Master's Thesis 2024 30 ECTS
Faculty of Science and Technology(REALTEK)

Deep learning for Direct DNA Domain Detection

August Noer Steinset
Data Science

Master's Thesis

Deep learning for Direct DNA Domain Detection

Written by:
August Noer Steinset

Supervisor:
Krisitan Hovde Liland

Co-supervisor:
Lars-Gustav Snipen

Norwegian University
of Life Sciences

Master of Science
Faculty of Science and Technology
May 15, 2024

Preface

Finalizing this thesis marks the end of my five years at NMBU. It has been five interesting years, and I am thankful for everything I have learned and all the relationships I have established during this phase of my life.
For the writing of this thesis, I would like to thank my supervisors, Kristian and Lars. Discussing the process and results with them has always been engaging, and I have left feeling assured and inspired for every meeting we have had.
I also want to thank my friends and family for maintaining normalcy in this unique semester. And a special thanks to Elise, whose unrelenting diligence almost inspired me.

Abstract

As more and more genomes are sequences it becomes ever more needed to have access to fast methods of analyzing them. One such analysis is finding regions of the genome that contain descriptions of proteins we are familiar with. Pfam is a database containing these descriptions, and tools like HMMER can match known proteins/protein substructures onto sequences. This process is slow, and this thesis explores how to speed it up by creating machine-learning models that filter out the sequences less likely to contain known proteins.
This is done by analyzing large datasets containing hundreds of species sampled by different means and by employing different target values. Through this valuable insight such as high GC content negatively impacting model performance and that machine learning models such as convolutional neural networks can describe specific Pfam patterns close to perfectly, with accuracies of 99.7% on balanced datasets.
The end result of the thesis is that a filtering mechanism should be possible to create, but it would require significantly more work to get it working to a degree where it both included the necessary. Ensuring that enough Pfam entries are represented would be a good starting step.
One of the models this thesis employs is the Tsetlin machine, hoping its unique structure would make it well suited to such data. The results did, however, not show this to be the case. While the Tsetlin machine might still be well suited for similar data, some changes would have to be made to how they were employed in this thesis.

Contents

1 Introduction **5**

2 Theory **7**
- 2.1 From DNA to protein . . . 7
  - 2.1.1 DNA . . . 7
  - 2.1.2 The Central dogma . . . 10
  - 2.1.3 Protein Families . . . 10
  - 2.1.4 Taxonomy . . . 11
- 2.2 Pfam and HMMER . . . 11
  - 2.2.1 Hidden Markov Models . . . 11
  - 2.2.2 Pfam . . . 12
  - 2.2.3 HMMER . . . 12
- 2.3 Machine Learning Theory . . . 12
  - 2.3.1 XGBoost . . . 13
  - 2.3.2 Neural Networks . . . 13
  - 2.3.3 Tsetlin Machine . . . 14
  - 2.3.4 Weighted clauses . . . 17
- 2.4 Metrics . . . 17
  - 2.4.1 Accuracy . . . 18
  - 2.4.2 Precision . . . 18
  - 2.4.3 Recall . . . 18
  - 2.4.4 F1 . . . 18
- 2.5 Related Work . . . 18

3 Method **19**
- 3.1 The original data . . . 19
- 3.2 Constructing the datasets . . . 19
  - 3.2.1 The General Dataset . . . 20
  - 3.2.2 The Order-Based Datasets . . . 20
  - 3.2.3 The Specific-Pfam Datasets . . . 20
- 3.3 Data preprocessing . . . 21
  - 3.3.1 Limiting the Length . . . 22
  - 3.3.2 Ballancing the Data . . . 22
  - 3.3.3 One hot encoding . . . 22
- 3.4 Creating Training, Test and Validation Sets . . . 24
- 3.5 Creating Models . . . 24
  - 3.5.1 Tsetlin Machine . . . 25
  - 3.5.2 Convolutional Tsetlin Machine . . . 25

3.5.3 XGBoost 25
3.5.4 Dense neural network 25
3.5.5 Convolutional Neural network 26
3.6 Code 27

4 Results **28**
4.1 General 29
4.2 Order 33
4.2.1 Corynebacteriales and Lactobacillales 35
4.2.2 Summary Order-based 38
4.3 Specific Pfam Domains 39
4.3.1 Distrubution of Pfam domains 39
4.3.2 One Pfam domain 40
4.3.3 Multiple Pfam Domains 42
4.3.4 Pfam 1-20 42
4.3.5 Results of epochs for TM 44
4.3.6 Summary specific Pfam 44
4.4 Summary results 44

5 Discussion **46**
5.1 Model results analysis 46
5.1.1 Capturing patterns 47
5.1.2 Length of sequences 48
5.1.3 Selecting a Model 48
5.2 Tsetlin machine results 48
5.2.1 Number of epochs 48
5.2.2 Number of clauses 49
5.2.3 Alternatives to direct amino sequences 49
5.2.4 Convolutional models 49
5.3 Creating a better model 50
5.3.1 Choosing an approach 50
5.3.2 Challenges 50
5.3.3 Reevaluating the model 52
5.4 Reflections on methodology 53
5.5 Use of AI 54

6 Conclusion **56**

A Table of R-packages, for instance **59**

List of Figures

4.1 CNN super Results on General 30
4.2 Tsetlin Machine Results on General 30
4.3 CNN super on General 30
4.4 XGBoost on General 30
4.5 Probabilitity Distribution for CNN Super on the Gerneral dataset 31
4.6 Probabilitity Distribution for XGBoost on the Gerneral dataset 32
4.7 Accuracies of different models on the order-based datasets 33
4.8 GC Content of Ordered Datasets 34
4.9 Plot showing the length of LORFs within the Corynebacteriales dataset together with the portion of results that results in a positive Pfam hit. 35
4.10 Plot showing the length of LORFs within the Lactobacillales dataset together with the portion of results that results in a positive Pfam hit. 35
4.11 CNN super on Corynebacteriales 36
4.12 TM on Corynebacteriales 36
4.13 CNN super on Lactobacillales 36
4.14 TM on Lactobacillales 36
4.15 CNN super on Corynebacteriales 37
4.16 CNN super on Lactobacillales 37
4.17 The accuracies of different model probabilities as well as the distribution of model probabilities for CNN Super on the Corynebacteriales dataset 37
4.18 The accuracies of different model probabilities as well as the distribution of model probabilities for CNN Super on the Lactobacillales dataset 38
4.19 Log transformed counts of the Pfam domains in dataset 39
4.20 CNN super accuracies on the 20 different specific Pfam datasets. Lower bound at 50% accuracy 40
4.21 Tsetlin machine accuracies on the 20 different specific Pfam datasets. Lower bound at 50% accuracy 40
4.22 CNN super on Pfam 1 41
4.23 TM on Pfam 1 41
4.24 CNN super on multiple Pfam 42
4.25 TM on multiple Pfam 42
4.26 Roc Curve for CNN Super on Pfam 1-20 43
4.27 Probability Bins for CNN Super on Pfam 1-20 43

List of Tables

2.1 Reading Frames . . . 8
2.2 Standard Genetic Code Table . . . 9
2.3 Tsetlin automaton . . . 14
2.4 Tsetlin automaton with reward . . . 14
2.5 Tsetlin automaton with reward, changing signs . . . 15
2.6 Type 1 feedback for included literal . . . 16
2.7 Type 1 feedback for excluded literal . . . 16
2.8 Type 2 feedback for included literal . . . 16
2.9 Type 2 feedback for excluded literal . . . 17

3.1 The sequence ABA . . . 22
3.2 1D one hot encoding . . . 23
3.3 2D one hot encoding . . . 23

4.1 Performance on the General Dataset . . . 29
4.2 Overview of Pfam entries for Pfam 1-20 . . . 40
4.3 Performance on the Pfam1 dataset . . . 41
4.4 Performance of models on the Pfam 1-20 dataset . . . 42
4.5 Performance Metrics by number of Epochs for CTM . . . 44
4.6 Performance Metrics by number of Epochs for TM . . . 44

A.1 R-packages used to construct datasets . . . 59
A.2 Python packages used for both models and vizualisations . . . 59

Chapter 1

Introduction

With the advent of new technologies within DNA sequencing, a vast quantity of genetic data has become available to us. Only a couple of decades ago, at the tail of the human gene project, the cost of sequencing a single human genome was estimated to cost 100,000$; today, it is estimated to cost a percent of that[1]. And if the development of sequencing technologies continues, the tools and avenues of research that depend on it must strive for the same. With all this new data, the potential for insight into the fundamentals of life becomes ever closer to being within man's grasp.

Metagenomics is a new research area that has become commercially viable thanks to these developments. By looking at the genetic composition of entire environments, not just individual organisms, it is possible to obtain the genetic profiles of microbial species never seen. When one of these environments is sequenced, the result is a large collection of DNA fragments known as reads. These fragments are then compared to others and, when it makes sense, merged into larger units known as contigs. These contigs can then possibly be merged into entire genomes but can also contain interesting information by themselves.

With all this data, comes the desire to set it into systems and gather metadata; to explore how it relates to already categorized data. Questions such as what organisms it belongs to, if new, how it is placed within the established taxonomy, and in the case of microorganisms, how it genetically interacts with other organisms in its environment.

One of these key analyses is trying to understand which sequences contain pieces of genetic code to be created into protein. To do this the areas of the genome, or contig, that hold the potential to hold genes are identified as Open Reading Frames(ORFs). This is done by locating the sequences that signal the termination of protein synthesis and the sequences that signal the beginning and selecting the area between. Further one might wish to understand what kind of function the given protein will have in the organism it was created in and what is a better way than to look at its relatives.

Proteins exist in many forms; in the same way, organisms relate to one another, as do proteins. Through the process of evolution, proteins have mutated or changed from other factors, again and again. Ultimately, this results in many variations for proteins that once had a common origin. This relationship is not limited to the entirety of a protein either, parts of proteins can be specifically preserved, while the rest varies. When one of these regions is recognized within a DNA sequence, there is a good chance it is related to other proteins we know the functionality of, making it possible to infer the new proteins function as well.

One way these protein domains have been described is by using profile hidden Markov models. Stored in databases such as the Pfam database, it has become possible to compare any given sequence to every one of these patterns using tools like HMMER and find where it matches up. Unfortunately, this process is quite time-consuming, so reducing the amount of data analyzed would be preferable. One possible avenue to do this is through employing machine learning algorithms.

As computational resources continue to become more available, the field of machine learning has experienced a rise in popularity and applications. One of its most resounding victories in protein has been solving the problem of protein folding by Google's DeepMind(link). By creating models by observing large amounts of data and seeing how it relates to a target variable, patterns that are not intuitive to the human mind can show themselves and be used to make predictions on data not yet seen.

Within machine learning, a wide array of tools can handle many challenges, from the simple linear regression method to the more complex deep learning methods such as convolutional neural networks. One new tool that aims to provide an alternative to deep learning methods is the Tsetlin machine. The model has been shown to perform similarly to traditional methods on well-known datasets such as the MNIST, and with it being based on analyzing binarized data it might be particularly suited for genetic data. While DNA itself is not binary in nature, its 4 possible bases make it possible to convert to binary while at the same time not being particularly sparse.

This study aims to analyze how well modern machine learning methods could learn to recognize how likely any given sequence of genetic information is to contain any of these recognized protein families or protein regions. This will be done with the hope it can be used to develop a filter that can be run on the sequence data before it is fed to tools such as HMMER so that only the sequences likely to contain some sort of recognized protein structure will be fed to it. A second aim is to see how well the Tsetlin machine functions compared to the more established machine learning methods and to see if its unique structure makes it more suited for sequence data.

Chapter 2

Theory

As the main focus of this thesis is evaluating the performances of models and trying to build a method for identifying ORFs that contain Pfam entries, the theory presented will focus on building an understanding of what these terms mean.
To do this what constitutes both an ORF and a Pfam entry will be explained, together with an explanation of how they are found. It will be presented by starting with the DNA and building towards more complexity. Most information presented should be common knowledge within their respective fields of biology/machine.
This section will also give an overview of how the machine learning methods employed in the methodology section, 3, operate. However, it will focus more on presenting an intuition of how these work, rather than going into the details of their mathematical foundations.

2.1 From DNA to protein

2.1.1 DNA

Deoxyribonucleic acid, often referred to as its short form DNA, is the fundamental building block of life. It contains the genetic blueprint that instructs the development and functioning of all living organisms as we know them. Structurally, DNA is composed of two long strands twisting around one another forming their characteristic double helix structure, first described by Watson and Crick in their 1953 paper "Molecular Structure of Nucleic Acids: A Structure for Deoxyribose Nucleic Acid"[2]. A sequence of sugars and phosphate groups constructs the strands themselves. These stands are connected through a complementary set of nucleotides, nitrogenous bases connected by hydrogen bonds. Adenine(A) connects with thymine(T) and Guanine(G) connects with cytosine(C).

DNA's primary function is to store genetic information so that it can be used by a cell to produce any protein it might need. As the environment around an organism changes, it, too, must change. The mechanisms through which an organism can adapt its genetic code to the environment are many, from the random changes brought by mutations to the exchange of genetic information from similar organisms. The result is that DNA is carried on through time, but that it also changes. As time has passed, this has resulted in organisms diverging from one another, forming the plethora of organisms that draw the tree of life together. One of these divergences is the split between the prokaryotes (bacteria and arcae) and eukaryotes, and this thesis will focus on the former of the two.

Unlike eukaryotes, prokaryotes do not store their DNA in a protected membrane within their cell; rather, it floats freely in the cytoplasm, often in a single circular chromosome. This compact chromosome gives the prokaryotes a much larger gene-to-non-gene ratio than their eukaryotic counterparts, making exploratory research simpler.

The sequence of nucleotides determines the construction of amino acids and, thus, protein. Three nucleotides in succession, called a codon, correspond to one of twenty amino acids. As three nucleotides construct a single amino acid, three potential ways exist to read the sequence. These unique ways are often referred to as their own unique reading frames. The way these reading frames are found is illustrated in the table below:

A	T	G	C	T	T	A	C	G	T	A
M			L			T				
	C			L			R			
		A			Y			V		

Table 2.1: Reading Frames

From Table 2.1, it is shown that the randomly generated sequence "ATGCTTACGTA" can be read in three different. As these sequences are generated from DNA, they have three complementary reading frames that read in reverse order. Each way describes its own sequence of amino acids and, therefore, different protein structures. A triplet of nucleotides will always refer to the same amino acid, with only a few exceptions. This is referred to as the standard genetic code, and the way the nucleotides refer to a select amino acid is covered in the Table 2.2.

It is only within the coding regions of the DNA that the concept of a codon holds any real meaning as the rest of the genetic material never will be read in this way. However, identifying the sequences, including the reading frame, that will be coding regions, is not straightforward. Instead of identifying only these, it can be limited to sequences with the potential to be coding regions instead.

To better describe how to limit the sequences to those that can be coding regions, the specific codons of AUG, UAA, UGA, and UAG must be described. These codons hold particular importance as they note the beginning and the end of a gene. AUG is the single sequence of nucleotides that signals the beginning of the protein-coding part of a gene, while the three codons of UAA, UGA, and UAG describe the ending. They are, therefore, respectively referred to as start and stop codons.

To discard the parts of the genome that can not be coding regions, it is divided into multiple Open Reading Frames(ORF). These are the areas that hold the potential to be coding regions. The precise definition of an ORF varies between sources, according to work done by Sieber et al. [3]. Within their article, they argue that the most correct definition would be the area bounded by two possible stop codons, but they also describe other approaches. As the genetic structure of prokaryotes is simpler, defining an ORF as the region between a start codon and a stop codon works just as well. This is also the definition that the R package MicroSeq[4] uses, which is how ORFs will be found for this thesis.

While not all ORFs end up being coding regions, in fact, most will not; the ones that do will then be used by the cell to produce protein. The pipeline where DNA eventually results in protein, with RNA as a middle step, is often referred to as the central dogma of molecular biology.

Table 2.2: Standard Genetic Code Table

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

2.1.2 The Central dogma

According to M. Cobb [5], the central dogma of molecular biology was first described by Francis Crick in his 1957 lecture entitled Protein Synthesis. In the article, Cobb claims that Crick's definition of it was that "Once information has got into a protein, it can't get out again". Furthermore, Crick envisioned four movements between DNA, RNA, and Protein that he knew existed and two theoretically possible movements. The ones he knew existed were:

- From DNA to DNA(replication)
- From DNA to RNA(first part of protein synthesis
- From RNA to Protein(second part of protein synthesis)
- From RNA to RNA(viruses replicating)

In addition, he supposed the following could exist:

- From RNA to DNA(Reverse Transcriptase)
- From DNA to Protein

What was important to Crick was that there would be no movement from protein into DNA, RNA, or other Protein. According to E. Koonin [6], this has been refuted by the existence of prions.
Today, the central dogma is still used but refers to two of the original directions Crick thought of. Firstly, the movement from DNA to RNA, and secondly, the movement from RNA to protein. These are referred to as transcription and translation, respectively.

Transcription

Transcription is the first part of the process and describes how DNA is copied into its RNA equivalent. This takes place where the DNA is located, in the nucleoid in the case of prokaryotic cells. RNA, short for ribonucleic acid, is a molecule much like DNA but has a few structural differences. It is most often single-stranded, with that strand being built with a different sugar, ribose. And while it utilizes the nucleotides much the same way DNA does, it binds with uracil(U) instead of T. While the details vary between organisms, the general explanation is that the molecule RNA polymerase goes through the relevant parts of the DNA thread and creates a faithful copy of it in the form of mRNA. This includes the codons that will be read for protein synthesis and other regulatory sequences that affect the production.

