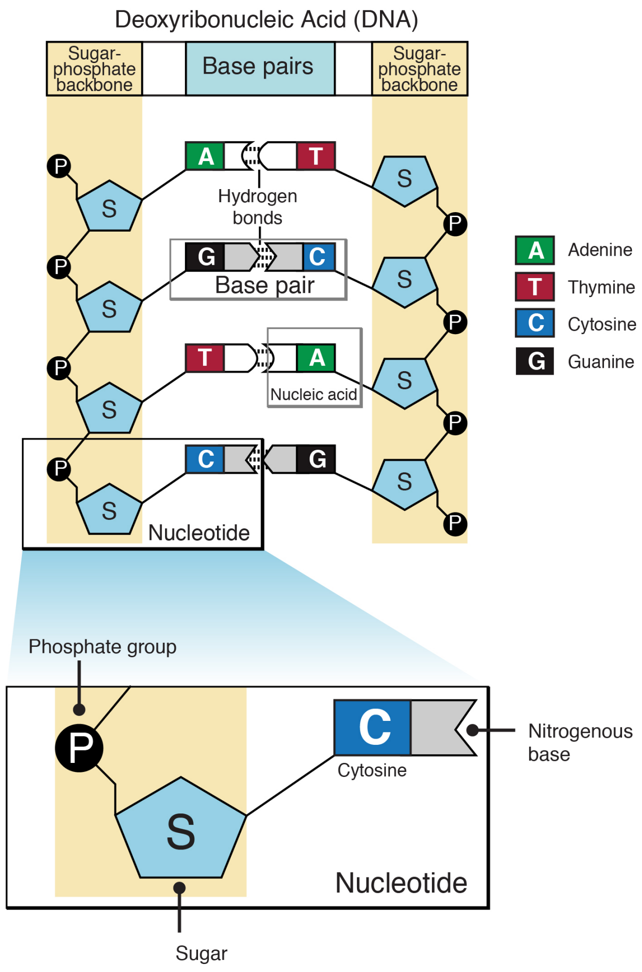


Figure 1: Illustration of a nucleotide and the structure of a DNA double helix, taken from National Human Genome Research Institute [4] (public domain). The Figure shows how each strand is built up of covalently linked nucleotides, through the phosphate and sugar part of the nucleotide. This makes up the sugar-phosphate backbone. The nitrogenous bases form complementary base pairs with each other.

A DNA molecule consists of two complementary strands, and is often referred to as *double stranded DNA*, or *DNA double helix.* Each strand is built up of covalently linked nucleotides through the phosphate and sugar part of the nucleotide, which make up the sugar-phosphate backbone, or one DNA strand. Covalent bonds are ordinary chemical bonds that involve the sharing of electron pairs between atoms. To make a complete DNA molecule one needs two DNA strands. The nitrogenous bases can form complementary base pairs with each other. Adenine correctly pairs with thymine (A-T), and guanine correctly pairs with cytosine (G-C). Hence, the DNA strands in a DNA double helix mirror each other and are called *complementary strands* as one can predict one strand from the other.

3.1.1 Reading Frames

There are three ways of reading a strand of DNA depending on which position of the strand one starts reading from [5]. Three consecutive nucleotides make up a *codon* in a DNA sequence. Each codon codes for a specific amino acid. In Section 3.2 we will discuss this further. Lets say one has the sequence of bases: GCTACGGGG. The reading could start from the first base giving a total of three codons: GCT, ACG and GGG. The reading could start from the second base giving two codons: CTA and CGG. Lastly the reading could start from the third base also giving two bases: TAC and GGG. Looking at the resulting codons we see that the codons found can differ entirely or only partially given the reading frame. Double-stranded DNA will also have a reverse complimentary strand which will have additional three reading frames. Therefore, there are in total six possible reading frames in double-stranded DNA.

3.2 Genes and Protein Synthesis

Protein synthesis involves two processes where the genetic code is directly involved: transcription and translation. In transcription the sequence of base pairs, i.e. genetic information in a DNA strand, is transcribed to a ”transport” sequence called *ribonucleic acid (RNA)* or *messenger RNA* (mRNA), in the case of coding genes. RNA has a very similar structure as DNA. The purpose of mRNA is to act as a messenger, carrying instructions from DNA to control the synthesis of proteins. In some viruses RNA rather than DNA carries the genetic information [6].

The next step is then translation, where the amino acids are selected based on a sequence of three nucleotides which together make up a codon. Translation is facilitated by a ribosome which moves along a transcribed mRNA strand until it finds a start codon. Each codon, other than the stop codons, code for a particular amino acids. As the ribosome moves along the mRNA, a polypeptide chain is formed by amino acids being linked together in the order specified by the codons and the respective *transporter RNA (tRNA)* which carries the correct amino acid to the binding site [7]. The process of translation is illustrated in Figure 2. When a sequence of nucelotides in DNA is both transcribed and translated, it means that this sequence encodes the synthesis of a gene product and is regarded as a gene. The whole process of DNA being transcribed to mRNA and then translated to protein is known as *the central dogma of molecular biology.*

3.3 Coding and Non-coding DNA

As mentioned in Section 1 a gene product can be either RNA or protein. One can divide DNA sequences into the non-coding and coding DNA. When referring to coding DNA sequences one means a sequence of DNA that codes for proteins. Non-coding DNA on the other hand include the genes that code for RNA. Some non-coding DNA can play a part in the regulation of gene expression, and in eukaryotes some non-coding DNA lies between genes in a DNA sequence and

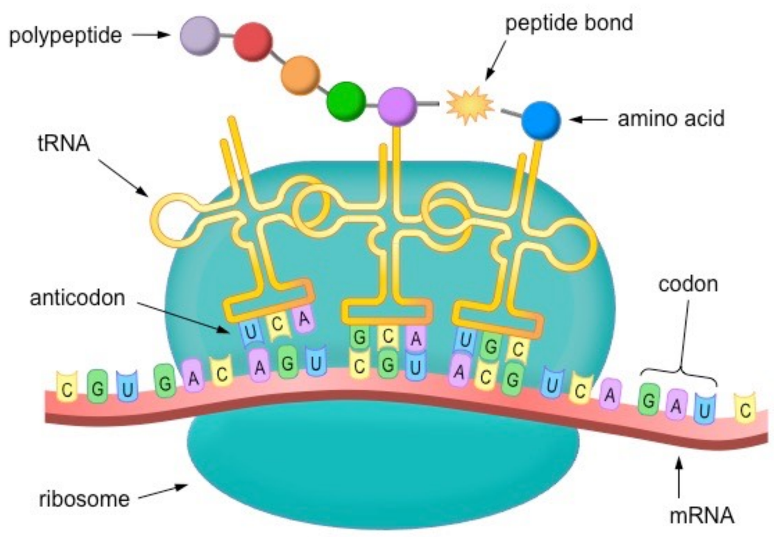


Figure 2: Illustration of protein synthesis, taken from Brent Cornell [8] (public domain). The Figure shows how a ribosome facilitates translation by moving along a transcribed mRNA strand while tRNA correctly matches the peptides with corresponding codons, creating a polypeptide chain.

are called *introns* [9]. Introns are mostly present in only eukaryotic genomes, and need to be spliced out of a mRNA sequence before translation.

3.4 Finding Genes

Before gene prediction and gene annotation can begin one needs to have sequenced genomes to work with. As stated by Snipen [10]: "sequencing refers to the technology that allows us to actually read the sequence of DNA or RNA molecules". This data then needs to go through some processing so that the quality of the sequenced data is verified.

In the process of sequencing, one gets a number of overlapping and often also repeated fragments called *reads*. These reads are then aligned and merged to form longer fragments called *contigs*. This is done to reconstruct the original DNA fragment and the process is referred to as sequence assembly.

Once one has a set of genomic sequences one may wish to find out where exactly the genes are in these assembled sequences also called *gene prediction*. This will be the first step to find out what the function of the identified genes are, which is called *gene annotation* [11]. As mentioned in Section 1, this paper will focus on the intrinsic approach to gene prediction. Below are brief explanations of the two approaches to finding genes.

3.4.1 Extrinsic Approach

The extrinsic approach, also called *similarity based searches*, is a reference based approach where one compares the contigs to a database of already known and annotated genes. RNA genes are quite conserved genes, meaning if one has seen them in one organism one will recognise them in another organism. This makes RNA genes easily identifiable by using the extrinsic approach, since it is likely they are identical to genes found earlier. As stated by Ayal [12]: "Local alignment and global alignment are two methods based on similarity searches. The most common local alignment tool is the BLAST family of programs, which detects sequence similarity to known genes or proteins".

3.4.2 Intrinsic Approach

This approach involves recognizing genes based on some rule or patterns without using external data, instead it uses gene structure as a template to detect genes [12]. These methods are also referred to as *ab-initio prediction methods.* Protein coding genes, which are the majority of the genes in a genome, evolve fast and can be quite different from one organism to another. Therefor the intrinsic approach is a widely used approach when it comes to protein coding genes.

3.4.3 Open Reading Frames

Wikipedia [13] states that: "an open reading frame is the part of a reading frame that has the ability to be translated". Almost all open reading frames start with a start codon and end with a stop codon. There are three possible stop and start codons as one can see in Figure 3. Stop codons have no other meaning, they simply determine the end of a coding gene. Start codons on the other hand also occur inside genes as ordinary codons. Hence, one cannot be certain if a start codon is really the start of an ORF or if it is just inside an ORF, coding for an amino acid. A good way to search for genes is therefore by starting with identifying stop codons, and not necessarily start codons. The three known stop codons are: TGA, TAG and TAA [14]. The three most known start codons in prokaryotes are: ATG, GTG and TTG. Eukaryotes rarely alternate start codons, and mainly uses only the ATG codon as a start codon [15].

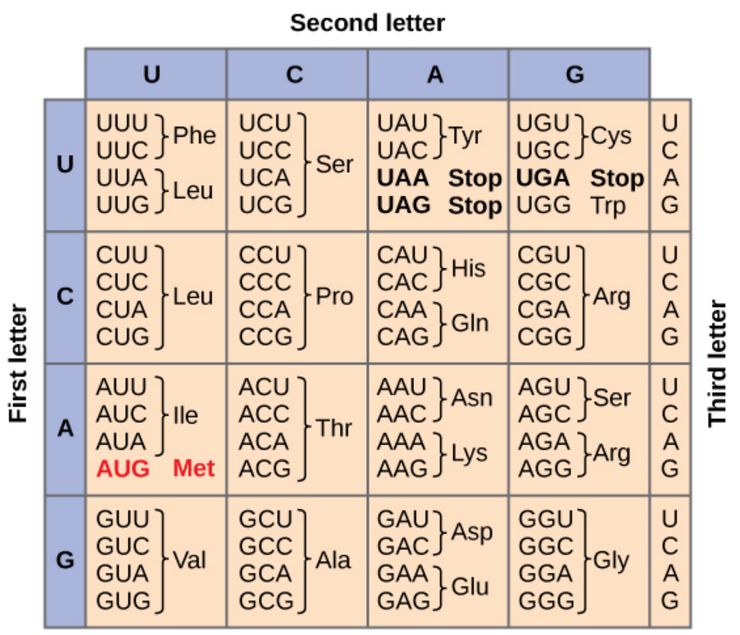
An ORF starting in a given reading frame can contain multiple ORFs as there may be multiple start codons withing this ORF before the next stop codon is identified. In Figure 4 we can see how the start of an ORF is identified by the first possible stop codon in a given reading frame, 'TAA', and continues until the next stop codon is located downstream, 'TGA'. The longest possible ORF (LORF) is defined from the first stop codon and until the start codon that is furthest upstream in the sequence before the next stop codon. In Figure 4 this is 'GTG'. However, we can also see that there are two other start codons located in the middle of the ORF, giving us three possible ORFs in this reading frame.