Translation

Translation occurs after transcription and when the mRNA created during transcription is taken to the ribosome. The ribosome locates the start codon AUG and continues to read the sequences in triplets. As the ribosome reads the codons, the corresponding amino acids are brought to it and attached to a polypeptide chain that is gradually built up. When the ribosome encounters a stop codon, it ends the construction, and the protein described in the DNA is fully constructed.

2.1.3 Protein Families

In the same way that two species can be related to one another, so can the different proteins they produce. As protein is constructed from a sequence of nucleotides, it changes when evolutionary forces affect it.

From inserting a sequence, deleting some or changing out one nucleotide for another, the ways that DNA can change through time are many. With sufficient time, some sequences will be difficult to identify as having the same origin. Two proteins sharing a common origin means that it is likely they hold somewhat similar functions within their respective organisms.
The speed at which proteins change differs. Some that hold critical functions for the organism will remain stable for long periods of time. Maybe even small changes result in the organism's early termination long before it can reproduce. Others change quickly because they are located in regions that experience frequent mutations and do not hold these critical functions.
When presented with new genetic information, numerous analyses should be made. One of them is identifying the areas that are coding and how these coding regions relate to the organisms we already know. Many aspects of the novel organism can be inferred by identifying its proteins.

2.1.4 Taxonomy

The field of taxonomy aims to sort species into an established hierarchy of organisms. It is based on a hierarchical order of categories, called ranks, with each rank pointing to a point of evolutionary divergence. In this way organisms that are closely related to one another end up sharing many of the same categorical entries, while two very different ones might differ in all.
While the precise details of ranks can differ depending, among other things, on the desired resolution these are the most common ones.

- Domain
- Kingdom
- Phylum
- Class
- Order
- Class
- Genus
- Species

2.2 Pfam and HMMER

When presented with an unknown ORF, one area of interest is trying to identify which, if any, of already known protein families this current ORF could belong to. The most common way to store these patterns is through hidden Markov models (HMM), and databases such as Pfam stores these patterns for all the protein families they have identified. Using tools like HMMER, the unknown ORF is compared to the twenty thousand patterns stored within the database, and if any of them were to match any of the Pfam entries, the sequence in question has a much higher probability of containing a protein-coding gene.

2.2.1 Hidden Markov Models

HMMs, which form the mathematical foundation for the models found within the Pfam database, are statistical models that model the probability of a sequence of events. An HMM is a collection of states and the transitions between them. A state refers to the location within the sequence, in the case of ORFs it would be the given nucleotide or amino acid. For each state, there exists a finite number of possibilities to obtain the next element in the sequence; the probability of transitioning into another state is marked with a probability. Due to this design, the HMMs excel at explaining the complicated patterns found within sequence data. Random mutations or other insertions make the sequences in question vary drastically from each other despite originating from the same sequence originally, so having models that still recognize them as the same is critical.

2.2.2 Pfam

Pfam was first developed in the mid-1990s but has continuously developed since then. Pfam is a database that stores p-HMM for both protein families and domains. With its 36th release, Pfam now stores 20 795 entries and 659 clans. The clans refer to what other sources refer to as superfamilies, how different families cluster together. Pfam itself is divided into two databases, Pfam A and Pfam B. Pfam A contains documented entries, while Pfam B is computationally clustered together. The result is that Pfam B contains many more entries than Pfam A, but the certainty that its entries relate to parts of the DNA that is expressed is less certain.
According to the Pfam database paper [7], there exists a total of 6 different categories that Pfam entries are sorted into with Pfam Families and Domains being by far the majority.
As discussed in Section 2.1.3 a family refers to genes, and thus proteins, that are similar enough that they most likely had a common evolutionary ancestor. These constitute a majority of the Pfam database, with 11 242 different families having been identified. Domains, on the other hand, refer to smaller units of the protein. Unlike the families, where only it only makes sense that one is present, multiple pfam domains might be present in a protein at once. The Pfam domain number 6406. The other categories are much less represented and represent smaller regions, such as specific repeats.

2.2.3 HMMER

HMMER is a tool developed by Sean R. Eddy, [8], to both create HMM profiles and use known profiles, such as those stored in Pfam, to annotate new sequences. The software is detailed in its extensive user manual, which can be found on the website HMMER.org.
When HMMER is used to annotate a sequence, using the hmmscan command, it compares a sequence with the p-HMM it has access to. As it was developed in tandem with the Pfam database, the p-HMM used by the database is perfectly suited for HMMER scans. When it has identified the similarities between the sequences and the profiles it compares them to, it gives a detailed result containing, among others, where the Pfam entry was found, what it was, and how certain it is there. It is these outputs from HMMER that will serve as the basis for the true values for identifying if a Pfam entry is present anywhere in the given sequence.

2.3 Machine Learning Theory

The field of machine learning is vast and contains many different mathematical and statistical models used to predict an output from various features. From the simple linear regression model to artificial neural networks, the range of complexity displayed by the models is large. As data becomes more and more accessible, machine learning models become an ever more popular approach to exploring their patterns and extracting valuable information.

Machine learning comes in two variants, supervised and unsupervised. Unsupervised models try to obtain information from the provided data without needing it to be labeled. A natural application of unsupervised learning is discovering clusters of similarities within data, such as dividing up images of animals into approximations of species.

A supervised machine learning model needs a target variable to try to predict. Such supervised methods will be used in this thesis. By observing the selected target variable and finding out what other data points correlate with it, it can extract the data patterns that imply the value. These patterns are then remembered and can be applied to new data where it is unknown. The exact process of learning varies from model to model.

For this thesis, the machine learning models chosen were XGBoost, Artifical neural networks, convolutional neural networks, the Tsetlin machine, and the Tsetlin machine with convolutions. Below a basic overview of each is presented, with a particular focus on the tsetlin machine.

2.3.1 XGBoost

XGBoost, short for eXtreme Gradient Boosting, is an implementation of gradient boosting that uses a large ensemble of decision trees to make predictions. Developed by Tianqi Chen, its methodology and applications are detailed in the research paper [9].

The principle behind XGBoost is that it constructs multiple decision trees in sequence, each trying to correct some of the errors of its predecessor. These trees are then combined into an enabling model, where each contributes to the final prediction, be it regression or classification, The decision tree, the building block of the XGBoost model. is a simple sequence of binary choices. As each choice has choices of its own, it eventually forms a reverse tree, hence its name. The initial node in a decision tree is referred to as its root, the ones in the middle of its branches, and the final nodes, those that contain the model's prediction, are called the leaves.

2.3.2 Neural Networks

Neural networks form a subset of machine learning called Deep learning. The fundamental building block of the neural network is the artificial neuron, a construct intending to mimic the function of the human neuron. In its simplest implementation, it looks at an incoming value, compares it to a threshold, and gives a binary output depending on whether this threshold was surpassed.

A simple neuron in itself cannot provide much in terms of meaningful analysis, but when put together and organized into complex structures, they can learn complicated patterns within data and the on-linear relationships that other machine learning models struggle with. One of the greatest successes within the realm of neural networks is AlphaFold[10], developed by Google DeepMind, that were able to predict the 3-dimensional structure of proteins; another is the transformer architecture[11], also developed by Google, that has given us the Large Language Model that has brought AI models into everyday conversations.
The basic principle of a neural network is to have a predefined architecture, the way the neurons are structured, and have them iterate over large datasets. Through a process known as back-propagation, the model adjusts its weights when it encounters errors and, through iteration, is able to learn the best possible weights to explain the patterns within the dataset; for supervised methods, this involves making a prediction for a given feature.

Convolutional neural networks

Convolutional neural networks differ in that they analyze the data as an image or as a sequence. It does this by looking for specific patterns. Therefore, the convolutional network does not care

where in the data some pattern it learns to identify is located, just that it is located somewhere. The convolutional network identifies these patterns through the use of filters. A filter refers to a sub-sequence, or area, that scans over the data and gives a value depending on how the content in the data matches the values in the filter. With a perfect match, it might output 1, and where no matches are found, it might output 0. These values can then be used and put into another neural network or have more convolutional layers added so that more complex patterns can be identified.
In the case of DNA, this would mean identifying sequences such as triplets of nucleotides that form amino acids. Through this, it will be able to identify the occurrence of specific amino acids anywhere within the data. This alone will be meaningless, and not able to predict much, but by adding another layer of convolutions on top, sequences of amino acids anywhere within the DNA could provide significant information.

2.3.3 Tsetlin Machine

The Tsetlin Machine is a machine learning model developed by Ole-Christoffer Granmo[12] with the aim of providing comparable results to the more established deep learning models while providing a higher degree of interpretability and reduced memory usage.

As the artificial neuron functions as the building block for neural networks, the Tsetlin Automaton is the building block for the Tsetlin Machine. First described by the Soviet mathematician Michael Lvovitch Tsetlin in 1961, from whom it gets its name, the Tsetlin automaton is a form of Learning Automata, a mathematical model for making decisions in uncertain environments. Learning automata tries to find the optimal solution through a structured approach to trial and error, while trying to minimize the amount of resources used when finding the optimal solution so that this solution can be exploited. One classic example of the application of such automata is the multi-armed bandit, where the participant must find out which of the many slot machines gives the best payout. The balance of how much resources to invest when finding the best slot machine is one of the key questions to ask in such a system.

The Tsetlin Automaton is simple in its design. It consists of 2N stages and a reward and penalization mechanism to move between these. If the stage is lesser or equal to N it gives a negative result and if it is greater, it gives a positive one. When it makes a prediction, either correctly or not, it will be granted either a reward or penalty, depending on how its guess compares with the true value. This moves the automata's pointer in either the positive or negative direction.

	O				

Table 2.3: Tsetlin automaton

The table 2.3 shows a Tsetlin automaton with its pointer being located on the left side of the threshold, thus marking it as 0.

		O			

Table 2.4: Tsetlin automaton with reward

The table 2.4 shows a Tsetlin automaton being granted a reward thus moving the pointer towards the right. It is still not crossing the threshold, so the pointer still outputs 0.
An additional reward changes the pointer rightwards and over the threshold in 2.5. This automaton now outputs 1.

			1		

Table 2.5: Tsetlin automaton with reward, changing signs

The way the Tsetlin machine utilizes these automata is by assigning each variable and its negated equivalent their own automata. The variables and their counterparts, that contain the data the Tsetlin machine will utilize, are referred to as literals. The automaton decides which of the literals are to be included in a larger unit called a clause, with the rest being noted as excluded. This is decided by making predictions, noting which were right and which were wrong, and adjusting the automaton based on specific feedback mechanisms. If such an adjustment pushes the pointer within the Tsetlin automaton over the threshold, it goes from excluded to included, and if it goes below, the included literal will be excluded.
The clause, that contains only the included literals, represents one pattern that the Tsetlin Machine tries to recognize. If the pattern described within a clause is fulfilled, i.e. all variables or their negations that were included being present in the data, the clause will give a 1 as output, else it will return 0. Half of the clauses in the Tsetlin Machine will have their polarity reversed, meaning that a 1 as output is changed to 0 and vice versa. This allows for the capture of different kinds of patterns. The number of clauses included in an instance of the Tsetlin machine is a hyperparameter that can be tuned to make the model more applicable to the data at hand.

The types of feedback the Tsetlin machine provides to each individual automaton are called type 1 and type 2 feedback. It is through these feedback mechanisms it is decided which of the literals should or should not be included in the individual clauses. Type 1 feedback primarily aims at reducing the rate of false negatives while Type 2 concerns itself with the false positives. The balancing act between the two kinds of feedback ensures that the best-suited literals, and thus patterns, are discovered.

These kinds of feedback are not always given to the different automata but are dealt out by a probability decided through a function based on the parameter T. For type 1 feedback the probability of it being given is:

$$\frac{T - \max(-T, \min(T, f_{\Sigma}(X)))}{2T}$$

while for type 2 it is:

$$\frac{T + \max(-T, \min(T, f_{\Sigma}(X)))}{2T}$$

With $f_{\Sigma}(X)$ representing the sum of the clauses, i.e. taking the correct guesses and subtracting the wrong ones. The effect of this in practice is that if the sum exceeds the threshold T the respective probabilities the chance for feedback stabilizes at:

$$\frac{T - \max(-T, \min(T, T+1))}{2T} = \frac{T - T}{2T} = 0$$

and

$$\frac{T + \max(-T, \min(T, T+1))}{2T} = \frac{2T}{2T} = 1$$

for type 1 and type 2 feedback respectively. When the count of the votes is less than -T, these results are flipped. Everything in the middle of these T values is a gradual transition from one state to the other. These reflect the probabilities for feedback when a clause evaluates positive, for the negative it is the opposite

Type 1 feedback operates by looking at how the output of the literal, 1 or 0, relates to the output of the clause, also 1 or 0. It then decides, by a set probability, if the automaton assigned to the literal in question receives a reward or penalty or if it does not experience changes at all. The probabilities for the different outcomes are different based on whether the literal was included or not in the clause. For illustrative purposes, these values will be explained as either large or small, but they are set by the formulas $P = 1/s$ and $P = (s-1)/s$, with s being a tunable hyperparameter. As s is often a large number, $1/s$ will be small, and $(s-1)/s$ will be large; these sizes refer to the probabilities of them occurring, not the size of the reward/penalty.

Table 2.6: Type 1 feedback for included literal

	Clause = 1	**Clause = 0**
Literal = 1	Large reward	Small penalty
Literal = 0	Impossible	Small penalty

When a literal is included in the clause, it will receive a large reward if both the clause and literal are simultaneously true. The literal can never be false if the clause is true, as the clause depends on all included literals being true. If the clause is false, it will receive a small penalty, regardless of the literal value. If neither a reward nor a penalty were given, nothing would change.

Table 2.7: Type 1 feedback for excluded literal

	Clause = 1	**Clause = 0**
Literal = 1	Large penalty	Small reward
Literal = 0	Small reward	Small reward

When a literal is excluded from the clause, but the clause and the literal are both true the associated automaton will most likely receive a penalty. In all other cases, there will instead be a small chance of getting a reward. If neither a reward nor a penalty were obtained, nothing would change.

Type 2 feedback works in the same way, except it does not operate with probabilities the same way type 1 does. Instead, it either gives a penalty or not.

Table 2.8: Type 2 feedback for included literal

	Clause = 1	**Clause = 0**
literal = 1	Nothing changes	Nothing changes
Literal = 0	Impossible	Nothing changes

When a literal is included, type 2 feedback never changes any of the automata.
However, it penalizes a literal if it is excluded from a true clause. The effect of this would be to reinforce further that an excluded literal remains excluded.

Three key hyperparameters present themselves for the practical implementation of the Tsetlin machines: The number of clauses, s, and T. The number of clauses represents the number of patterns that the Tsetlin machine is able to describe. Using more clauses adds to the computational needs while increasing the risk of overfitting. However, many clauses are necessary for more complex data to capture the many interactions of features within the dataset. s represents the value used to regulate the feedback mechanisms. Choosing a high s makes the model

Table 2.9: Type 2 feedback for excluded literal

	Clause = 1	**Clause = 0**
literal = 1	Nothing changes	Nothing changes
Literal = 0	Penalty	Nothing changes

change less from the feedback received resulting in a more stable model that trains slower but less prone to overfitting. A low s has the opposite effect. T influences how likely an automaton is to receive the feedback mechanisms as the sum of clause evaluations approaches T, whose strength is decided by S.

Due to the Tsetlin machine being based on propositional logic, it is easier to understand for humans. Its collection of AND statements is possible to translate into meaningful information, unlike the large collections of weights one gets from artificial neural networks. This in part, aims to solve some of the "black box" issues one encounters from more established models and would make it possible to utilize AI in areas where this insight would be necessary or interesting. This insight will be difficult to access in the work done for this thesis, as the features in question are individual amino acids in long sequences of LORFs.

2.3.4 Weighted clauses

When employing a large number of clauses in a Tsetlin Machine, many of them may turn out to be similar. Instead of using each clause independently, similar clauses can effectively be merged through a weighting mechanism. This approach, a primary feature of the weighted Tsetlin Machine as discussed in the article by K. Darshana Abeyrathna et al. [13], allows for the assignment of weights to each clause. By doing so, the relative impact of each clause on the model's outcome can be quantified, enhancing both the efficiency and interpretability of the model. This method not only simplifies the model by reducing the number of active clauses but also focuses the learning process on the most important features.

Convolutional Tsetlin Machines

The convolutional tsetlin machine, as described by Grandmo in the paper [14] is a variation of the Tsetlin machine that treats each clause as its own convolutional filter.
By using convolutions, the convolutional Tsetlin Machine aims to be able to tackle some of the same issues that the Convolutional Neural Network is able to. The main advantage of convolutional models is their ability to find a certain pattern anywhere within the data, thus allowing the analysis of data that doesn't have fixed feature positions, such as images and sequences.
In the convolutional Tsetlin machine, each clause represents a filter; in the case of this thesis, it would represent a sequence of amino acids. Each clause also contains information in the form of coordinates so that it is able to identify where such patterns occur in the image/sequence.

2.4 Metrics

To evaluate how well different models predict from data, the field of machine learning employs different metrics. In this thesis, the metrics of Accuracy, Precision, Recall, and F1 will be employed.
The formulas presented contain the terms TP,FP,TN, and FN. These refer to:

- True positives, those who were actually true and identified as true.
- False positives, those who were actually false but identified as true.

- True negatives, those who were actually false and identified as false.
- False negatives, those who were actually true but identified as false.