3.5 Features used in Gene Prediction Tools

All gene prediction tools apply complex filtering on ORFs across a region of DNA or genome in order to locate genes. The filtering performed is unique to each tool and may be dependent on properties (features) such as GC content, ORF length, codon usage, upstream motifs and overlapping genes [17]. How these may serve as useful features in a gene prediction tool is briefly explained below.

3.5.1 Open Reading Frame Length and GC content

Coding genes tend to be longer than random ORFs. Random ORF lengths follow a geometric distribution. This distribution is said to depend on the GC-content of the genome [18]. If we take a look at Figure 3 we can see that the stop codons contain a higher density of T and A, less G and no C. A genome with low GC-content, contains more T and A and is therefore likely to contain more stop codons and in turn shorter ORFs. Low GC-content can in theory therefore be an indication of shorter ORFs. In any biological genome the long ORFs are overepresented, compared to what one statistically would expect, based on the random distribution of lengths. This is due to natural selection during evolution. Long ORFs are more likely to be coding genes and are thereby more necessary to an organism [14]. This is an attempt at explaining how GC-content and ORF length can be used by gene prediction tools as important features.



First letter	Second letter: U	Second letter: C	Second letter: A	Second letter: G	Third letter
U	UUU Phe	UCU Ser	UAU Tyr	UGU Cys	U
U	UUC Phe	UCC Ser	UAC Tyr	UGC Cys	C
U	UUA Leu	UCA Ser	**UAA Stop**	**UGA Stop**	A
U	UUG Leu	UCG Ser	**UAG Stop**	UGG Trp	G
C	CUU Leu	CCU Pro	CAU His	CGU Arg	U
C	CUC Leu	CCC Pro	CAC His	CGC Arg	C
C	CUA Leu	CCA Pro	CAA Gln	CGA Arg	A
C	CUG Leu	CCG Pro	CAG Gln	CGG Arg	G
A	AUU Ile	ACU Thr	AAU Asn	AGU Ser	U
A	AUC Ile	ACC Thr	AAC Asn	AGC Ser	C
A	AUA Ile	ACA Thr	AAA Lys	AGA Arg	A
A	AUG Met	ACG Thr	AAG Lys	AGG Arg	G
G	GUU Val	GCU Ala	GAU Asp	GGU Gly	U
G	GUC Val	GCC Ala	GAC Asp	GGC Gly	C
G	GUA Val	GCA Ala	GAA Glu	GGA Gly	A
G	GUG Val	GCG Ala	GAG Glu	GGG Gly	G

Figure 3: Table showing which codons code for which amino acid in an RNA strand. In RNA, the uracil nucleobase replaces thymine. Each amino acid is coded for by 2-4 different codons. Only the most common start codon is marked in red. In Section 3.4.3 other common start codons are mentioned. The figure is taken from Lumen [16] (public domain).

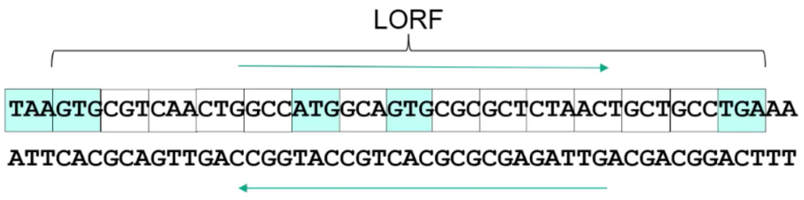

LORF
TAAGTGCGTCAACTGGCCATGGCAGTGCGCGCTCTAACTGCTGCCTGAAA
ATTCACGCAGTTGACCGGTACCGTCACGCGCGAGATTGACGACGGACTTT

Figure 4: Illustrates the definition of the "Longest possible ORF" (LORF) for a given stop-codon. There are three possible start codons between two stop codons in the figure. The LORF is defined as the stretch of sequence starting with the start codon furthest downstream from the first stop codon in both complementary strands. Figure taken from Snipen [14].

3.5.2 Codon Usage

It is to some degree possible to use the codon content of ORFs to separate between the coding and non coding genes. Most amino acids are coded by several codons, as we can see in Figure 3. Some of these codons are typical or nontypical in a certain genome. This varies a lot between genomes, however when identifying genes in a genome it could be useful to make a probabilistic model to capture this information as it can aid with gene prediction.

3.5.3 Upstream Motif

Upstream of a gene there is a *ribosomal binding site (RBS)*. This is often a short motif, usually 5-15 bases upstream of start-codon. This area is expected to be AG-rich, and by identifying AGGAGG or other patterns upstream of start codons, these upstream motifs can serve as valuable features in a model that aims to identify genes.

3.6 Evaluation Metrics

When evaluating the performance of gene prediction methods the metrics accuracy, recall, precision and specificity are often used. These metrics are based on the true positive (TP), false positive (FP), true negative (TN) and false negative (FN) values. True positive are the correctly identified coding genes and true negative are the correctly identified non coding genes. False positives are non-coding genes incorrectly predicted as coding genes, and false negatives are coding genes incorrectly predicted as non-coding genes. The metrics are defined as follows:

- **Accuracy**: presents the proportion of correct predictions considering the total number of instances and is defined by equation 1.
- **Recall**: also called sensitivity, gives the proportion of correctly identified coding genes and is defined by equation 2.
- **Precision**: gives an estimate of how good a method is at excluding non coding genes and is defined by equation 3.
- **Specificity**: gives the proportion of correctly identified non coding genes and is defined by equation 4.

$$Accuracy = \frac{TP + TN}{TP + TN + FN + FP} \tag{1}$$

$$Recall = \frac{TP}{TP + FN} \tag{2}$$

$$Precision = \frac{TP}{TP + FP} \tag{3}$$

$$Specificity = \frac{TN}{TN + FP} \tag{4}$$

4 Results

In this section state of the art gene prediction tools found in the reviewed literature are presented and discussed. This addresses objective number one and two of this paper. For more details on the articles reviewed for this section of the paper see Appendix A.