2.4.1 Accuracy

Accuracy refers to the percentage of guesses that were correct. It can be found through the formula:

$$\frac{TP + TN}{TP + FP + TN + FN}$$

Accuracy is, therefore, a good estimate for the general performance of the model but it does not say anything meaningful about what the model predicted right and what it predicted wrong. With unbalanced datasets accuracy might especially be a faulty metric as the model could predict everything to be of the majority output and still get a good result.

2.4.2 Precision

Precision refers to the percentage of the true positives within the predicted positives.

$$\frac{TP}{TP + FP}$$

Precision is thus a measure of how well the model excludes false positives from its predicted positives.

2.4.3 Recall

Recall refers to the percentage of the true positives that were actually labeled as positives.

$$\frac{TP}{TP + FN}$$

Recall is, therefore, a metric that tells how well the model is able to identify the positives within the data.

2.4.4 F1

F1, also known as the F-score, is a metric that combines elements of both precision and recall.

$$\frac{2 * TP}{2 * TP + FP + FN}$$

This score then measures the balance of precision and recall, i.e., taking into consideration how well the true positives are identified and ensuring not too many false negatives are included. The f1 score is therefore much more suited for unbalanced datasets compared to accuracy as it is able to

2.5 Related Work

The master thesis written by Yva Jacob Sandvik [15] explores the potential of using machine learning algorithms to predict whether any given ORF is a coding gene. While this is not an exact one-to-one match with the discovery of Pfam domains, as will be the goal of this thesis, several of the methods used within the thesis will be just as applicable.
In the article by Liland et al., [16], this work is continued by trying to use the Tsetlin machine for the same cause. In the article, it is presented that the Tsetlin machine was able to obtain better performances when compared to simple neural networks.

Chapter 3

Method

This chapter will explore the methodology used to obtain the results necessary to answer the research question. The question is whether a machine learning model can be trained to predict if any Pfam entry is present in DNA sequences and to what degree the Tsetlin machine will be a good model to predict this. It will go through the process of organizing and processing the initial data for machine learning applications and how these machine learning applications were constructed.

3.1 The original data

The data used in this thesis was initially taken from the RefSeq database hosted by the American institution NCBI [17]. For each organism defined within the database, only one assembly is marked as the reference genome, these are curated to ensure that they maintain a high standard. Not every reference genome is marked as complete, containing whole nucleotide sequences. Those were discarded, and in the end, 4273 reference genomes were selected containing a total of 150 540 245 LORFs

This work was done by my supervisor Lars-Gusav Snipen and he describes the process as the following:

”From each genome, we extracted all LORFs using functions in the R package microseq. Only LORFs of length 90 or more were considered, giving protein sequences of more than 30 amino acids after translation. The amino acid sequences were then run through the HMMER software (hmmscan command), and using the latest version of the Pfam database. This will then detect the occurrence of any known sequence pattern described by any of the 20 000 profile hidden Markov models of the Pfam database. Any LORF containing one or more hits was then marked as a ”Positive”, i.e., containing a pattern known from Pfam. The negatives were those LORF without such hits. This job was run on the local High performing Computing Cluster, and took approximately 1 month to complete.”

3.2 Constructing the datasets

To create models that would try to predict whether a Pfam entry would be detected within a given amino acid sequence, the data would have to be sampled and converted into a machine learning-friendly format. This part of the methodology will focus on sampling from the original data to create multiple new datasets. While a larger quantity of data would be preferable due

to more than 20,000 different Pfam entries to recognize, a selection had to be made to make it computationally feasible to run. The result was the creation of different datasets with the hope that the multiple viewing angles would provide some valuable insight.
In total, 35 different datasets were constructed using three different approaches. Firstly, a general dataset that samples as spread out as possible within the original data. Secondly, five order-based datasets that sample from a specific taxonomic order. Thirdly, 25 datasets that look for specific Pfam entries.

3.2.1 The General Dataset

The first dataset, the general dataset, is constructed to include as much variance as possible. By sampling from across the entire width of the taxonomy, with an emphasis on not overrepresenting any part, there is a hope of finding out the general performance of the models. The original thought was to include one organism from each taxonomical class (see link to theory) in the original data. In total, 98 different classes were represented in the RefSeq genomes. This was increased to three to make this dataset similar in size to the order-based datasets. As some organisms are researched much more than others, this sampling based on class would help combat their overrepresentation. As not all of the taxonomic classes contained three classes, only 215 different species were selected instead of the 248 if all classes had three representations in the RefSeq data.

The target variable chosen for this dataset would be a binary value indicating any recognized Pfam entry for the given sequence. If the HMMER output for the sequence in question contained either one or multiple Pfam entries, the value would be set to 1; if none were recognized, it would be set to 0.

3.2.2 The Order-Based Datasets

The order-based datasets try to take a different approach than the general one, and instead of casting the net wide, they try to narrow the search. By selecting specific taxonomical orders and only sampling from these, the organisms would be much more related to one another. In theory, this closer relationship would result in similar Pfam entries appearing throughout the data. The orders were chosen based on their frequency within the original data. The selected orders were the following: Bacillales, Corynebacteriales, Burkholderiales, Lactobacillales, and Enterobacterales.

The order-based dataset's target value would be the same as the general dataset's, with 1 representing any detected Pfam entry and 0 representing none.

3.2.3 The Specific-Pfam Datasets

The Specific-Pfam datasets differ the most from the others in that they operate with a different target variable. Both the general and the order-based datasets employ the presence of any detected Pfam entry as their target. The specific Pfam datasets instead look for specific Pfam entries. The entries were chosen by finding the most occurring ones for sequences of lengths between 100 and 200 amino acids within the original data. Then, by looking at all the. In total, 20 different Pfam entries were chosen, as these were the ones that numbered over 20,000, which was set as a minimum threshold. These were the following:

`PF00583.27`, `PF12802.9`, `PF13508.9`, `PF13673.9`, `PF01047.24`, `PF13412.8`, `PF12840.9`, `PF13302.9`, `PF13463.8`, `PF01381.24`, `PF00903.27`, `PF08281.14`, `PF00072.26`, `PF04545.18`, `PF00440.25`, `PF01022.22`, `PF00293.30`, `PF13560.8`, `PF08445.12`, `PF00578.23`

After these were found, 20 different datasets were created named Pfam_1 through Pfam_5. The first of the found Pfam entries operated as the target variable for Pfam_1 and the twentieth for Pfam_20. As some sequences contained multiple Pfam entries, these would be included in each of the datasets they qualified for, meaning that the sequences noting positive hits could have instances of the same sequence existing in multiple datasets.
The negative values represented where these specific Pfam entries were not present but were not randomly picked out. Half of the negatives would be made from sequences where no Pfam entry was detected, and the other half were ones where a Pfam entry was detected, but not one of those included in the positive hit. These were sampled from another table, and as a result, none of the 19 other Pfam entries would be included in the negative values for a specific Pfam dataset. This represents one potential flaw of this approach, as these were the most common Pfam entries within all the data.
In addition to the 20 datasets looking at individual Pfam entries, five additional ones were constructed with the precedence of multiple, but still specific, Pfam entries being used as the target values. These cumulative sets were constructed by combining the individual datasets representing the desired Pfam entries. With the addition of more Pfam entries, the classification will have to learn to distinguish between multiple patterns that describe each entry but still contain less noise in the form of infrequent entries than the general and order-based datasets. The five cumulative datasets were chosen as the following:

Pfam_1-2, containing both Pfam 1 and Pfam 2(total 2 entries)

Pfam_1-5, containing Pfam 1 through Pfam 5(total 5 entries)

Pfam_1-10, containing Pfam 1 through Pfam 10total 10 entries)

Pfam_1-15, containing Pfam 1 through Pfam 15(total 15 entries)

Pfam_1-20, containing Pfam 1 through Pfam 20(total 20 entries)

These cumulative sets will contain repetitive sequences since some are present in multiple specific Pfam sets.

3.3 Data preprocessing

With the datasets assembled came the necessary step of preparing them for machine learning. While different machine learning techniques can utilize different inputs, most requirements are similar between the models. To the extent that it was feasible, the data was prepared the same way for all the models, with all models training and evaluating the same data.
The first step was to limit the length of the sequences so that they all had similar dimensions. Afterward, the data was balanced by selecting an equal number of sequences with positive and negative hits. This step also involved sampling out a specific number of LORFs so that, to the degree that it was possible, similar datasets were of the same length. Then came the one-hot encoding to transform the sequences from letters into binary values. The last step involved splitting the data into training and testing sets and a validation set.

3.3.1 Limiting the Length

While LORFs have already undergone a selection process regarding length, as the sequences shorter than thirty amino acids have already been cut off, further cutoffs were deemed necessary. Initially, the intention was to create different models for various length categories from the various datasets.
From observing the data, a general pattern revealed itself. Short LORFs rarely contained any Pfam entries, while the opposite was true in those much longer. Thus, a happy medium between 100 and 200 amino acids was selected. In this region, there was a much more equal distribution between the sequences that contained Pfam entries and those that did not.
By setting these limits, the number of sequences containing fewer amino acids than the maximum would allow would be fewer. The result is a model trained and tested on less sparse data than if the maximum length had been selected higher.
Limiting the max length of the sequences would also have computational advantages. Due to the one hot encoding, which will be discussed later, the sequence length would be represented by a series of zeroes and ones, twenty times as long as the sequence was in letters. A max length that was much higher would, therefore, also result in many more features for the various machine learning models to have to interpret and thus increase, among other things, the memory footprint of the models.

3.3.2 Ballancing the Data

The final act of balancing came after the data had been limited in length. While the data was much more balanced in the selected interval of 100-200 amino acids than among the LORFs in general, there was still not 50% of each,
While not strictly necessary, as most machine learning methods operate by tracking a loss function rather than optimizing for pure accuracy, having a balanced dataset would allow the models to observe more of the possible Pfam entries. With the computational requirements being the limiting factor rather than the amount of data available
The general and order-based datasets would benefit the most from training and testing on balanced data because they had to learn to recognize more than 20,000 different Pfam entries.

3.3.3 One hot encoding

With balanced datasets having been assembled, the final part of making them ready for the models involved transforming them from sequences of characters into binary values. As the genetic code can describe twenty different amino acids, this is the number of binary values one would need for each position to represent the sequences in a one-hot encoded format. With a sequence length of 200, this would mean 4000 features are present in each row of the trainable data.
An example of how one hot encoding works in practice will be presented below. To make it more intuitive, the example will be presented with an equivalent of 2 amino acids possible and a max length of 5.

Table 3.1: The sequence ABA

0	0	A	B	A

There are two main approaches when one-hot-encoding the sequence shown in Table 3.1. The one outlined below in Table 3.2 shows how it will be done in a one-dimensional format. Where the resulting binary translation is a single sequence in itself. By creating two positions for every position present in the original sequence for a total of 10 possible positions, each of those represents only the presence of one letter at one specific position.

Table 3.2: 1D one hot encoding

1		2		3		4		5	
A	B	A	B	A	B	A	B	A	B
0	0	0	0	1	0	0	1	1	0

Table 3.3 shows how it can be done in a two-dimensional format instead. With two sequences with the same length as the original. Each position represents the presence of that specific letter in the original sequence. The result is a matrix with a width the same as the original sequence but a height determined by the number of unique elements within; for the example sequence the matrix would be of size 2x5.

Table 3.3: 2D one hot encoding

A				
0	0	1	0	1
B				
0	0	0	1	0

The one hot encoding employed resulted in a dataset of 3 dimensions, using the format explained in the 3-dimensional one hot encoding example. This would be used for the models employing convolutional models. Flattening this data into the 2-dimensional approach was employed for the rest of the models. Thus, the final dimensions of the one hot encoded data would be n*200*20 for the 3-dimensional data and n*4000 for the flattened, where n represents the number of elements in the dataset (which would be between 40 000 and 200 000, depending on the dataset).

Presented with the choice of padding from either the front or the back, padding from the front was chosen. This was originally chosen so that a sequence if cut, would contain the region closest to the stop codon, as this might have a greater chance of containing Pfam domains. As only sequences less than the maximum threshold none of them were cut. By padding from the front, the end result is that the sequences become aligned at the end of the sequence.

To reduce the memory requirements of the models by making the data less sparse, one option considered involved encoding the sequences into 5-bit instead of 20-bit, as the one hot encoding would result in. A 5-bit encoding holds the potential to include 32 different values and could, therefore, represent the 20 possible kinds of amino acids. The result, however, would result in an additional layer of interpretation needed for the machine learning models, having to recognize that the amino acids were described in sets of 5 and that each position did not hold any information by itself. Instead, they had to understand the data in these blocks. Because of this, this approach was discarded, and the original 20-bit implementation was chosen.

One hot encoding assumes that the different values within one location have no clear order between them. Some amino acids are more like each other than others, partly due to the similar nucleotide coding described in the amino acids, and do not fully comply with the assumption one hot encoding makes. The result is the potential of losing some valuable insights into the data. One considered approach was to train embedding layers instead of employing one hot encoding. This would be trivial for many models, as a simple line of code would implement the embedding layers, but for the Tsetlin machine, it would prove more challenging as it requires binary inputs. The same one-hot encoding was used for all datasets to make the models' performances more comparable, as comparison was one of the primary goals.

3.4 Creating Training, Test and Validation Sets

With the dataset one hot encoded, it can already be successfully read by any of the models, with minor tweaks for individual input needs. To ensure that the performance levels observed matched the actual performance expected on previously not seen data, it was essential to ensure that there was no data leakage from the data the models were tested on into the ones they were trained on. A testing set of 20% of the samples was thus extracted from the training set, not to be touched upon again until the final performance calculations were run.
As multiple of the models trained also wanted a dedicated validation set to avoid overfitting the models, in essence making the model obtain a better result on the training data than what would be generalizable outside of them, this was also extracted from the test data at 20%. The validation models would use this non-Tsetlin set to find out for how long they should continue to iterate over the training data.
Unfortunately, due to an error in the code, the validation and test sets were mixed up, resulting in the final distribution for the training, validation, and testing sets being respectively 80%, 16%, and 4%. Having testing sets of this size, 4% of 40 000 in the worst-case scenario, might lead to the different metrics being less precise.
With datasets numbering in the hundreds of thousands, cross-validation was not chosen. Increasing the training time of the models by a factor of 5 was considered impractical, assuming a 5-fold cross-validation was employed when the total running time without it already approached 15 hours.

3.5 Creating Models

With the data finally ready to be analyzed by the machine learning models, a selection of relevant ones had to be chosen. First was the Tsetlin machine, the main object of interest for this thesis. Two variations of it were selected: the vanilla classifier and its convolutional version. The second model chosen was a non-deep learning machine learning model in the form of XGBoost. XGBoost is a model that tends to perform well on tabular data and often beats neural networks. It is a very fast model to train and test on and will also provide a good benchmarking result with which to compare.
Finally, two deep learning approaches were chosen: a neural network consisting of only dense layers, a dense neural network, and a neural network using convolutional layers, a convolutional neural network. Due to convolutional models performing well on sequence data as shown with tools such as CNN-MGP [18], a more complex model was also trained to see how much performance a bit of tweaking could result in.
The metrics chosen to compare the models were the following: accuracy, f1, precision, recall, training time, and test time. As the data were trained and tested on balanced data, looking at the accuracy should give a good picture of how well the models perform. The f1, precision, and recall will provide additional insight into what part of the prediction the model struggled with most. Finally, the training and testing time should indicate how many resources the various models used, but it may not be a perfect reflection.
Except for the Tsetlin models, the output of the models was a score between 0 and 1. With the deep learning models, this resulted from the final activation function being the sigmoid activation function. This value was binarized around the threshold of 0.5, meaning that any value equal to or above the threshold would result in a true hit for either the presence of a specific Pfam entry or a general one, depending on the dataset.

3.5.1 Tsetlin Machine

The Tsetlin machine was implemented using the TMU package developed by Ole-Christoffer Grandmo and Per-Arne Andersen [19]. Their implementations of the base model were chosen because they are the most complete ones available. For the specific model chosen, the vanilla classifier and the coalesced version were considered. From the research paper where it was introduced [20], it was suggested that the coalesced version worked better for Tsetlin Machines with the number of clauses set between 50 and 1000. As the number of clauses had to be much higher, this was not chosen.
The Tsetlin machine was one of the models that used the flattened version of the dataset. And to ensure that loading all of the data wouldn't cause memory issues a system of batching the data and provided it to the algorithm in pieces was implemented.
As for the number of clauses, the amount chosen was 10 000. Through testing, it was found that results obtained with fewer clauses were significantly worse.
The other tuned hyperparameters would be the values of T and s. The values chosen for these were based on the results of O. Tarasyuk et al. [21] who found that a reasonable estimate for the value of these hyperparameters would be $sqrt(\frac{C}{2}) + 2$ for the value of T and $2.534 * log(\frac{C}{3.7579})$ for the value of s.

3.5.2 Convolutional Tsetlin Machine

The convolutional Tsetlin Machine was constructed the same way the standard implementation was outlined above. Only two differences in the way they were set up differed. Firstly, the convolutional model utilized the data stored in 3 dimensions instead of the 2d matrix fed into the standard model. The other difference was using the parameter patch_dim, where the filter size 3x20 was chosen. This filter was designed to recognize specific sequences of three amino acids. This corresponds to the 3x20 filter used for the convolutional neural network model.

3.5.3 XGBoost

XGBoost was implemented through the XGBoost package for Python [9]. XGBoost requires the data to be inputted using its own DMatrix structure. The package contains the methods for this transformation, so this was utilized.
The model used the default parameters for the implementation, except for the training rounds being set to 1000. Early stopping was implemented with the validation set with patience of 15 rounds, meaning that 15 rounds of no reduction in the loss function would result in the process ending early.

3.5.4 Dense neural network

The dense neural network was implemented through the Python package TensorFlow using Keras [22]. It was run with the CPU, meaning no CUDA cores would be utilized in the testing or training.

Listing 3.1: Neural Network Model in Keras

```
# Define the model
model = Sequential()
model.add(Dense(128, activation='relu', input_shape=(X_train.shape[1],)))
model.add(Dense(64, activation='relu'))
model.add(Dense(1, activation='sigmoid'))

# Compile the model
```

```
model.compile(optimizer='adam', loss='binary_crossentropy',
              metrics=['accuracy'])

# Early stopping callback
early_stopping = EarlyStopping(patience=3, restore_best_weights=True)
```

3.5.5 Convolutional Neural network

The CNN models were built from the same Tensorflow package as the dense neural network outlined above[22]. It also employed the same optimizer, loss function, and early stopping implementation.
The convolutional layers employed were one-dimensional (Conv1D).
The main difference between the two models was the complexity of their architecture, with the second model focusing on trying to reduce the amount of overfitting that would occur when training the models. The simpler model was made to show what a less complex architecture, and thus more lightweight, implementation could expect in terms of performance but also to be a better comparison with the Tsetlin machine having a more comparable level of complexity.

Simple

The simple convolutional used 32 Conv1d filters with a filter size of 3 and fed them into one dense node with a sigmoid activation function. In practice, this would mean that the model would look for 32 sequences of three amino acids and observe their pattern within the data.

Listing 3.2: CNN simple with 1D Convolutions in Keras

```
# Define the model
model = Sequential()
model.add(Conv1D(32, 3, activation='relu',
                 input_shape=(X_train.shape[1], 20)))
model.add(Flatten())
model.add(Dense(1, activation='sigmoid'))

# Compile the model
model.compile(optimizer='adam',
              loss='binary_crossentropy',
              metrics=['accuracy'])

# Early stopping callback
early_stopping = EarlyStopping(patience=3, restore_best_weights=True)
```

Super

The more complex model still used the same filter size but used 64 filters instead. It also layered multiple convolutional layers upon one another to capture longer patterns. Other features, such as dropout and regularization, were added to combat overfitting.

Listing 3.3: CNN super with 1D Convolutions in Keras

```
# Define the model
model = Sequential()
```

```
# Convolutional layer with L2 regularization
model.add(Conv1D(64, 3, activation='relu', input_shape=(200, 20),
kernel_regularizer=l2(0.01)))
model.add(BatchNormalization())
model.add(MaxPooling1D(2))

# Second convolutional layer
model.add(Conv1D(128, 3, activation='relu', kernel_regularizer=l2(0.01)))
model.add(BatchNormalization())
model.add(MaxPooling1D(2))

# Third convolutional layer
model.add(Conv1D(256, 3, activation='relu', kernel_regularizer=l2(0.01)))
model.add(BatchNormalization())
model.add(MaxPooling1D(2))

# Flatten and Dense layers with dropout and L2 regularization
model.add(Flatten())
model.add(Dense(256, activation='relu', kernel_regularizer=l2(0.01)))
model.add(Dropout(0.5))  # Increased dropout to prevent overfitting
model.add(Dense(1, activation='sigmoid'))

# Compile the model
model.compile(optimizer=Adam(learning_rate=0.0001), loss='binary_crossentropy',

early_stopping = EarlyStopping(patience=15, restore_best_weights=True)
```

3.6 Code

The code for this project was developed in Python and R and executed using a Bash script. The code can be accessed at the GitHub (https://github.com/Pinglepi/Master). The datasets were generated using RStudio, which looks at the raw data outlined in this chapter's first part. These datasets have been stored locally along with their respective target values and some accompanying metadata, such as GC content and length of sequence.
To manage memory efficiently, the Bash script was designed to iterate through each dataset individually since loading them simultaneously in a single Python script would lead to memory issues. Each model was trained using the same split for training, testing, and validation, ensuring a fair comparison between the models.
When a model was finished training, it would be tested on the test set, and the results would be saved in a CSV file. A file for the prediction results was saved with the actual values of the target value, the predicted values, and the models relevant to the probability values. In addition, the metrics of accuracy, f1, precision, recall, training time, and testing time were saved in a separate file. These were run through with the visualize python file to produce the illustrations in the result sections.

Chapter 4

Results

This section details the performance of six distinct machine learning models applied to 31 datasets containing LORFs linked to Pfam domains. Results are detailed through comparisons of models, showcasing metrics such as accuracy, precision, and recall. Given the dual objectives of this thesis, which focus on evaluating the Tsetlin machine relative to other models and assessing the predictive power of machine learning models trying to identify Pfam domains, this section primarily examines the best-performing model alongside the Tsetlin machine.
Multiple datasets were employed in this analysis, including a general dataset that samples LORFs from as diverse a set of organisms as possible. Five datasets were specifically sampled based on a more taxonomy, and 20 focused on individual Pfam domains. Additionally, five cumulative datasets were analyzed to explore the effects of targeting multiple Pfam domains and how additions impacted performance. The machine learning models employed in this study are:

- Tsetlin machine
- Convolutional Tsetlin machine
- XGBoost
- Dense Neural Network
- Convolutional Neural Network(Simple)
- Convolutional Neural Network(Super)

Refer to the 'Models' section in the Methodology chapter for a more detailed description of how these models were constructed.

4.1 General

The general dataset consisted of 200 000 LORFS sampled from 3 species, if possible, for each taxonomic class represented in the original data. As there were 98 classes represented in the data, this would result in sampling for a total of 248 different organisms; as there were some underrepresented classes, only 215 were selected. Of these sampled LORFs, half represented LORFs where a Pfam domain was detected, and the other half where a Pfam domain was not. For more information on how the general dataset was created, look at subsection 3.2.1, where this was described in more detail.

Table 4.1: Performance on the General Dataset

Model	Accuracy	F1	Precision	Recall	Training Time(s)	Testing Time(s)
TM	0.704	0.715	0.690	0.741	370.675	3.973
TM conv	0.576	0.512	0.604	0.445	270.199	11.691
DNN	0.766	0.741	0.831	0.668	24.937	0.207
XGBoost	0.709	0.702	0.719	0.687	8.452	0.010
CNN simple	0.755	0.744	0.780	0.710	64.206	0.199
CNN super	0.772	0.730	0.897	0.616	918.068	0.518

Looking at the performance of the models on the general dataset, the models appear to perform in 3 distinct tiers of performance, if one primarily looks at accuracy.
First off, there are the neural network models, consisting of the dense neural networks and the two convolutional. These all achieved similar levels of performance with accuracies and f1 scores within 2% points of each other. While having similar scores, their distribution of correct guesses varied.
The convolutional neural network had two different models, one named simple and one as super. While the super network had the highest accuracy, the simple one had the highest f1. The reason for this is that the accuracy of the complex CNN was achieved by its higher precision, i.e., having fewer false positives, which was much higher than the other models. However, when it came to recall, the complex CNN scored among the worst, suggesting the model was too strict when identifying Pfam domains. Therefore, the simple CNN achieved a better f1 score, a better compromise between precision and recall.
The dense neural network achieved a performance that lies between the two convolutional networks, but the differences are so small then becomes so small that any comparison becomes close to meaningless. Simply supplying a new random seed could change the order of these.
Secondly there's the tier of XGBoost and the Tsetlin Machine(TM). These models obtained accuracies of around 70%. While similar in performance, these models had very different values regarding training and testing time. XGBosst was the fastest model trained, needing only 8 seconds, while the TM needed 371 seconds.
Interestingly the Tsetlin machine was able to obtain the highest recall score of any model, but its precision score was lacking. This suggests the model was a bit overzealous when trying to identify the presence of Pfam domains within the sequences.
Lastly comes the TM with convolutions, performing by far the worst of any model. With an accuracy just above 50%, it is barely better at identifying Pfam-containing sequences than randomly guessing.
The TM with convolution's inability to settle on a stable model was not shown in the results but was identified during training. For each epoch trained, the result was seemingly random, sometimes achieving slightly subpar performances whilst others were close to random guessing.

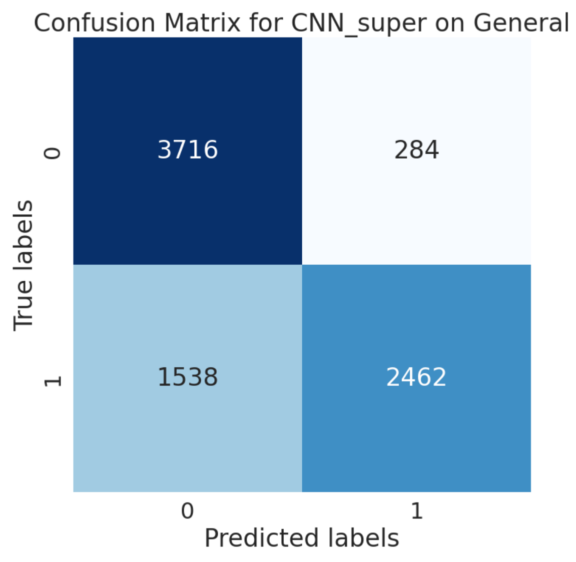

Figure 4.1: CNN super Results on General

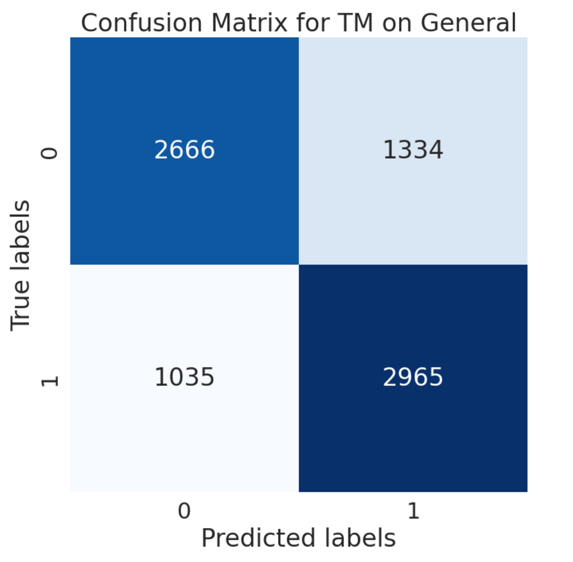

Figure 4.2: Tsetlin Machine Results on General

From looking at the confusion matrices, the results discussed under the general dataset table 4.1 are visualized. The results from the CNN super 4.1 show a model that underestimates the number of true labels within the dataset but guesses correctly often when it predicts 1. On the other hand, the TM matrix 4.2 shows a model that misses more but has a more even spread between false positives and false negatives.

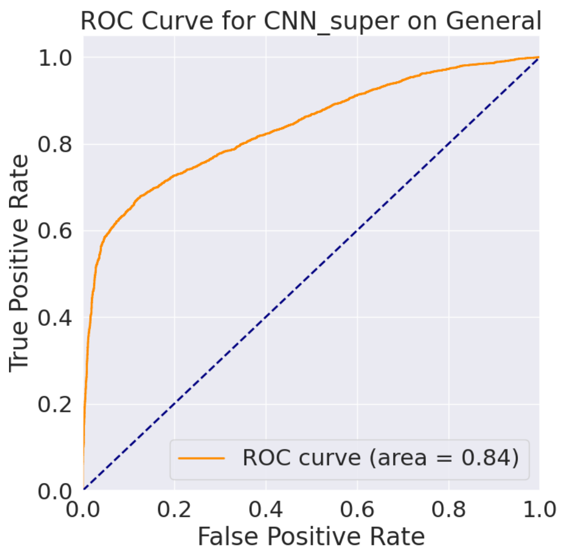

Figure 4.3: CNN super on General

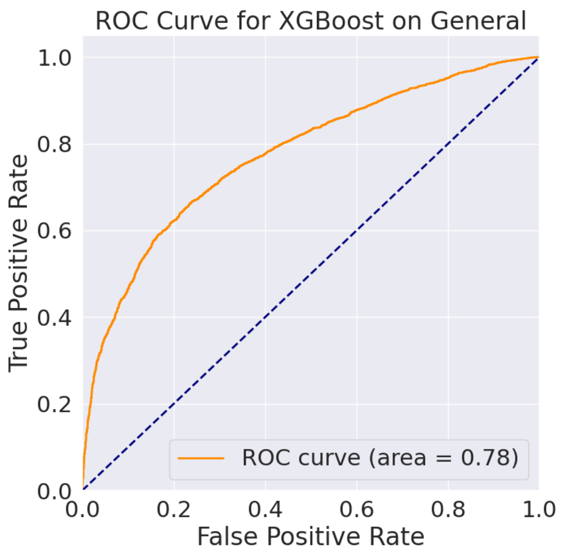

Figure 4.4: XGBoost on General

Comparing the roc curve for CNN with the one of XGBoost, a model performed in a tier below shows the model's differences in ensuring predictions separate between true and false positives. In particular, CNN 4.3 shows the ability to predict 60% of the true positives with ease but that it struggles much more with the remaining 40, having to include a large number of false positives to catch the true ones.
XGBoost, on the other hand, has a much more even distribution of predictions it struggles with, seemingly only being able to identify 30% with the same certainty as the complex CNN model.

However, it does catch up, so when including 40% false positives, the models perform about equally.

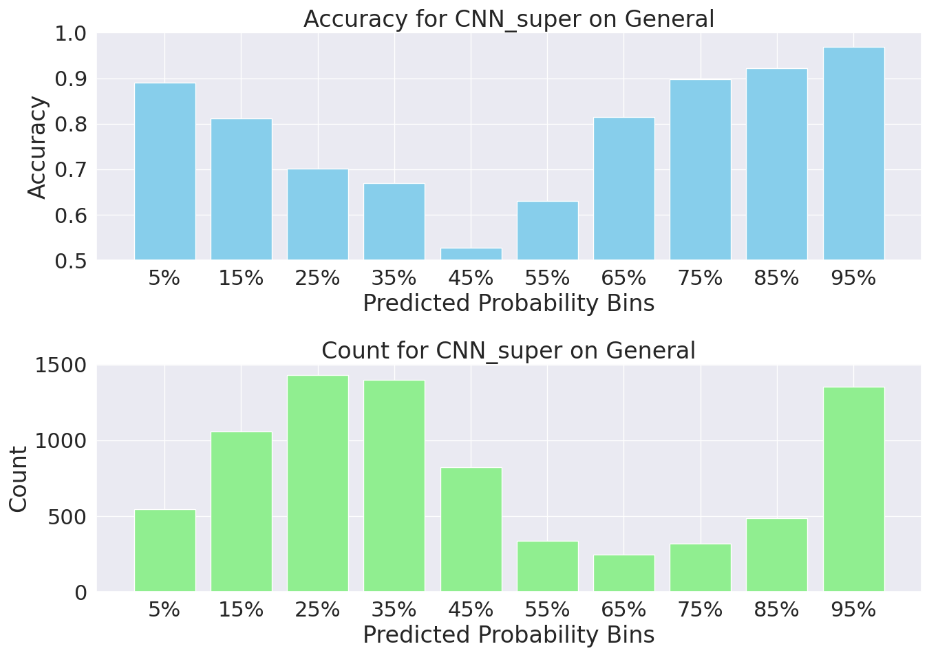

Figure 4.5: Probabilitity Distribution for CNN Super on the Gerneral dataset

The CNN model shown in Figure 4.5 shows a distribution of accuracies that one would expect. The probabilities are the output from the machine learning algorithms that gives a percentage for how likely it deems the sequence in question to contain a Pfam entry. It is the most accurate when making certain predictions, i.e. those from 0-10% a and 90-100%, while the probabilities around 50% trends towards 50%. For this plot and those similar, the cutoff for accuracy is set at 50%, as the results above these would be below random guessing, which would not make sense.

As for the most common probabilities obtained from the CNN, it seems the positive tends towards the 90-100 percentile while the negative guesses are a bit more uncertain obtaining a peak in the 20-30th percentile. This correlates well with the results from the Roc curve in Figure 4.3, which showed that 60% or so of the sequences were trivial to identify.

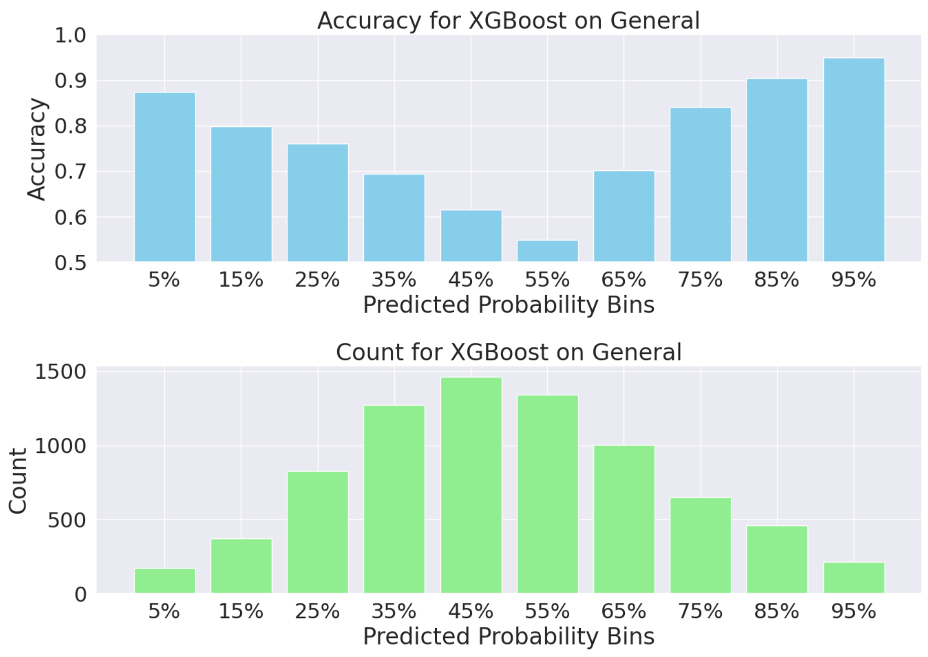

Figure 4.6: Probabilitity Distribution for XGBoost on the Gerneral dataset

The results for the XGBoost, shown in Figure 4.4, displayed a similar distribution when it came to accuracies to the CNN model but showed a significant difference regarding the distributions of the probabilities themselves. These seemed to peak in the middle, around 50%, and something resembling a normal distribution.