Wang, Chen, and Li [19] states that many different algorithms have been used over the years for modelling gene structure, some of the most common and successful ones are dynamic programming, hidden markov models, linear discriminant analysis, linguist methods and artificial neural networks. Other machine learning algorithms have also been explored in recent years and are very relevant when it comes to the development of improved gene prediction tools.

4.1 Categorization of Existing Solutions

In Table 2 we see the gene prediction tools that have been selected for this paper. They are categorized based on the techniques they implement to make them easier to present. These tools were selected because they are recent and state-of-the-art models found in the field and literature. Some tools implement similar techniques, some are based on other gene prediction tools and some use more modern techniques from machine learning. GeneRFinder is a machine learning based tool, and was included in this paper to serve as a modern comparison to other state of the art tools.

Prediction tool	Article	Publication dates	Basic approach
FragGeneScan	[20–22]	2021, 2010	Markov model based
GeneMarkS	[20, 23, 24]	2021, 2012, 2001	Markov model based
GeneMark.hmm	[20, 23, 25]	2021, 2012, 1998	Markov model based
GLIMMER	[20, 23, 26]	2021, 2012, 2011	Markov model based
Prodigal	[20, 21, 23, 27]	2021, 2012, 2010	Dynamic programming
MetaGene	[20, 21, 28]	2021, 2006	Stochastic approach
geneRFinder	[21]	2021	Machine learning algorithm

Table 2: Categorization of gene prediction tools depending on their fundamental approaches.

Gene prediction tool	Model or non-model based
FragGeneScan	Model based
GeneMark.hmm	Model based
geneRFinder	Model based
GLIMMER	Non-model based
Prodigal	Non-model based
MetaGene	Non-model based
GeneMarkS	Non-model based

Table 3: Categorization of gene prediction tools depending on whether they are model based or not.

In Table 3 the categorization of gene prediction tools is performed depending on if the tools are model based or not. Model based prediction tools require a rigid set of parameters that are pre-tuned to a particular organism before performing the actual prediction [20]. GeneRFinder is in this category as it uses a supervised machine learning algorithm, meaning it requires a training set and training before actually performing predictions.
According to Dimonaco et al. [20], model based tools (in this case FragGeneScan and GeneMark.hmm) perform worse than tools that do not need to be pre-tuned to a particular organism. They say this may be due to the fact that rigid model-based methods are unable to pick up subtle feature differences that may occur in even strains from the same species. In some cases model based tools are shown to perform satisfactory when the target and model genomes are different. However, the unreliability of these tools give non-model based tools such as Prodigal, GeneMarkS and GLIMMER an advantage.

4.2 Hidden Markov Model Approach

Markov models describe the process of moving from one state to another where the probability of each event depends on the state attained in the previous event [29]. This is useful when needing to compute the probability for a sequence of observable events, although, in many cases the events we are interested in may be hidden. In this case these hidden events could be the

transition between coding and non-coding genes in a DNA sequence for instance.
We can think of these hidden events as causal factors in our probabilistic model, and that the hidden states also influence the transition probabilities [30]. In these cases hidden Markov models (HMMs) can be used to address both the observable events as well as the hidden ones. Eddy [31] states that HMMs are "a formal foundation for making probabilistic models of linear sequence 'labeling' problems", and are therefor at the heart of a diverse range of tools in genetics. GeneMark.hmm, FragGeneScan and GeneMarkS all use an architecture based on hidden markov models.

4.2.1 GeneMark.hmm

GeneMark is a generic name for a number of ab initio gene prediction programs. GeneMark.hmm was designed to improve the gene prediction accuracy of short genes and gene starts [32]. The original GeneMark tool used a similar Markov-chain algorithm, so the purpose of GeneMark.hmm was to integrate this into a hidden Markov model framework. According to Lukashin and Borodovsky [25] it is based on the hidden markov model architecture and uses a modified version of the Viterbi algorithm to determine the most likely sequence of hidden states [23]. The transition between coding and non-coding regions is interpreted as the transition between hidden states. To avoid predicting overlapping genes as shorter than they are GeneMark.hmm uses a a post-processing step which picks alternative gene starts when scores are above a certain threshold.

4.2.2 FragGeneScan

In the article by Rho, Tang, and Ye [22] a common challenge for gene predictor tools developed for whole genomes is said to be identifying genes directly from short reads, or reads where the sequencing error rates are high. FragGeneScan is said to be developed for the purpose of improving the prediction of protein coding genes in short reads, by combining sequencing error models and the codon usage in a hidden markov model.

4.2.3 GeneMarkS

Besemer, Lomsadze, and Borodovsky [24] state that GeneMarkS is a combination of GeneMark.hmm and the Gibbs sampling method, and it was the next step in the development of the GeneMark family of gene prediction programs. It is a self-training gene prediction tool and uses the heursitic Markov models to run GeneMark.hmm. It then uses the prediction output from this step to build new models and produce new predictions. The process is repeated until the predictions from two steps are sufficiently related [23].