4.2 Order

For the order-based datasets 200, 000 LORFs were sampled, when possible, for each of the 5 chosen taxonomic orders of species. 100 000 of these were those that identified a Pfam domain within the sequence indicated by having a target value of 1 and the rest being marked with 0. The five chosen orders were Bacillales, Corynebacteriales, Burkholderiales, Lactobacillales, and Enterobacterales. The number of species analyzed were respectively 248, 229, 221, 191, and 192.

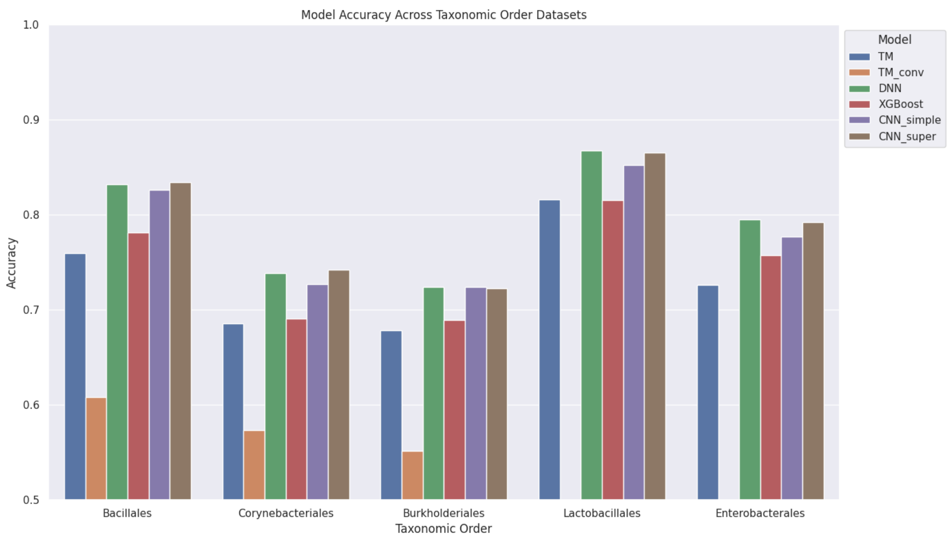

Figure 4.7: Accuracies of different models on the order-based datasets

The orders did not show a significantly higher performance compared to the general set despite the more closely related samples that were used to construct the datasets.
Interestingly the performance between the datasets varied quite a bit between each order, suggesting that some orders were more difficult to create good models for than others.
The general ordering of the models was similar across the orders, with the exception of the dense neural network being the best at predicting within the Lactobacillales order. Lactobacillales was also the order where the most accurate classification levels were possible.

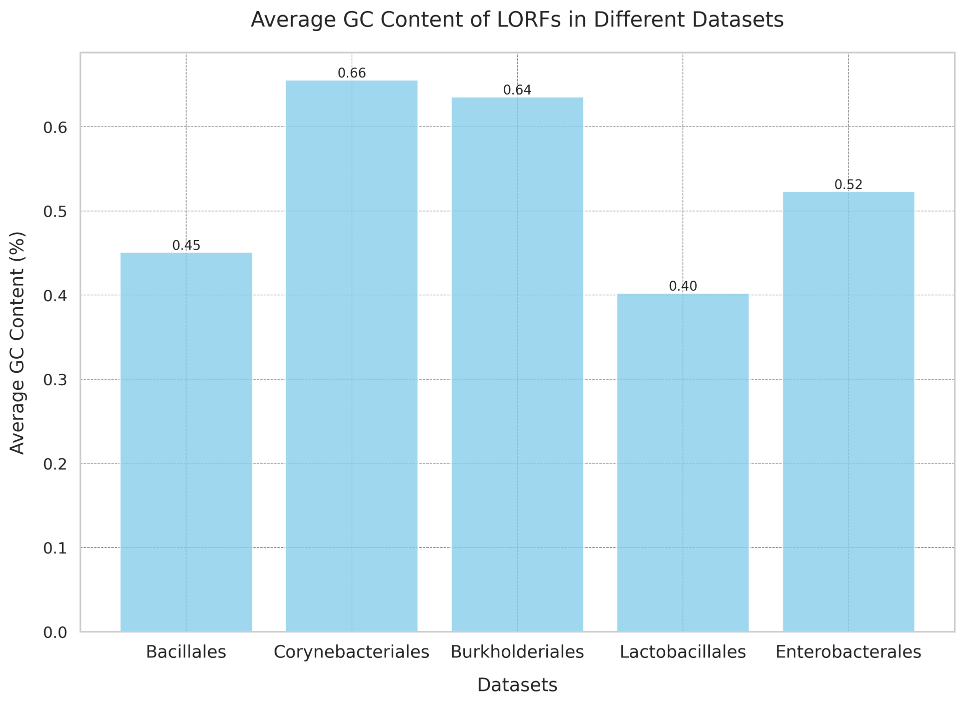

Figure 4.8: GC Content of Ordered Datasets

Figure 4.8 shows the GC content of the five different ordered datasets. Overlapping it with the previous figure 4.7 shows a potential pattern where the datasets with higher GC are also more difficult to use for prediction. Ordering the datasets by accuracy generates the same sequence of datasets as ordering them by GC content.

4.2.1 Corynebacteriales and Lactobacillales

Selecting the datasets with both the highest and lowest GC content, as noted in fig 4.8, might result in some interesting insights into how this affects model prediction power.
In addition to representing the outer bounds of GC content, the datasets show a similar pattern when it comes to performance following the accuracy metric. Lactobacillales shows the highest accuracy, whilst Corynebacteriales show the second worst. This is shown in figure 4.7

Sequence lengths

Before looking at the performance metrics in more detail, a short showcase of the lengths of sequences found within each dataset, as well as the ratio of positive Pfam hits will be shown.

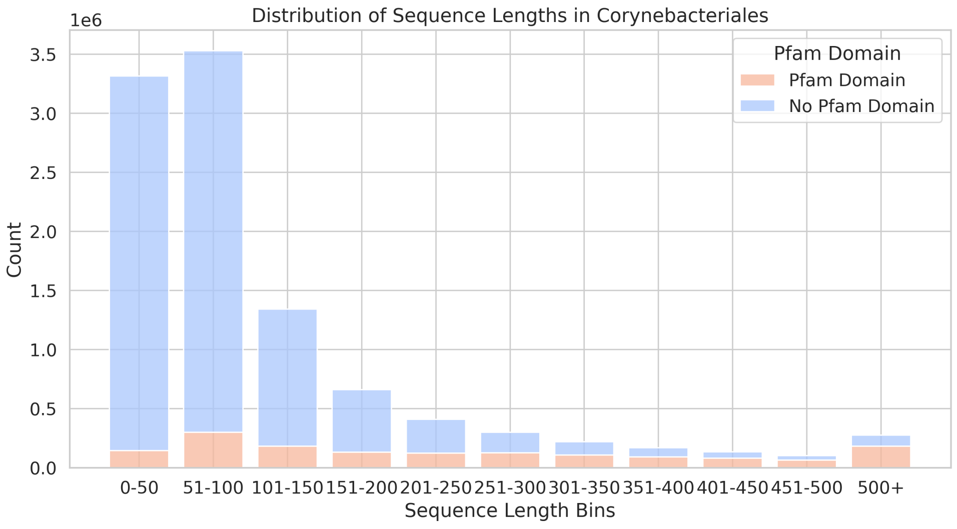

Figure 4.9: Plot showing the length of LORFs within the Corynebacteriales dataset together with the portion of results that results in a positive Pfam hit.

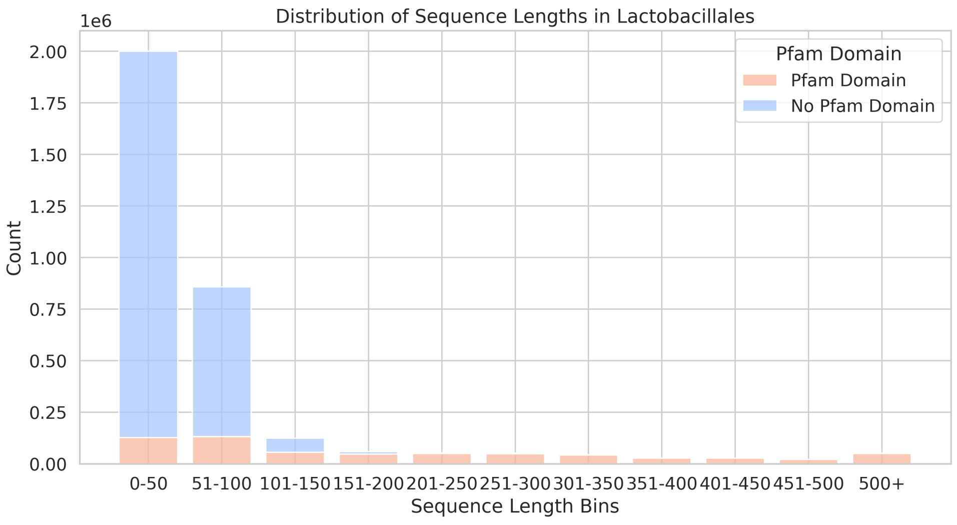

Figure 4.10: Plot showing the length of LORFs within the Lactobacillales dataset together with the portion of results that results in a positive Pfam hit.

The figures 4.9 and 4.10 show that the distributions between the lengths of the LORFs within the dataset differ significantly. In particular, while sequences of lengths between 0 and 100 constitute the clear majority for both datasets, in Lactobacillales, very few exist outside of this range.

Performance

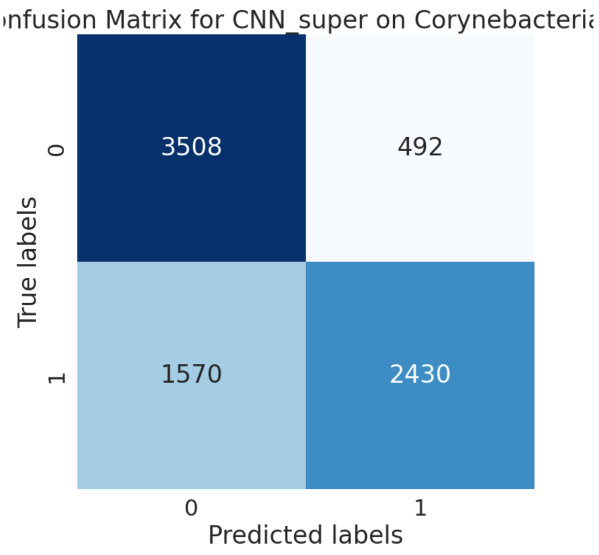

Figure 4.11: CNN super on Corynebacteriales

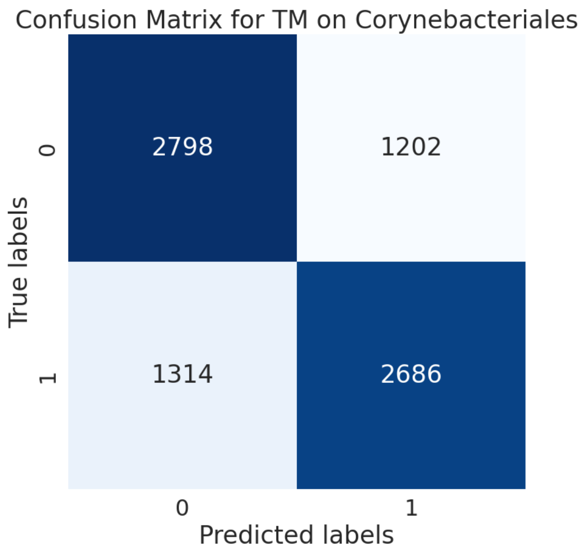

Figure 4.12: TM on Corynebacteriales

The same trend that was observed on the general set 4.1 continues on the ordered ones. The tuned CNN model 4.11 predicts with higher accuracy, meaning a higher percentage of predictions are correct than the TM model 4.12. However, this is carried by the CNN's ability to recognize the false negatives as the TM better differentiates between true and false positives.

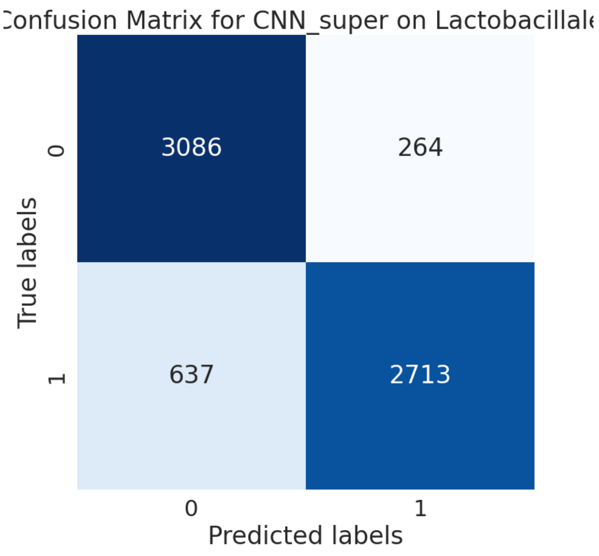

Figure 4.13: CNN super on Lactobacillales

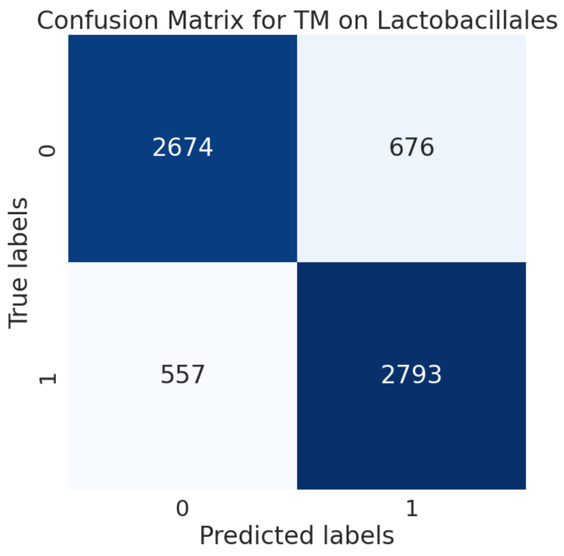

Figure 4.14: TM on Lactobacillales

Interestingly, both models show an even increase in performance when compared with the Corynebacteriales matrices, with the number of false negatives and false positives being about.

It also becomes apparent that Lactobacillales contain fewer samples than Corynebacteriales as the sums of the rows of the confusion matrices differ between the datasets.

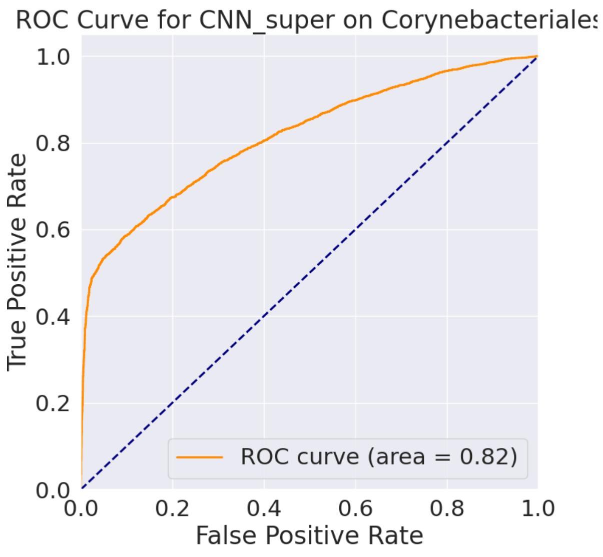

Figure 4.15: CNN super on Corynebacteriales

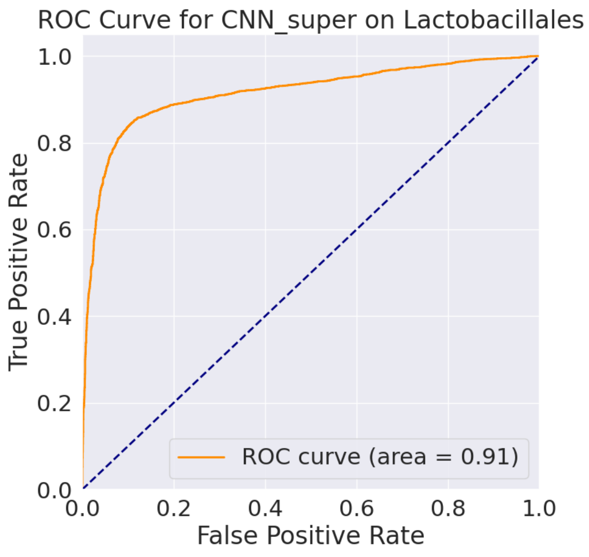

Figure 4.16: CNN super on Lactobacillales

The differences in ease of prediction become even more apparent when looking at the roc curves. Within the order of Corynebacteriales 4.15, already from the point of including around 50% of the true positives, any increase would come at the cost of a large increase in false negatives. This does not occur within Lactobacilalles before around 85% of the true positives are present, as shown in 4.16.

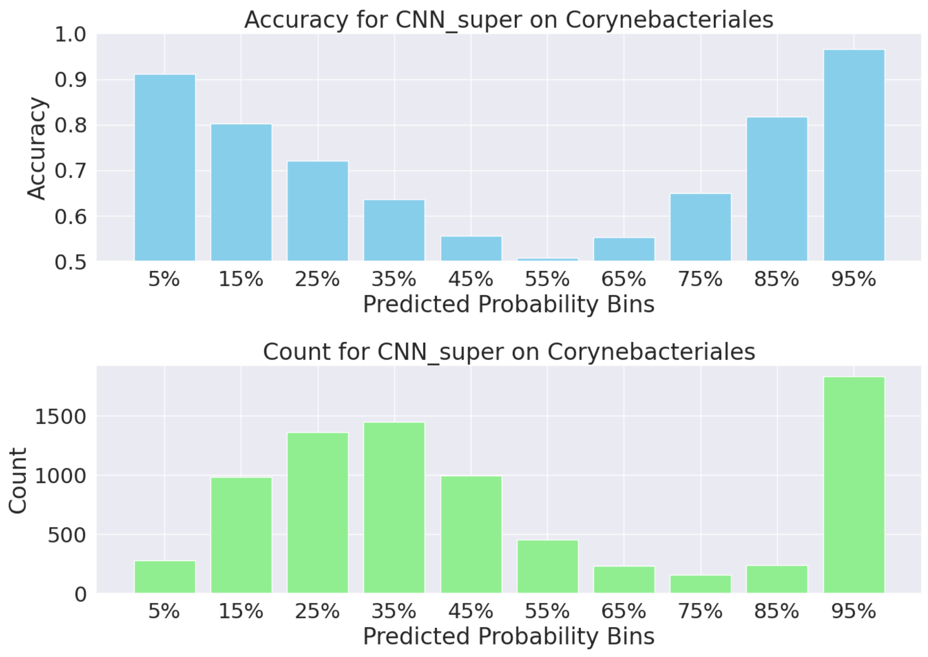

Figure 4.17: The accuracies of different model probabilities as well as the distribution of model probabilities for CNN Super on the Corynebacteriales dataset

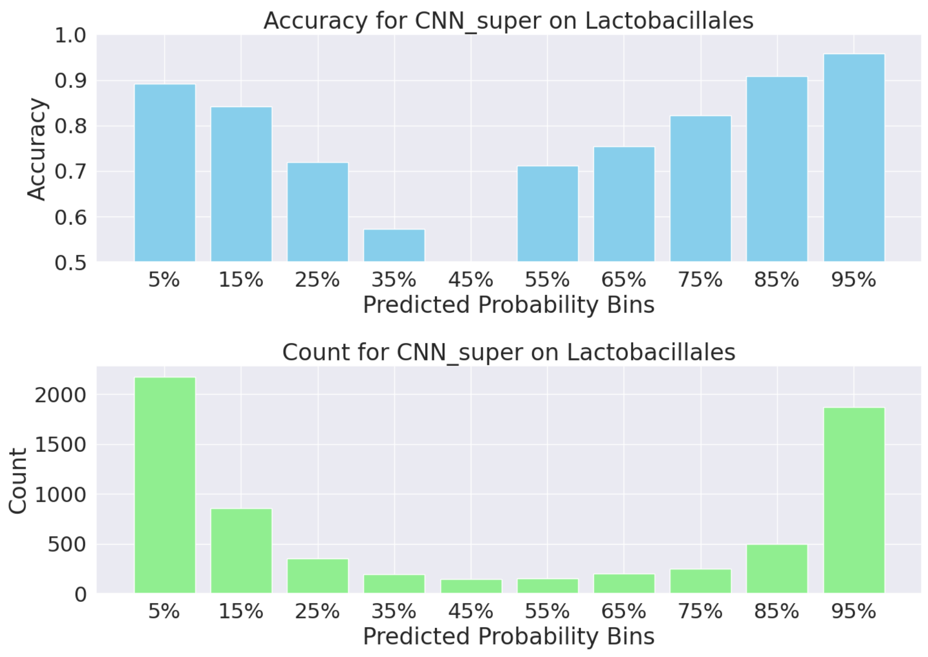

Figure 4.18: The accuracies of different model probabilities as well as the distribution of model probabilities for CNN Super on the Lactobacillales dataset

It seems that one of the main differences between the probability distributions shown in Figure 4.17 and Figure 4.18 is that the model for Lactobacillales is much more often certain of LORFs not containing any pfam entries. Interestingly the curve for Corynebactillales mirrored the one found for the general dataset, while the Lactobacillales one differed.

4.2.2 Summary Order-based

The order-based datasets showed that the prediction power varied greatly depending on what taxonomical branches were studied. Some datasets created models that performed better than the general dataset, and others performed worse.

The neural network models displayed a similar level of accuracy relative to one another across all the different datasets with XGBoost and the Tsetlin Machine following after. The convolutional Tsetlin machine were at the level of randomly guessing for a few of the datasets.

4.3 Specific Pfam Domains

This section will show the results from the models trained on the Specific Pfam datasets. The structure of these datasets is described in greater detail in the methodology chapter 3

To recap, there exist 20 different Pfam datasets and 5 cumulative sets. These twenty datasets each use the presence of a unique Pfam domain as its target variable. As the Pfam domain names are somewhat cumbersome, the datasets have instead been given numbers with names ranging from Pfam 1 to Pfam 20. the cumulative sets show intervals instead of integers, e.g., Pfam 1-5.

4.3.1 Distrubution of Pfam domains

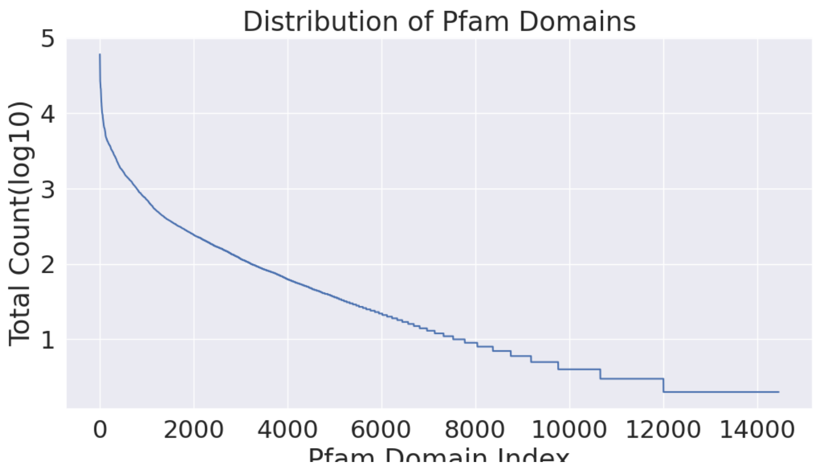

Figure 4.19: Log transformed counts of the Pfam domains in dataset

From the HMMER search on the RefSeq dataset, limited to those between 100 and 200 sequences, all found Pfam families and domains were identified. In Figure 4.19, their distribution was visualized. 14462 different entries were identified, but their prevalence varied widely. The most common Pfam entry, PF00583.27, was identified in 60 502 LORFs, while about half of the recognized domains were found in the single digits.

For the 20 most common ones Table 4.2 shows what kind of entry they are as well as the clan they belong to. The information was found for each of them through the UniProts webpage [23].

Shown in Table 4.2, the selected Pfam entries contained some variation but some Clans were much more represented than others. Particularly Acetyltrans and HTH clans seemed to be common.

Table 4.2: Overview of Pfam entries for Pfam 1-20

Pfam ID	Pfam Type	Clan
PF00583.27	Family	Acetyltrans
PF12802.9	Family	HTH
PF13508.9	Domain	Acetyltrans
PF13673.9	Domain	Acetyltrans
PF01047.24	Domain	HTH
PF13412.8	Domain	HTH
PF12840.9	Domain	HTH
PF13302.9	Domain	Acetyltrans
PF13463.8	Domain	HTH
PF01381.24	Domain	HTH
PF00903.27	Domain	Glyoxalase
PF08281.14	Domain	HTH
PF00072.26	Domain	CheY
PF04545.18	Domain	HTH
PF00440.25	Domain	HTH
PF01022.22	Domain	HTH
PF00293.30	Domain	NUDIX
PF13560.8	Domain	HTH
PF08445.12	Family	Acetyltrans
PF00578.23	Domain	Thioredoxin

4.3.2 One Pfam domain

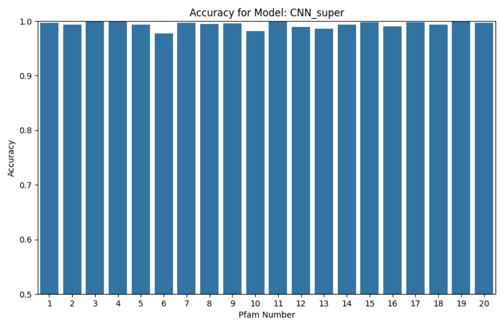

Figure 4.20: CNN super accuracies on the 20 different specific Pfam datasets. Lower bound at 50% accuracy

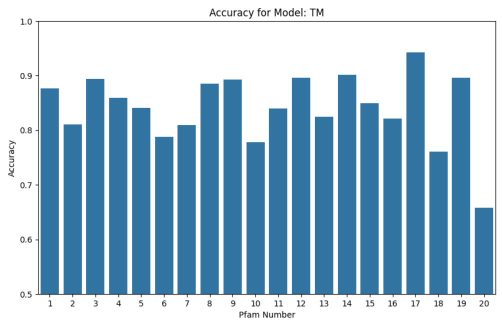

Figure 4.21: Tsetlin machine accuracies on the 20 different specific Pfam datasets. Lower bound at 50% accuracy

From the tables 4.20 and 4.21, it becomes obvious that the identification of specific Pfam domains is something the Tsetlin machine struggles with, especially compared to the tuned CNN, which excels.

The tuned CNN is able to identify all the selected 20 Pfam domains nearly perfectly, with most datasets only having a few misclassified sequences. Only a few datasets show models with performance outside the 99% accuracy range, and these are all still in the high 90s.

In comparison, the TM shows more varied performance with accuracies ranging from mid-60s to low 90s. Interestingly the accuracy in detecting the Pfam domain numbered 20 is lower than the TM obtained with the general model.

Pfam 1

The Pfam 1 dataset contained 40 000 entries sampled from the entirety of the original RefSeq data. Half of these entries represented sequences where the Pfam family PF00583 was located. The other half was constructed from 50% LORFs not containing any Pfam entry and 50% LORFs containing some kind of Pfam entry that is not the target.

Table 4.3: Performance on the Pfam1 dataset

Model	Accuracy	F1	Precision	Recall	Training Time(s)	Testing Time(s)
TM	0.876	0.882	0.841	0.927	68.564	0.751
TM conv	0.556	0.692	0.530	0.998	52.798	2.068
DNN	0.929	0.929	0.923	0.935	6.040	0.081
XGBoost	0.925	0.925	0.924	0.926	4.340	0.004
CNN simple	0.962	0.962	0.957	0.968	31.448	0.077
CNN super	0.997	0.997	1.000	0.994	873.357	0.194

While the tuned CNN has had by far the largest training time in all the datasets, the Specific Pfam datasets, such as Pfam 1, have it even higher. With training times, 3 orders of magnitude are longer than the fastest models. While not observable in the data itself, this was because the model had continuous improvements, so the model didn't stop training until it had finished the maximum of 100 epochs.

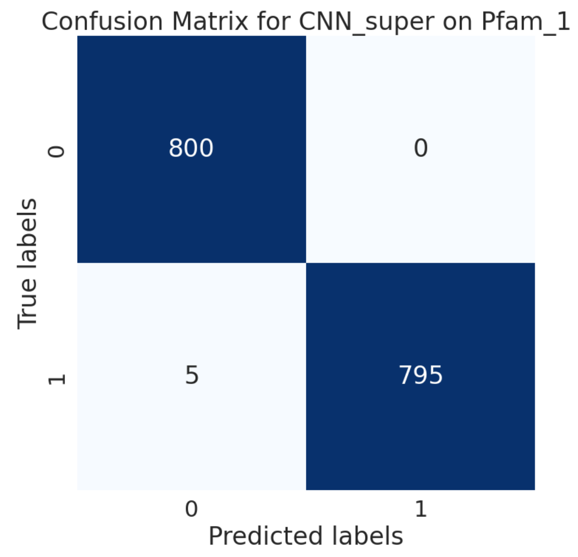

Figure 4.22: CNN super on Pfam 1

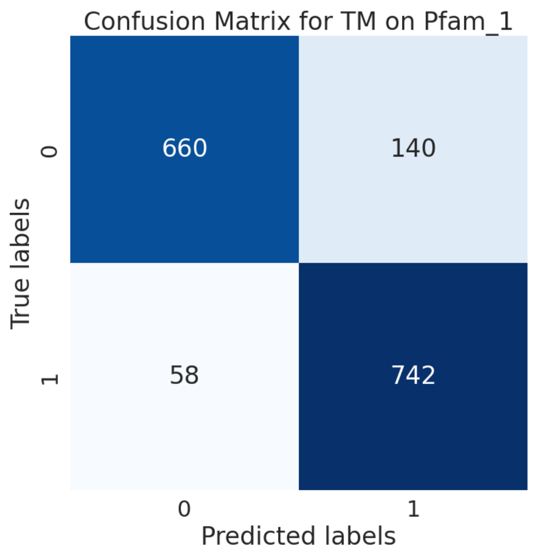

Figure 4.23: TM on Pfam 1

4.3.3 Multiple Pfam Domains

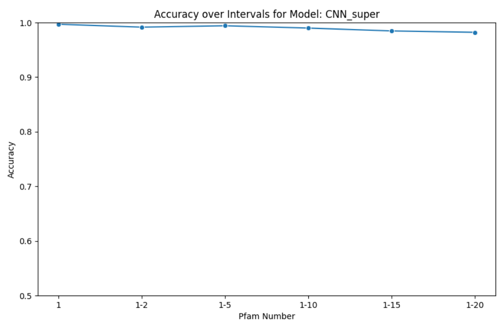

Figure 4.24: CNN super on multiple Pfam

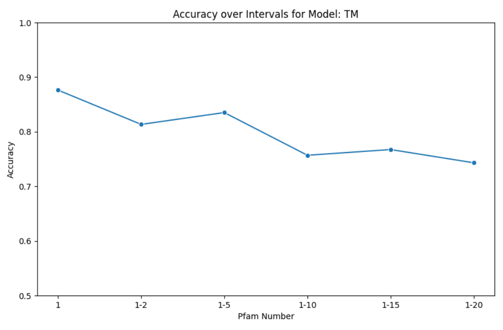

Figure 4.25: TM on multiple Pfam

When adding additional Pfam domains to the datasets, so that multiple Pfam domains decide the target variable, the gap between the performance of the two models continues to widen. As shown in the figure 4.24 the CNN contains similar results for everything between 1 and 20 Pfam domains present. The TM, as shown in figure 4.25, however, shows a declining performance with the additional added domains.

4.3.4 Pfam 1-20

To take a closer look at the results of where the performance of the Tsetlin machine and the other models are the widest, the tabular data for the dataset containing 20 Pfam domains as positive targets will be presented.

Table 4.4: Performance of models on the Pfam 1-20 dataset

Model	Accuracy	F1	Precision	Recall	Training Time(s)	Testing Time(s)
TM	0.743	0.758	0.716	0.806	368.099	4.079
TM conv	0.507	0.669	0.504	0.998	266.088	10.007
DNN	0.882	0.885	0.866	0.904	26.125	0.207
XGBoost	0.856	0.857	0.852	0.862	30.955	0.013
CNN simple	0.890	0.891	0.880	0.903	205.262	0.210
CNN super	0.982	0.982	0.980	0.984	4521.854	0.508

Combining all of the 20 detected Pfam domains resulted in a worse performance overall for all datasets compared to when just trying to single out one. The tuned CNN experienced a less significant decrease than the other models in that it retained an accuracy in the high 90s, with an accuracy of 98.

The other, non-Tsetlin, models achieved accuracy in the mid to high 80s, a significant reduction compared to their results in Table 4.3 for one Pfam domain.

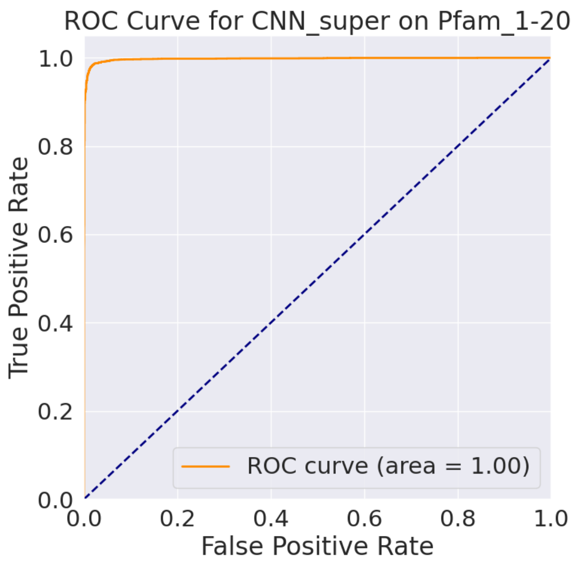

Figure 4.26: Roc Curve for CNN Super on Pfam 1-20

While it is a bit difficult to see from the resolution in figure 4.26, not many false positives would have to be included for close to every true positive to be included.