4.2.4 GLIMMER

Glimmer uses interpolated markov models (IMM), which is a variable-length Markov model, for capturing gene composition as stated by Engebretsen [23]. According to Kelley et al. [26] Glimmer uses a flexible ORF-based framework to differentiate the sequences into coding and non-coding genes. This framework incorporates how prokaryotic genes can overlap upstream features like ribosomal binding sites (RBS). Glimmer extracts the longest ORFs and scores them using log-likelihood ratio. In order to reduce the number of false positives that appear due to overlapping ORFS, dynamic programming is used to find the set of ORFs with a maximum score given the constraint that genes cannot overlap for more than a certain threshold.

4.3 Prodigal

Prodigal uses a dynamic programming based architecture and is regarded as one of the most popular prediction softwares these days [14]. It is easy to use, runs fast and the results seem to be at least as good as other tools. In Table 3 we can see that Prodigal is a non-model based tool, meaning that it is unsupervised. This is possible because Prodigal constructs a training set of genes by extracting necessary properties such as start codon usage, RBS motif usage, GC frame plot bias and other information required to build a sufficient training profile. From the GC bias information prodigal creates preliminary coding scores for each gene. These scores are then later used in a series of dynamic programming connections to score every ORF longer than 90 base pairs in the entire genome. The implementation of Prodigal is quite complex and is described in detail in the article by Hyatt et al. [27].

4.4 MetaGene

According to Noguchi, Park, and Takagi [28], MetaGene uses log-odds ratio for scoring ORFs throughout the algorithm. The process of gene prediction of a given sequence can be divided into two stages. The first stage involves extracting all ORFs and giving them a score based on their composition and length. In the second stage a high-scoring combinations of ORFs are derived by using their scores of orientations and distances to neighbouring ORFs, in addition to the scores from stage one. This two-fold approach makes it possible to predict overlapping genes with appropriate scores [28].

4.5 geneRFinder

The geneRFinder uses an algorithm called Random Forest (RF) classifier which learns from properties of sequences extracted from ORFs, and makes predictions based on features captured from these regions, as stated by Silva et al. [21]. The RF classifier is a supervised machine learning algorithm, based on the decision tree algorithms, used to solve regression and classification problems. Briefly explained, it is a collection of multiple random decision trees and by using methods such as bagging and multiple iterations it becomes much less sensitive to the training data, and thereby reduces overfitting and increases precision. After the model has been trained it can be tested on independent datasets having different genome complexities and sequence sizes and still identify coding sequences.

4.6 Comparing Performance

In this subsection we discuss and compare the performance of the gene prediction tools presented in section 4. Engebretsen [23] is the oldest article selected for this review. In its comparison of state of the art prediction tools, the tools Prodigal and GeneMarkS perform the best in terms of accuracy, precision and recall. In the study it is expected to be approximately 900 genes per megabase in the target genome. Of which Prodigal and GeneMarkS predicts 914 and 899 genes per megabase respectively. This indicates that GeneMarkS is slightly more conservative than Prodigal, considering it also made some false positive predictions. Prodigal was able to predict more of the annotated genes and despite being less strict in its predictions it also achieved better precision and recall scores. GeneMark.hmm and GLIMMER on the other hand were too little conservative in their predictions, as they both predicted more than 950 genes, meaning they predicted many false positives. The fact that GeneMarkS performs better than GeneMark.hmm in terms of prediction performance supports the claim that GeneMarkS is an improvement of GeneMark.hmm. When comparing GLIMMER to the other state of the art programs it is mentioned that GLIMMER is more versatile than the other programs, meaning its performance could potentially be improved by more domain knowledge.

In contrast to Engebretsen [23], the article written by Dimonaco et al. [20] compares a wide range of gene prediction tools for the purpose of proving that their performance highly depends on the organism of study. It concludes the comparison with that Prodigal was ranked the best overall, being the most well-rounded tool and performing best in terms of the 12 chosen metrics. However, in regards to its performance on specific genomes it only ranked the best when predicting genes in two out of six model organisms. This supports the suggestion that prediction tools should be carefully selected based on the organism and questions at study.

Silva et al. [21] tested the performance of geneRFinder on a dataset of 12 geneomes. When determining whether a sequence was coding or not the sequence length was considered the most important feature. The sequence lengths in the dataset ranged from 100 to 2000 base pairs and geneRFinder achieved an accuracy and sensitivity score of 75 percent for sequences of all lengths, and a score over 90 percent for sequences longer than 600 bp. In other words, the longer the sequence is the easier it is to classify if the sequence is coding or not. However, geneRFinder's performance on shorter sequences was also sufficient. Prodigal and FragGeneScan were also used in this study as a comparison to geneRFinder. FragGeneScan is considered one of the best gene prediction tools for shorter sequences, while Prodigal on the other hand is often used as a complementary tool in annotation pipelines to identify the longer sequences [21].

Due to the fact that varying inputs to gene prediction tools can affect their performance differently, Silva et al. [21] performed their study using standard datasets compiled to provide a fair gene prediction benchmark when evaluating gene predictors. Four datasets were used, one test dataset, one dataset with low genome complexity, one with medium genome complexity and one with high genome complexity. In all the four cases the accuracy and specificity score for geneRFinder was over 90 percent, and between 20 to 70 percent better than Prodigal and FragGeneScan. The exact accuracy, recall and sensitivity score for all the three tools can be found in Table 4. Prodigal was second best in terms of accuracy and specificity, but FragGeneScan had the highest score for sensitivity of 99 percent for all four datasets. However, both geneRFinder and Prodigal had more than satisfactory scores above 90 percent as well. In other words FragGeneScan and Prodigal were slightly less conservative in predicting genes, however, they struggled to identify which genes were non-coding. GeneRFinder on the other hand can achieve high specificity when trained to find non-coding sequences as well, and not just genes. This is because its selected features capture the signals present in coding and non-coding sequences independently, as opposed Prodigal and FragGeneScan which use features based on other segments of the sequence such as the ribosomal binding site for instance.

Prediction tool	Accuracy	Recall	Specificity
geneRFinder	**93.8%**	**94.6%**	**93.6%**
Prodigal	40.1%	97.6%	28.6%
FragGeneScan	29.6%	99.9%	15.5%

Table 4: Accuracy, recall and specificity scores from gene predictions on the medium complexity dataset. Taken from Silva et al. [21]. GeneRFinder clearly performs the best overall in comparison to Prodigal and FragGeneScan.

4.7 Other Machine Learning Approaches

Apart from the RF classifier, other machine learning approaches have also been tested for the purpose of gene prediction. In this section these will be briefly presented. Campos et al. [17]

tested algorithms such as Generalised Linear Model (GLM), Artificial Neural Networks (NN), Gradient Boosting Method (GBM), Support Vector Machines (SVM) and RF classifier for the purpose of identifying essential genes in eukaryotes.
The results showed that prediction performance and the selected best predictive features varied with which machine learning algorithm used and species studied. All the machine learning algorithms used here outperformed random guessing based on true probabilities. However, RF's performance was superior to all other machine learning algorithms in most cases, and all the algorithms prediction performance increased in correlation with more data being added to training datasets.

5 Discussion

In this section the challenges with gene prediction and existing tools is discussed, and potential future directions are presented. This addresses objective three of this paper. Despite much progress in gene prediction tools over the past decade, the search for improved methods has somewhat stagnated. Other methods within genomics seem to be prioritised despite gene prediction being an important step in many pipelines within structural and functional genomics.
The importance of accurate predictions of protein coding genes has never been greater and there are still interesting and challenging research directions that deserve further efforts. Figure 5 illustrates a summary of the challenges discussed in Subsection 5.1 and the future directions presented in Subsection 5.2.

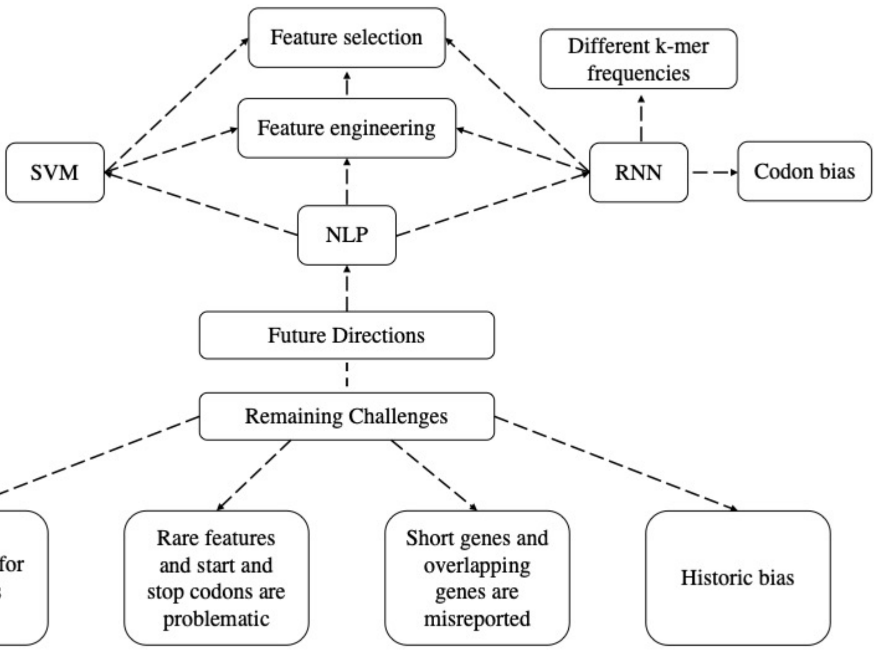

Feature selection
Different k-mer frequencies
Feature engineering
SVM
RNN
Codon bias
NLP
Future Directions
Remaining Challenges
No one tool for all species
Rare features and start and stop codons are problematic
Short genes and overlapping genes are misreported
Historic bias

Figure 5: Summary of section 5. The figure illustrates the remaining challenges with gene prediction and existing tools used for this purpose. It also presents possible future directions for the development of improved gene prediction tools. See Section 5.2 for explanation of abbreviations.

5.1 Remaining Challenges

As learnt after reading the paper by Dimonaco et al. [20], a prominent challenge with gene prediction tools is that different state-of-the-art tools appear to work differently depending on the species. In addition, certain types of genes such as short genes, overlapping genes and those with alternative codon usage are handled differently and are still tough to classify even for the best performing tools.

Rare features and the accurate finding of start codons are still problems for state-of-the-art tools according to Engebretsen [23]. *Rare features* refers to cases such as prokaryotic introns or cases where start and stop codons encode rare amino acids depending on RNA motifs located upstream of the start or stop codon. Engebretsen [23] suggest these features might be best handled outside of the gene prediction tools.

Dimonaco et al. [20] states that short genes and overlapping sequences are often misreported. A majority of the state-of-the-art tools still have hard-coded limitations when it comes to minimum ORF length, as well as algorithmic weights against short ORFs. This is a problem as short genes are both common and important in prokaryotic genomes. Overlapping genes occurring either in the same reading frame or a different one are also hard to handle. Most tools return either only one of the overlapping genes or neither of them in the predictions. Dimonaco et al. [20] also mention that there is a general under-representation of short genes in genomic databases, which brings us to the final challenge: historic bias.

When assessing the performance of a gene prediction tools we often use a ”verified” reference genome. These have been thoroughly studied, sequenced, assembled and annotated. However, there is no real ground truth and there exists no actual verified genes. Historic biases like this have over the years caused biases in ORF prediction tools, as well as gene prediction tools. In turn this hinders and taints the discovery of new genes and gene prediction tools, as their verification is based on current, possibly incorrect, knowledge.