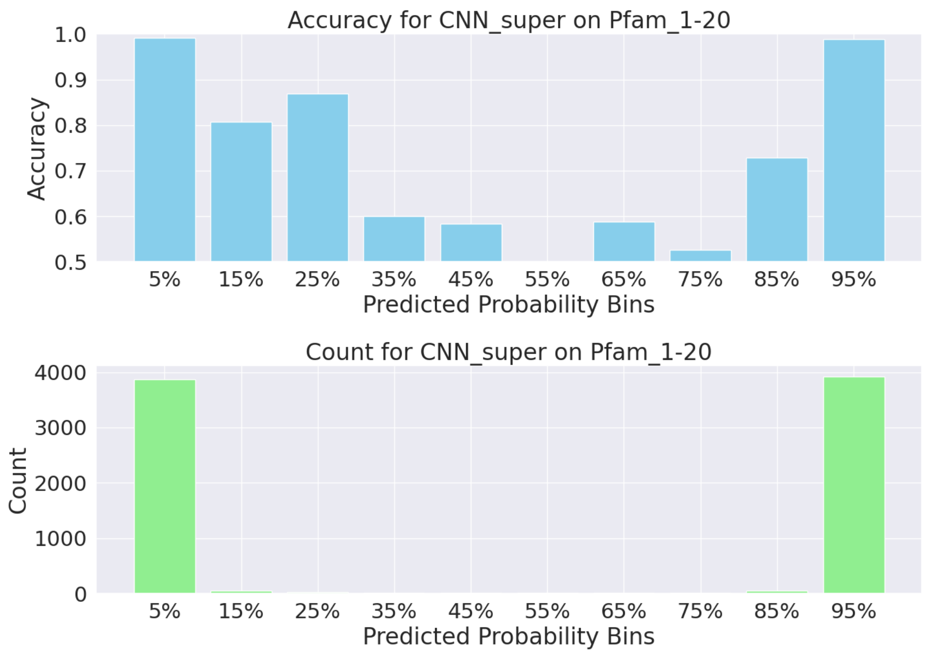

Figure 4.27: Probability Bins for CNN Super on Pfam 1-20

When it comes to the distribution of the predicted probabilities, the tuned CNN seemed to make certain predictions with almost all of the predictions being in the 0-10% and 90-100%.

4.3.5 Results of epochs for TM

As presented in the methodology section 3, the TM was only run using 1 epoch. A secondary script was run to see how epochs affected the 2 models. These are their results for the Pfam 1-20 dataset.

Table 4.5: Performance Metrics by number of Epochs for CTM

Epoch	F1 Score	Accuracy
1	0.10	0.52
2	0.43	0.58
5	0.70	0.60
10	0.70	0.59
15	0.69	0.64
20	0.08	0.51

The Convolutional Tsetlin Machine never seemed to stabilize around any value, with the obtained accuracies and f1 scores fluctuating between epochs.

Table 4.6: Performance Metrics by number of Epochs for TM

Epoch	F1 Score	Accuracy
1	0.75	0.76
2	0.77	0.77
5	0.77	0.77
10	0.76	0.76
15	0.76	0.76
20	0.78	0.77

The standard Tsetlin machine however seemed to see a slight gain from running multiple epochs.

4.3.6 Summary specific Pfam

Convolutional models excelled at detecting specific Pfam domains/families, with the tuned model in particular reaching near-perfect results. When increasing the variation of the data by inserting more Pfam entries to the target variable, the performance went down, with the tuned CNN showing the least reduction.

It was also shown that the TM might have reached better performance if it was given more epochs for each iteration. However, this did not seem to be the case for the convolutional model.

4.4 Summary results

Having looked at the different datasets has resulted in some insight into how the selected machine learning models operate on sequence data.

In general, when the accuracies of the models are concerned, the neural networks perform the best on all variants of the data. For the datasets that look for the presence of any Pfam domain, the general dataset, and the order-based ones, there is no clear ranking between which one performs the best, with both the simple CNN, the more complex CNN, and the DNN performing at similar levels. However, the complex CNN models achieved significantly better results in detecting specific Pfam domains than any other models.

This higher performance came at a higher computational cost, with the tuned CNN model, in particular, showing by far the highest training time across all the datasets. The Tsetlin machines followed after, even when using CUDA cores something the other models did not. Then came the other neural networks, but by far, the fasting training came from the XGBoost models.

The interesting takeaway from observing results from the order-based datasets is that there are significant differences in the difficulty of prediction power based on where the data is gathered from in the tree of life. One seemingly correlating variable is the GC content of genomes.

When it comes to accuracies the Tsetlin machine was outcompeted by simple neural networks in every tested scenario.

Chapter 5

Discussion

In this chapter, I will go back to the research question initially presented in the introduction of this thesis and try to answer the two main questions: Would it be possible to create a machine-learning model that can calculate the probability of including a Pfam domain within any given LORF? And would the Tsetlin machine(TM) be a good fit for creating such a model?
With the results presented in chapter 4, both of these questions should be answerable to a degree. To do this, I will first go through the results, trying to see if there is a pattern in how the models predict on the different datasets, and further discuss how the various models performed. From this, I will decide what models will be carried forward.
Second, I will go through the performance of the Tsetlin machine in more detail, trying to understand why it performed as it did and what could be done to make it better for the datasets at hand. This discussion will end with a decision on the degree to which the Tsetlin machine is a good choice when predicting LORF content from amino acid sequences.
Thirdly, I will propose a way to build a model to detect the presence of any Pfam domain. To create this final proposal, I will go through some potential approaches and discuss the advantages and disadvantages of doing it one way or another.
Finally, I will discuss a set of ideas I find worthy of discussion but that doesn't necessarily fit into the overall structure of this discussion section. This will include some reflections on how this project was carried out, as well as some potential explorations of both the data and the models that could lead to some new valuable insights that I didn't have the opportunity to implement. I will also briefly discuss the use of AI in writing this thesis.

5.1 Model results analysis

The results presented in the result chapter 4 presented multiple models for various datasets, with performances that varied significantly between model and dataset. These differences in performance will provide the foundation for the analysis in both deciding what base model is the most suited for creating the desired filtering software and what such software would have to consider to be able to predict with a high degree of precision and recall.
The balance of recall and precision stands out as the most important of the metrics when creating such a filtering mechanism. A high precision would represent the strength of the filtering itself, with a high Precision noting that the filtering could remove the sequences not containing any Pfam entries.

While high precision is desirable, a high recall is essential as msising potential hits due to overzealous filtering would be unfortunate. The primary way to think of model performance is to ensure as high recall as possible, ensuring that few, preferably close to none, of the potential positive hits are missed while aiming for as high precision as possible as a secondary goal. It is worth noting, however, that the importance of these metrics by themselves does not necessarily mean that just looking them up in the results tables, such as the one for the general dataset as presented in Table 4.1, answers what model is the most suited. For most models, the output of running the model on any given data is a number between 0 and 1; this value must be binarized to make a prediction. In all tables presented in this thesis, this threshold has been set to 0.5. This is, however, not a must; this value can be set as anything, and changing this number is one of the primary ways one can manipulate the relationship between precision and recall.

This balance is better explored through plots such as the roc curve; see 4.3 for an example. The roc curves show the relationship between precision and recall. A way to think of it is that the upper X-axis represents the model having a high recall, and the leftmost Y-axis does the same for precision. Therefore, The ideal case is positioned in the upper left corner of the diagram in the data presented in this thesis. This is only seen when looking at datasets containing specific Pfam entries.

One of the interesting questions is how the models can predict the presence of these Pfam entries; what does it look for? In the case of the specific Pfam entries explored in the results from the datasets named Pfam 1 through Pfam 20 in this thesis, the answer seems obvious in that it tries to learn the pattern associated with the entry in question. The question becomes more challenging to answer when looking at the more broad predictions needed in the other types of datasets, the general and order-based ones.

5.1.1 Capturing patterns

Analyzing the Roc curves presented for both the CNN model and the XGBoost model, see Figure 4.3 and Figure 4.4 respectively, shows one intriguing difference between the two. The convolutional model can capture up to 60% of the possible true positives while introducing close to no false negatives. In contrast, the XGBoost model plot shows a much more symmetrical result. As the Tsetlin machine produces binary results, no easy method was found to create a comparable illustration for this model.
It should be noted that this pattern of the model being able to easily predict around 60% of the positives occurs for all neural network models and is therefore not limited to the CNN model as it is also present in the DNN. The percentage of the easily predictable sequences varies between datasets, as seen in the cases of the models constructed on the Lacobacillales and Corynebacteriales datasets, see Figures 4.15 and 4.16. A natural question to ask is what exactly it is that these models are detecting that makes classification so obvious for them?

Due to the nature of machine learning models, especially neural networks, this isn't easy to answer in retrospect. No picture intuitive to the human mind is painted in the many weights that make up a neural network. We can, however, speculate. One potential explanation is how many instances of each Pfam domain were present in the data the model was trained on. Since the general dataset was constructed by randomly selecting 100,000 LORFs that contained any Pfam entry from the around 250 selected species, the chance of describing all the Pfam entries equally would be close to 0.

Some Pfam entries were vastly more present in the representative data, and why no analysis was made specifically on the species selected for the general dataset. The number of specific Pfam entries was noted for all species in the selected RefSeq genomes, limited to ORFs of lengths between 100 and 200 amino acids. In fact, only around 15 000 of the 21 000 Pfam entries were present in the data at all, as shown in Figure 4.19, while most of the ones represented were done so in the single digits. This might lead to the most common entries being vastly overrepresented with the ones of low frequency being labeled as noise. Unfortunately, with how the datasets were constructed it was impossible to find this out by running additional tests. If the specific Pfam entry had been noted for each sequence, it would be interesting to select the ones that were uncommon and see how well the model predicted them.

5.1.2 Length of sequences

One of the limitations of this thesis was that only sequences of lengths between 100 and 200 amino acids were utilized, and the rest were discarded. This limitation was implemented for multiple reasons, described in its own subchapter in the methodology 3.3.1.
Disregarding the issue of memory management, the primary reason for the limitations, selecting a specific region allowed the possibility of analyzing the specific region in much greater detail as the datasets would have to be of a similiar size anyway.
Preliminary results showed that the different lengths of the sequences changed the difficulty the models had in predicting correctly, with short sequences being much more difficult to predict than long ones.

GC content

As shown in the figures 4.9 and 4.10, the GC content of the species impacted the distributions of lengths of ORFs found an order. It also changed how difficult they were to predict, with high GC content being more difficult to predict. This might have to do with GC-rich genomes having relatively fewer LORFs as they contain less AT, which make up most of the stop codons. An analysis wasn't done to see if this was the case in this thesis, but it might be worth looking into further.

5.1.3 Selecting a Model

While most neural networks seemed to perform similarly in both the general and the order-based datasets, the CNN super model had a significant lead in the specific Pfam datasets as seen in Section 4.3. It seems that adding additional layers to the convolutions helped the model to be able to detect specific patterns, though as many layers as were added might have been a bit excessive.

5.2 Tsetlin machine results

Through all the iterations of the TM trained on the different datasets, none performed better than the other models, with the Tsetlin machine always showing performance, measured by accuracy, below the others.
While the overall accuracies of the Tsetlin model were often lower than the others, its recall was, for the most part, competitive and, in the case of the general data set, actually superior. Still, its poor Precision made its total performance worse, as other models could achieve a similar recall with an accompanying reduction in their precision.

5.2.1 Number of epochs

The Tsetlin machine utilized in this thesis had one major disadvantage emposed on it by how it was run. It always only ran using one epoch.
This was partly because of the time it required. Even using a GPU, unlike the other models, the Tsetlin machine used significantly more time for training and testing than the other models. With the exception of the Convolutional networks, these could use 100 epochs as they saw continuous improvements. As a total of 186 models were trained, increasing the amount of time spent training the models significantly became difficult to accept.

Another problem was how using multiple TM within a single script was difficult. When more than 1 TM was implemented in a single script it frequently triggered illegal memory access error messages. This made implementing any sort of early stopping mechanism a difficult endeavor, as storing the previous epochs' weight had to be done another way.
The TM also didn't have a dedicated validation set to ensure that it would not overfit to training data. As it only ran for one epoch, this would not have had any effect, but if more epochs were implemented, some sort of early stopping would put the model on a more even playing field.

5.2.2 Number of clauses

For both implementations of the Tsetlin machine, the number of clauses chosen was 10 000. To what degree these 10 000 clauses could properly represent the 14 000 different Pfam entries found in the dataset, can be argued. However, with the datasets being as skewed as they were, this should not have affected it too much.

5.2.3 Alternatives to direct amino sequences

It might be that the Tsetlin machine struggled more with running through the amino acid encoded. From the work done by Liland et. al. [16], the Tsetlin machine showed better results than a multi-layer perceptron when trying to locate genes in a nucleotide sequence. Changing from amino acids to nucleotides might show several advantages.
First of all, it would yield a much less sparse dataset, and since the Tsetlin machine contains so many clauses, each of which contains automata of an equal amount to the features of the dataset, it might be more efficient when it comes to using computational resources.
Alternatively, one could experiment with feeding the model different variations of k-mer to see if any other gave results. The suggested 1-mer(nucleotides) and 3-mer(amino acids) might not be the best solutions, even though they both seem likely to be good choices.

5.2.4 Convolutional models

The performance of the Convolutional Tsetlin Machine was unimpressive in all tested datasets. One of its main problems was that it never truly stabilized, and running multiple epochs seemingly resulted in a new randomized output each time instead of showing gradual improvement. This might have been related to the way the filter dimensions were constructed. A filter size of 3x20 was chosen as this would best match the filter used in the convolutional neural network models. This might, however, have been a somewhat wrong way to think about it. These models would work differently as the TM would, in practice, identify 10 000 different combinations of amino acids, with only 4000 possible variations in the genetic code itself given 20 amino acids, these would also be accompanied by some information on where in the sequence it is it might struggle to compare with the CNN.
The CNN, the simple model especially, on the other hand, learns to recognize 32 sequences of three amino acids anywhere within the provided LORF and denotes their positions within it. A dense layer then looks at this data and makes a prediction of where these 32 sequences were found.
While the intention was that these approaches should be somewhat similar, it might be that the fundamental differences between the models made these similar hyperparameters have drastically different implications when it comes to the actual workings of the models. The results might have been model structures that favored the neural network over the Tsetlin machine. The possibility that a more carefully structured Tsetlin machine, especially regarding the filter size, might have made the Tsetlin machine more competitive.
It might be that the chosen values for the number of clauses, s, and T should have been chosen through a more thorough search instead of opting for the estimates obtained from Olga Tarasyuk

et al. [21]. These hyperparameters should have been easy to optimize better in a structured format such as a grid search.

5.3 Creating a better model

With the results analyzed, one of this study's key areas of interest remains: how would one create a model that can filter out LORFs that do not contain known Pfam entries? The model suggested will be referred to as the combined model in the following discussion.

5.3.1 Choosing an approach

Through the model result analysis and by looking at the performance metrics presented in the results chapter, I hope I have fully explained why choosing a Convolutional neural network would be the best possible choice to go further to create a working filtering mechanism for LORFs of the models tested in this thesis.
Ideally, such a model would be created using a sampling method as used in the general dataset and trained in a similar way, as having a single model that could model everything would be both computationally faster and easier to implement. Unfortunately, the performance of the general model, i.e., the model trained on the general dataset, is seemingly a bit lacking considering it contained such a large amount of false negatives. And from the ROC curve presented in Figure 4.3 there didn't seem to be a way to decrease this without limiting the filtering power.
As such, the results from the specific Pfam datasets will be used to see if any greater model can be created by using their great performances.

5.3.2 Challenges

Number of models

One problem with creating a universal model using the methodology that lies behind the models trained on the specific Pfam database is that it would need many models. With over 20 000 Pfam entries there would need to be an equal number of models. When operating on large datasets, such as the proposed usage of metagenomic data, this would result in interference times that would be undesirable for running a filtering mechanism.
Pfam 1, results of which are shown in Table 4.3, showed that the CNN super needed 0.194 seconds for evaluating 1 600 LORFs. The timeframe to compare it to will be the month the HMMER software used to process all 150540245 LORFs that exist in the reference data.

Dividing the interference time with the number of LORFS to multiply them with the total number of Pfam entries should give an approximate amount of time the combined model would use for a given sequence.

$$\frac{0.194 \times 20795}{1600} = 2.521$$

This implies the models would require about 2 and a half seconds to process a single sequence. Multiplying this again with the number of sequences in the original data would result in:

$$2.521 \times 150,540,245 \approx 379,571,232.87 \text{ seconds}$$

This would be equal to:

$$\begin{aligned}
\text{Minutes} : \frac{379,571,232.87}{60} &= 6,326,187.21\\
\text{Hours} : \frac{6,326,187.21}{60} &= 105,436.45\\
\text{Days} : \frac{105,436.45}{24} &= 4,393.19\\
\text{Years} : \frac{4,393.19}{365.25} &= 12.03
\end{aligned}$$

Needless to say, running a program for 12 years in the hope of making a filtering mechanism to speed up a process that took 1 month would be meaningless.