5.2 Potential Future Directions

In Campos et al. [17] the SVM algorithm is used among other machine learning algorithms. This is the only machine learning tool for *Natural Language Processing* (NLP) used in any of the articles reviewed in this paper. The article concludes by stating other machine learning methods should also be explored for the purpose of gene prediction, such as deep learning [33]. *Recurrent Neural Networks* (RNN) is a deep learning method that can be useful for modelling sequential data [34]. Sequential data is essentially just ordered data in which related data follow in sequence to each other, like for instance a DNA sequence.

In order for a machine learning model to be able to make sense of a sequence of words or letters one may need to perform some basic transformations to create some type of numeric representation, also called feature engineering. Machine learning algorithms then examine these features to find patterns. There are several different feature engineering methods worth exploring, and when it comes to DNA sequences a particularly relevant method is for instance n-grams (k-mer frequencies).

N-grams is a method used in NLP, which preserves the context of the ”words” in a sequence. It uses a (n-1)-order Markov model to predicts the next item in a sequence, and the larger the ”n” the more context is stored in the model. By calculating and comparing profiles of n-gram frequencies one can use n-grams for text categorization. In a DNA sequence an n-gram can take on 4^n values, and a 1-gram would be a single base, and a 3-gram would be a codon, i.e. a triplet of bases. However, a simple 3-gram representation would not take the alternative reading frames into account, which is necessary in the case of DNA sequences and gene prediction.

Throughout this paper several potential features in DNA sequences that can be relevant for gene prediction tools have been mentioned (Section 3.5). The selection of the most suitable features is also a potential area of focus for future studies, as well as the selection of reliable data that can be used for benchmarking purposes.

6 Conclusion

The objectives of this review were to: find current methods used for the prediction of protein coding genes in prokaryotes, find out if machine learning tools are applicable for predicting protein coding genes and what the current challenges are with the prediction of protein coding genes. 30 articles were screened, and 16 articles were included to fulfill these objectives.

Having reviewed several state-of-the-art gene prediction tools, as well as more modern computational methods such as geneRFinder, we can see how the older gene prediction tools were based on the power of Hidden Markov Models, whilst the newer are mostly based on machine learning techniques. This paper provides a basis for how state-of-the-art gene prediction tools are comparable to more modern tools using machine learning approaches. Future work on new computational methods could include exploring different feature engineering and feature selection techniques, recurrent neural networks and other deep learning methods used for natural language processing.

References

[1] Bayat A. "Science, medicine, and the future: Bioinformatics." In: *BMJ Bioinformatics* 324 (2002), pp. 1018–2022. ISSN: 0959-8138. URL: https://www.ncbi.nlm.nih.gov/pmc/articles/PMC1122955/#:~:text=Bioinformatics.

[2] Wikipedia contributors. *Gene Prediction.* 2021. URL: https://en.wikipedia.org/wiki/Gene_prediction.

[3] Yohann Jacob Sandvik. "A Literature Review of Time-Series Clustering Techniques and Machine Learning Techniques Used for Monitoring of Wind Turbines." In: (2019). URL: https://github.com/yohannjs/project_assigment.

[4] NIH National Human Genome Research Institute. *Nucleotide.* 2019. URL: https://www.genome.gov/genetics-glossary/Nucleotide.

[5] Susha Cheriyedath. *Start and stop codons.* 2019. URL: https://www.news-medical.net/life-sciences/START-and-STOP-Codons.aspx.

[6] Oxford Languages and Google. *RNA.* URL: https://languages.oup.com/google-dictionary-en/.

[7] Wikipedia contributors. *Ribosome.* URL: https://en.wikipedia.org/wiki/Ribosome.

[8] BioNinja Brent Cornell. *Translation.* URL: https://ib.bioninja.com.au/standard-level/topic-2-molecular-biology/27-dna-replication-transcri/translation.html.

[9] Genome.gov. *Non-coding genes.* URL: https://www.genome.gov/genetics-glossary/Non-Coding-DNA.

[10] Lars Snipen. *BIN310 - module 3 lecture.* URL: http://arken.nmbu.no/~larssn/teach/bin310/module2.html.

[11] Lars Snipen. *BIN310 - module 6 lecture.* 2021. URL: https://www.youtube.com/watch?v=sSfk6ww01A4.

[12] Sagar Ayal. *Gene Prediction - Importance and methods.* URL: https://microbenotes.com/gene-prediction-importance-and-methods/.

[13] Wikipedia. *Open reading frame.* 2021. URL: https://en.wikipedia.org/wiki/Open_reading_frame.

[14] Lars Snipen. *BIN310 - module 6 lecture.* 2021. URL: https://www.youtube.com/watch?v=sSfk6ww01A4.

[15] Wikipedia. *Start codon.* 2021. URL: https://en.wikipedia.org/wiki/Start_codon..

[16] Lumen. *Genetic Code.* URL: https://courses.lumenlearning.com/wm-nmbiology1/chapter/reading-codons/.

[17] Tulio L. Campos et al. "An Evaluation of Machine Learning Approaches for the Prediction of Essential Genes in Eukaryotes Using Protein Sequence-Derived Features". In: *Computational and Structural Biotechnology Journal* 17 (2019), pp. 785–796. ISSN: 2001-0370. URL: https://doi.org/10.1016/j.csbj.2019.05.008.

[18] Arthur M. Lesk. *Introduction to Bioinformatics.* 1st ed. Oxford University Press, 2013, pp. 109–136.

[19] Zhuo Wang, Yazhu Chen, and Yixue Li. "A Brief Review of Computational Gene Prediction Methods". In: *Genomics, Proteomics & Bioinformatics* 2.4 (2004), pp. 216–221. ISSN: 1672-0229. URL: https://www.sciencedirect.com/science/article/pii/S1672022904020285.

[20] Nicholas J. Dimonaco et al. “No one tool to rule them all: Prokaryotic gene prediction tool performance is highly dependent on the organism of study”. In: *Cold Spring Harbor Laboratory* (2021). URL: https://www.biorxiv.org/content/10.1101/2021.05.21.445150v1.full.

[21] Raíssa Silva et al. “geneRFinder: gene finding in distinct metagenomic data complexities.” In: *BMC Bioinformatics* 22.87 (2021). ISSN: 1471-2105. URL: https://doi.org/10.1186/s12859-021-03997-w.

[22] Mina Rho, Haixu Tang, and Yuzhen Ye. “FragGeneScan: predicting genes in short and error-prone reads.” In: *Nucleic Acids Research* 38.20 (2010). URL: https://academic.oup.com/nar/article/38/20/e191/1317565.

[23] Stian Engebretsen. “Evaluation of gene prediction methods for prokaryotes”. In: *Univeristy of Oslo, Department of Informatics* (2012). URL: https://www.duo.uio.no/bitstream/handle/10852/34173/thesis.pdf?sequence=1&isAllowed=y.

[24] John Besemer, Alexandre Lomsadze, and Mark Borodovsky. “GeneMarkS: a self-training method for prediction of gene starts in microbial genomes.” In: *Nucleic Acids Research* 29.12 (2001). URL: https://doi.org/10.1093/nar/29.12.2607.

[25] AV Lukashin and M. Borodovsky. “GeneMark.hmm: new solutions for gene finding.” In: *Nucleic Acids Research* 26.4 (1998). URL: https://academic.oup.com/nar/article/26/4/1107/2902172.

[26] David R. Kelley et al. “Gene prediction with Glimmer for metagenomic sequences augmented by classification and clustering”. In: *Nucleic Acids Research* 40.1 (2011). URL: https://doi.org/10.1093/nar/gkr1067.

[27] Doug Hyatt et al. “Prodigal: prokaryotic gene recognition and translation initiation site identification.” In: *BMC Bioinformatics* 11.9 (2010). URL: https://doi.org/10.1186/1471-2105-11-119.

[28] Hideki Noguchi, Jungho Park, and Toshihisa Takagi. “MetaGene: prokaryotic gene finding from environmental genome shotgun sequences.” In: *Nucleic Acids Research* 34.19 (2006). URL: https://doi.org/10.1093/nar/gkl723.

[29] Wikimedia Foundation. *Markov chain.* URL: https://en.wikipedia.org/wiki/Markov_chain..

[30] Stanford University. *Chapter A - Hidden Markov Models.* URL: https://web.stanford.edu/~jurafsky/slp3/A.pdf..

[31] Sean R. Eddy. “What Is a Hidden Markov Model?” In: *Nature News, Nature Publishing Group* (2004), pp. 1315–1316. URL: https://www.nature.com/articles/nbt1004-1315.

[32] Wikimedia Foundation. *Genemark.* URL: https://en.wikipedia.org/wiki/GeneMark..

[33] Gökcen Eraslan et al. “Deep learning: new computational modelling techniques for genomics.” In: *Nat Rev Genet* 30 (2019), pp. 389–403. URL: https://doi.org/10.1038/s41576-019-0122-6.

[34] Lexalytics. *Natural Language Processing (NLP).* 2021. URL: https://www.lexalytics.com/lexablog/machine-learning-natural-language-processing#unsupervised..

Appendices

Appendix A: Summary of articles included from search 1 and 3

Ref.	Title	Pub. date	Tools featured	Dataset used
20	"No one tool to rule them all: Prokaryotic gene prediction tool performance is highly dependent"	2021	FragGeneScan, GeneMark.hmm, GeneMarkS, GLIMMER, Prodigal, MetaGene	Six bacterial model organisms and their canonical annotations
23	"Evaluation of gene prediction methods for prokaryotes"	2012	GeneMark.hmm, GeneMarkS, GLIMMER, Prodigal, MED	1262 prokaryotic genomes from NCBI Entrez Genome database, for 902 different species. Of which 818 were bacteria, 84 archea and 3 bacteriophages.
21	"geneRFinder: gene finding in distinct metagenomic data complexities"	2021	geneRFinder, Prodigal, FragGeneScan, Orphelia, MetaGene	12 public genomes and respective annotations, 3 archea, 9 bacteria
17	"An Evaluation of Machine Learning Approaches for the Prediction of Essential Genes in Eukaryotes Using Protein Sequence-Derived Features"	2019	Gradient Boosting Method, Generalised Linear Model, Neural Network, Random Forest, Support-Vector Machines	Six model eukaryotic species from the Online Gene Essentiality database (OGEE)
26	"Gene prediction with Glimmer for metagenomic sequences augmented by classification and clustering"	2011	GLIMMER	Simulated datasets from 1206 prokaryote genomes in GenBank
27	"Prodigal: prokaryotic gene recognition and translation initiation site identification"	2010	Prodigal	The EcoGene (genome sequence database for Escherichia coli) Verified Protein Starts set
22	"MetaGene: prokaryotic gene finding from environmental genome shotgun sequences"	2006	MetaGene	The complete genomic sequences and the annotations of coding regions of 143 microorganisms that are available in GenBank.
28	"GeneMarkS: a self-training method for prediction of gene starts in microbial genomes"	2001	GeneMarkS	12 genomes available in the GenBank database.
24	"GeneMark.hmm: new solutions for gene finding."	1998	GeneMark.hmm	12 complete prokaryotic genomes
25	"FragGeneScan: predicting genes in short and error-prone reads"	2010	FragGeneScan	A total of nine complete genomes (with various GC contents) and their annotations were downloaded from the NCBI website