This makes the assumption that every LORF will be padded up to 200 or limited to that length. But still, for illustrative purposes it was shown that such a model would be undesirable.

One solution to this problem is grouping certain Pfam entries together, as was done with the cumulative datasets within the Specific-Pfam datasets. Randomly grouping the results together showed a slight decrease in performance, and it is likely that the downward trend would continue if more Pfam entries were added. With accuracies being reduced to 98.2%, as shown in Table 4.4, creating 1 000, models like this would result in a large number of false positives being added, as every model would add a few.

While not explored in this thesis it might be worth looking at grouping the Pfam entries by clan. Clans describe larger patterns within LORFs, even further back in evolutionary history. This would still result in many models, but hopefully, this could be a way to group together multiple Pfam entries without sacrificing too much performance.

Unequal frequency of Pfam entries

As the distribution of Pfam entries varies significantly, not all Pfam entries are well represented in the data. Around 6000 of the described Pfam entries are not present in the RefSeq data, given the limitations of being between 100 and 200 sequences, and thousands more are only represented in the single digits. One solution to this is augmenting the data with synthetic data. As the Pfam families and domains are described using p-HMMs, using these p-HMMs to construct random sequences that comply with the given pattern could potentially be used. One would have to consider that Pfam domains often only describe a small section of the sequence however, with many sequences containing multiple Pfam domains. The rest of the sequence would also have to be constructed in a realistic way so that true hits would be predictable from training on the simulated data.

Hopefully, the need for this might be mitigated somewhat by sampling for lengths outside of the range of 100 to 200 amino acids. Some Pfam entries might not be represented in this range simply because they are longer or similar to this range. If the model is longer, it will not fit into the LORF, and if it is of equal range, there will be little space for variations or inserted sequences. Regardless, even if entries are added from other categories of length, some action would have to be taken to ensure even rare Pfam entries are represented.

Length and GC

From the results of the order-based datasets and the provided analysis presented in 5.1.2, either GC, length, or both will have to be taken into consideration.

It might be a good idea to develop different models for sequences of different GC content. If it shows that a model trained only on high GC sequences performs better than one trained for all sequences then adding an additional model or two wouldn't be especially computationally taxing. It would increase training time significantly but interference would remain similar as the sequences would only be sent to the model appropriate for their GC content.

The models analyzed were only trained and tested on LORFs of lengths between 100 and 200, leaving a gap in the knowledge of how well the models function on other lengths. As the model in question is a CNN, it is expected to search through the whole LORF for a specific sequence of amino acids, and it will require that a large section of the LORF is visible to the CNN. Padding all models up to a large max size would be impractical, as it would increase both the computational cost and could also impact performance negatively. By adding much padding the models could become sensitive to the length of LORFs to a greater idea, as LORF length and how common Pfam entries are closely correlated, see Figure 4.9 and 4.10. One potential solution is to add train multiple models for different model lengths and send LORF of a certain length to its appropriate model.

If both Length and GC were taken into consideration it might lead to problems. If both length and GC were divided into three, this would result in nine different models. This could represent a significant amount of training needed.

5.3.3 Reevaluating the model

Given their great performance, using the models trained for the specific Pfam entries seemed lucrative. However, running 20 000 models would not make sense, given the time it would take. If natural groupings of Pfam entries, such as by clan, could lead to large groups still achieving good performance this could be a possibility. if 20 000 models could shrunken down to around 659, the number of Pfam clans, then, by using the same math as the time calculations above the model would need:

Changing from approximately 20 000 models to 659 gives the following.

$$\frac{0.194 \times 659}{1600} \approx 0.0799$$

Multiplying the above result by 150, 540, 245, we get:

$$0.0799 \times 150,540,245 \approx 12,028,730.10 \text{ seconds}$$

Converting from seconds into more intuitive time units gives the following:

$$\text{Minutes}: \frac{12,028,730.10}{60} = 200,478.84$$
$$\text{Hours}: \frac{200,478.84}{60} = 3,341.31$$
$$\text{Days}: \frac{3,341.31}{24} = 139.22$$

While 139 days would be better than 12 years for searching through every Pfam entry, the result would still be unusable. As such, the idea of using the models trained on specific Pfam models should be discarded, and therefore, the focus should go back to the general models. Calculating the time needed for just the general model to run through all the data gives an estimate of:

Time needed per sequence:

$$\frac{0.518}{8000} \approx 6.475 \times 10^{-5}$$

The time needed for all data:

$$6.475 \times 10^{-5} \times 150,540,245 \approx 9,747.48 \text{ seconds}$$

To a more intuitive time format:

$$\text{Minutes} : \frac{9,747.48}{60} = 162.46$$
$$\text{Hours} : \frac{162.46}{60} = 2.71$$

or close to three hours. This is a time that makes much more sense for such a tool. Especially given the potential markup by using a GPU, which, according to Buber and Diri[24], could give about speed ups of about 500%.
However, as discussed earlier, this model might just capture the most common Pfam entries, as suggested for how easily it predicts 60% or so of the data but struggles with the rest. Hopefully, this could be remedied by creating a better dataset by sampling wider regarding width and creating some artificial entries for the less represented entries. It might also be necessary to add some sort of correction regarding either GC content or length, as the order-based datasets showed that this could result in some troubles.

Such a model would look a lot like the models developed for the cumulative specific Pfam datasets but with many more well-represented Pfam entries. 20 000 was the number chosen for each Pfam in the cumulative test, but this could probably be reduced significantly. The models did, however, show a decrease in performance for each Pfam entry added, but this might be counteracted somewhat by adding more filters to the model, allowing it to detect more kinds of patterns. This might be preferable to adding more layers, as was done in the tuned CNN model, as so many sequences of amino acids had to be excluded given that only 64 out of a possible $20 * 20 * 20 = 8000$ possible triplets of sequences were included.

5.4 Reflections on methodology

The focus of this thesis became trying to create many models for many datasets, so not enough effort was put into optimizing the models. No models were optimized through methods such as grid searches or similar, and the architectures of the neural networks were not built up by iterating over variations to choose the best; instead, they were chosen somewhat randomly.

This also means that there is a large probability that better results would be obtainable by most models. While I maintain that the largest area of improvement is the construction of a better dataset to train on, it should not be forgotten that several percentage points of gain for metrics such as accuracy might be available, even from slight tuning.

This might be especially relevant since the same hyperparameters were run on all the datasets, even though what might be necessary to capture the difference between specific Pfam patterns and the general case of any Pfam entry being present. Or that the sequences with high GC content might contain repeated segments that a specialized approach might be able to handle better.

Better training, testing, and validation splits should be used when creating and evaluating the models. Only setting 4% of the LORFs in the dataset as testing data might have influenced all the metrics negatively regarding their accuracies. A suggestion would be to use a split of 70/15/15, ensuring that most of the data would be used for training and that a significant number of sequences would also be used for testing and validation.

Using DNA directly instead of going through amino acids, could also improve, among others, the computational speed of the models. while nucleotides would need three entries for everyone in the amino acid chains; since only 4 nucleotides exist compared to the 20 amino acids, it could be represented much sparser. If using a convolutional model in this way, a stride of three should probably be implemented to avoid using different reading frames than the ORF itself would suggest. Alternatively, some sort of embedding layer should be used to better represent the non-independent relationships between the different amino acids.

One of the most interesting tests that I didn't have the opportunity to run would be to see how the general model operated on datasets with specifically labeled Pfam data. To see if the general models were able to detect uncommon Pfam entries, even though its training data was as skewed as it was.

It might also result in a more precise model if the Pfam entries were limited to just families. As it is now, it tries to learn if any given sequence maps onto a larger pattern observed in proteins with the same evolutionary origin and identifies smaller units within the DNA. Constructing many different models proved to be consuming given many Pfam entries. If these were limited to the broader categories of family or domain instead of the general entry, it might be found that different models work better for one type than the other.

While it would require a non-trivial amount of effort, the results indicate that creating a filtering mechanism would not be impossible as detecting Pfam entries in Lorfs seem to be something machine learning models are able to do.

5.5 Use of AI

Artificial intelligence tools were an important part of the process of writing this master thesis. The AI tools used in specifics were Grammarly, Chatgpt, and Github Copilot.

While no part of this thesis was written by an AI, the AI tool Grammarly was used to correct spelling mistakes and improve sentence structures. In addition, OpenAis chatGPT was used to provide feedback on some paragraphs, and some of the suggested improvements were implemented. These often involved making more precise statements and reducing the words used to talk around topics.

While AI was used to help in the writing process, it was during the coding that it provided the most support. Gitbubs copilot has become an integrated process in how I code, by making quick sketches of how I image specific functions and methods to use. Indirectly, it has also made me better at documenting my code, even if that documentation is often worded as instructions.

ChatGPT was also helpful with the coding, particularly when trying to write R code to process the LORFs and HMMER outputs into proper datasets, as this is a language I am less familiar with. It also served as a sparring partner when considering new approaches to handle the numerous issues that appeared throughout the process. One such example was how running multiple datasets in sequence caused the computer to run out of memory and thus crash; asking how I could automate the process using a bash script provided me some rough outlines that I then improved on to make a working script for the software. At some point, I also experimented with using Pytorch for the models, as this worked better for my computer setup regarding GPU; ChatGPT was able to ”translate” the models from tensorflow to Pytorch. This implementation was not used, however, but similar ones were.

Chapter 6

Conclusion

Throughout this thesis, the results of 6 different models trained for each of the 31 created datasets, have been presented and analyzed. The main differences between these datasets have been how their content has been sampled from a much greater dataset and what is set as their precise target value. The datasets include the general dataset that samples as wide as possible regarding taxonomy, the order-based models that sampled from specific portions of the taxonomy and the specific Pfam datasets that sampled from the whole dataset used using the presence of specific Pfam entries as target variables.

The result showed variation between the different models, from which some key insights could be gathered. Firstly, the neural network models could identify specific Pfam entries in sequences with high accuracy. These performances were then reduced for each additional Pfam entry added. The general dataset, containing a much larger number of Pfam entries, performed significantly worse. The general set models were, however, able to easily identify around 60% of the entries, with one provided assumption being that this could be because only some Pfam entries saw significant representation.

It was therefore suggested that a new model should be created that used a better sampling of Pfam entries instead of reaching for a wide taxonomy. What accuracy levels such a model would reach is unknown, but it could be better. As the multi-layered CNN model performed the best on the cumulative dataset, it would be a good point to start from when creating a better model. Exploring the potential of using other encoding formats, such as nucleotides instead of amino acids, as well as fine-tuning the model in regards to hyperparameters, are also suggested as potential regions of improvement.

The performance is as good as it was suggested that creating a tool that would be able to filter out ORFs not likely to contain Pfam hits is a genuine possibility, even when no such tool was created from the models created in this thesis.

While the Tsetlin machine has earlier shown promise when detecting genes from DNA sequences, the results found in the work presented in this thesis do not show the same for Pfam entries in amino acid sequences. The TM underperformed compared to all other models tested on all datasets. While this doesn't show that the Tsetlin machine is unsuited for analyzing amino acid sequences in regards to identifying Pfam entries, it suggests that if it isn't unsuited, a different approach has to be chosen.

Bibliography

[1] NHGRI. *The Cost of Sequencing a Human Genome.* 2021. URL: https://www.genome.gov/about-genomics/fact-sheets/Sequencing-Human-Genome-cost.

[2] J. D. WATSON and F. H. CRICK. "Molecular structure of Nucleic Acids: A structure for deoxyribose nucleic acid". In: *Nature* 171.4356 (Apr. 1953), pp. 737–738. DOI: 10.1038/171737a0.

[3] Patricia Sieber, Matthias Platzer, and Stefan Schuster. "The definition of open reading frame revisited". In: *Trends in Genetics* 34.3 (Mar. 2018), pp. 167–170. DOI: 10.1016/j.tig.2017.12.009.

[4] Lars Snipen and Kristian Hovde Liland. *The Cost of Sequencing a Human Genome.* 2023. URL: https://cran.r-project.org/web/packages/microseq/microseq.pdf.

[5] Matthew Cobb. "60 years ago, Francis Crick changed the logic of biology". In: *PLOS Biology* 15 (Sept. 2017), e2003243. DOI: 10.1371/journal.pbio.2003243.

[6] Eugene Koonin. "Does the central dogma still stand?" In: *Biology direct* 7 (Aug. 2012), p. 27. DOI: 10.1186/1745-6150-7-27.

[7] Jaina Mistry et al. "Pfam: The Protein Families Database in 2021". In: *Nucleic Acids Research* 49.D1 (Oct. 2020). DOI: 10.1093/nar/gkaa913.

[8] Sean R. Eddy. "Accelerated profile HMM searches". In: *PLoS Computational Biology* 7.10 (Oct. 2011). DOI: 10.1371/journal.pcbi.1002195.

[9] Tianqi Chen and Carlos Guestrin. "XGBoost: A Scalable Tree Boosting System". In: *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining.* KDD '16. ACM, Aug. 2016. DOI: 10.1145/2939672.2939785. URL: http://dx.doi.org/10.1145/2939672.2939785.

[10] John M. Jumper et al. "Highly accurate protein structure prediction with AlphaFold". In: *Nature* 596 (2021), pp. 583–589.

[11] Ashish Vaswani et al. *Attention Is All You Need.* 2023. arXiv: 1706.03762 [cs.CL].

[12] Ole-Christoffer Granmo. *The Tsetlin Machine – A Game Theoretic Bandit Driven Approach to Optimal Pattern Recognition with Propositional Logic.* 2021. arXiv: 1804.01508 [cs.AI].

[13] K. Darshana Abeyrathna, Ole-Christoffer Granmo, and Morten Goodwin. "Extending the Tsetlin Machine With Integer-Weighted Clauses for Increased Interpretability". In: *IEEE Access* 9 (2021), pp. 8233–8248. DOI: 10.1109/ACCESS.2021.3049569.

[14] Ole-Christoffer Granmo et al. *The Convolutional Tsetlin Machine.* 2019. arXiv: 1905.09688 [cs.LG].

[15] Yva Jacob Sandvik. "Evaluation of machine learning approaches for prediction of protein coding genes in prokaryotic DNA sequences". In: (2022).

[16] Kristian Hovde Liland et al. “Tsetlin Machine in DNA sequence classification : Application to prokaryote gene prediction / A match made in silico”. In: *2023 International Symposium on the Tsetlin Machine (ISTM)*. 2023, pp. 1–7. DOI: 10.1109/ISTM58889.2023.10454960.

[17] Nuala A. O’Leary et al. “Reference sequence (RefSeq) database at NCBI: current status, taxonomic expansion, and functional annotation”. In: *Nucleic Acids Research* 44.D1 (Nov. 2015), pp. D733–D745. ISSN: 0305-1048. DOI: 10.1093/nar/gkv1189. eprint: https://academic.oup.com/nar/article-pdf/44/D1/D733/9482930/gkv1189.pdf. URL: https://doi.org/10.1093/nar/gkv1189.

[18] Amani A. Al-Ajlan and Achraf El Allali. “CNN-MGP: Convolutional Neural Networks for Metagenomics Gene Prediction”. In: *Interdisciplinary Sciences, Computational Life Sciences* 11 (2018), pp. 628–635. URL: https://api.semanticscholar.org/CorpusID:56895112.

[19] Ole-Christoffer Grandmo and Per-Arne Andersen. *Tsetlin Machine Unified (TMU) - One Codebase to Rule Them All]*. 2024. URL: https://cran.r-project.org/web/packages/microseq/microseq.pdf.

[20] Sondre Glimsdal and Ole-Christoffer Granmo. *Coalesced Multi-Output Tsetlin Machines with Clause Sharing*. 2021. arXiv: 2108.07594 [cs.AI].

[21] Olga Tarasyuk et al. “Systematic Search for Optimal Hyper-parameters of the Tsetlin Machine on MNIST Dataset”. In: *2023 International Symposium on the Tsetlin Machine (ISTM)*. 2023, pp. 1–8. DOI: 10.1109/ISTM58889.2023.10454969.

[22] François Chollet et al. *Keras*. https://keras.io. 2015.

[23] The UniProt Consortium. “UniProt: the Universal Protein Knowledgebase in 2023”. In: *Nucleic Acids Research* 51.D1 (Nov. 2022), pp. D523–D531. ISSN: 0305-1048. DOI: 10.1093/nar/gkac1052. eprint: https://academic.oup.com/nar/article-pdf/51/D1/D523/48441158/gkac1052.pdf. URL: https://doi.org/10.1093/nar/gkac1052.

[24] Ebubekir BUBER and Banu DIRI. “Performance Analysis and CPU vs GPU Comparison for Deep Learning”. In: *2018 6th International Conference on Control Engineering & Information Technology (CEIT)*. 2018, pp. 1–6. DOI: 10.1109/CEIT.2018.8751930.

Appendix A

Table of R-packages, for instance

R-package name	Version	Purpose of use
Dplyr	1.1.4	Manipulate dataframes
Microseq	2.1.6	Read Fasta files
Micropan	2.1	Read HMMER outputs

Table A.1: R-packages used to construct datasets

Python-package name	Version	Purpose of use
Pandas	2.2.0	Handling the dataframes
Numpy	1.26.4	Manipulating arrays
Scikit-learn	1.4.0	Metrics and confusion matrix
Matplotlib	3.8.4	Plotting
Seaborn	0.13.2	Plotting
XGBoost	2.0.3	Implementing XGBoost model
Tensorflow	2.15.0.post1	Implementing neural networks
TMU	0.8.2	Implementing Tsetlin machines

Table A.2: Python packages used for both models and vizualisations