THE
ROBERT GORDON
UNIVERSITY
ABERDEEN

A sequence-length sensitive approach to learning biological grammars using inductive logic programming

Thierry Mamer

A thesis submitted in partial fulfilment
of the requirements of
The Robert Gordon University
for the degree of Doctor of Philosophy

January 2011

Abstract

This thesis aims to investigate if the ideas behind compression principles, such as the Minimum Description Length, can help us to improve the process of learning biological grammars from protein sequences using Inductive Logic Programming (ILP). Contrary to most traditional ILP learning problems, biological sequences often have a high variation in their length. This variation in length is an important feature of biological sequences which should not be ignored by ILP systems. However we have identified that some ILP systems do not take into account the length of examples when evaluating their proposed hypotheses.

During the learning process, many ILP systems use clause evaluation functions to assign a score to induced hypotheses, estimating their quality and effectively influencing the search. Traditionally, clause evaluation functions do not take into account the length of the examples which are covered by the clause. We propose L-modification, a way of modifying existing clause evaluation functions so that they take into account the length of the examples which they learn from.

An empirical study was undertaken to investigate if significant improvements can be achieved by applying L-modification to a standard clause evaluation function. Furthermore, we generally investigated how ILP systems cope with the length of examples in training data.

We show that our L-modified clause evaluation function outperforms our benchmark function in every experiment we conducted and thus we prove that L-modification is a useful concept. We also show that the length of the examples in the training data used by ILP systems does have an undeniable impact on the results.

Acknowledgments

I would like to thank my first principlal PhD supervisor Dr Chris Bryant for his invaluable help and feedback during this whole project, from the start of the literature review up until the successful completion of this thesis. His expertise in the field of machine learning, data mining and molecular biology were crucial to the sucess of this work.

I would also like to thank Professor John McCall, who was my secondary supervisor at the start of this project, but took over the role of principal supervisor after Dr Chris Bryant left The Robert Gordon University. His experiences as a researcher and his ability to see things from a different perspective often proved very valuable.

I would like to thank both my supervisors for their continuous support, even though this project greatly exceeded the originally intended timeframe, and for believing in me and that I would finally get the job done.

Furthermore, I would like to thank Dr Daniel Fredouille who was an advisor to the project at its early stages. His feedback and explanations were very usefull to understand the reviewed literature.

I would also like to express my thanks to some of my fellow PhD students from the Robert Gordon University who supported and motivated me all throughout the project, I especially owe gratitude to Ulices Cervinio Beresi who helped me in countless ways and who I will always consider a dear friend.

I would like to express my thanks to the Robert Gordon University in general and in particular to the School of Computing and its staff for giving me with the chance to undertake this project and providing me with such a good working environment.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Many Scientists have striven to find ways to represent data more efficiently, using as few resources as possible. One of the simplest examples is the decision to represent more common letters by shorter code-words during the design of Morse-code in 1838. Modern data compression was pioneered by Claude Shannon when *Information Theory* (Shannon 1948) was developed. Today, data compression usually implies the translation of certain information into a different language or encoding scheme, so that it occupies as little storage as possible. This makes storing and transmitting information much more efficient. However, data compression has advantages that go beyond simply saving storage. Observing the compressibility of certain information can actually tell us something about the information that we are trying to compress. It indicates the presence of regularity in our information, which can form the basis for important conclusions. Compression principles like *Minimum Description Length* formalize this idea and enable us to use computing techniques to determine the presence of regularities, make use of them, possibly discover their nature and draw conclusions from them.

One of the aims in Machine Learning is to learn from data, capture useful information, characteristics of interest, and draw conclusions based on those characteristics. This is done by analysing data, often large amounts of data. Some machine learning paradigms construct hypotheses based on the data that is being analysed. A common problem is that often many different hypotheses could be drawn from observed data, but not all of

those hypotheses might be desirable, and there is a need to judge which, of the possibly many hypotheses learned, are the best ones. This is sometimes referred to as the *Model Selection problem*. Past research (Kirchherr, Li & Vitanyi 1997) (Stahl 1996) has shown that compression principles are very useful in solving model selection problems.

In Molecular biology, vast amounts of data are being accumulated and scientists need tools to enable them to analyse this data. Biological sequences, like DNA, RNA or protein sequences consist of long strings of characters defined by chemical alphabets. Bioinformaticians use computer science techniques to analyse and learn from such biological data. One structure that can be used to represent and classify biological sequences are grammars, so called *biological grammars*. Machine learning techniques can be applied to learn biological grammars by analysing given biological sequences.

1.1 Research Question

The research question which was addressed in this thesis was as follows:

> Can compression principles such as the Minimum Description Length principle be used to improve the process of learning Biological Grammars using Inductive Logic Programming, and how?

This thesis investigates the application of Inductive Logic Programming (ILP) to the learning of biological grammars, specifically grammars describing protein sequences. It is investigated how an ILP system copes with the large variation in the length of protein sequences. The aim is to improve the way individual grammar rules are evaluated as they are induced, with respect to the fundamental ideas of the compression principles. It is also investigated how the complete biological grammars, learned by an ILP system are evaluated with respect to the biological data they are learned from.

1.2 Motivation

It is known that compression principles, like Minimum Description Length, are a useful tool to evaluate proposed theories which aim to explain a given observation. In Machine

Learning, where the aim is to automatically propose such hypotheses, there are quite a few hypothesis evaluation methods in use that exploit this fact. However, we still felt that when it comes to the specific task of learning biological grammars, most of those methods ignore one important property of the data which they learn from: the length of the examples. It is often the case that biological training examples, like protein sequences, have a high variation in their length, however most of the current hypothesis evaluation methods fail to take this variation into account.

We set out to investigate if there are any advantages in taking into account the length of examples in training data when evaluating grammar rules learned by machine learning tools, specifically tools using the Inductive Logic Programming paradigm.

1.3 Contribution to Knowledge

The work described in this thesis is solely the author's work, unless otherwise stated. A lot of the ideas developed in this thesis have arisen from discussions with the author's PhD supervisors and from the feedback given from peers in the ILP community.

The contributions to knowledge made by this thesis are in the field of Machine Learning, in particular in Inductive Logic Programming, and in Molecular Biology. As far as the author is aware this is the first work which applies a sequence-length sensitive approach to learning biological grammars using inductive logic programming.

This thesis proposes the concept of *L-modification*, a change of the variables used by ILP clause evaluation functions such that not only the count of covered examples is taken into account, but also the length of the examples themselves. This modifications aims to enable ILP tools to take the length of examples into account where they would be ignored otherwise. The results presented in this thesis show that the concept of L-modification, does enable an ILP system to learn better biological grammars than using an unmodified clause evaluation.

This study investigates:

- the effect of L-modification on learning biological grammars when compared to an unmodified benchmark,

- the effects of L-modification when applied to different ranges of lengths of examples,
- the relationship between the length of examples and the computational cost with respect to L-modification,
- the potential benefits of partial L-modification,
- the dominance of different parts of an L-modified clause evaluation function,
- and finally, the usefulness of L-modification when learning from noisy data.

Some of the ideas described in Chapter 6 have been proposed in:

> Mamer, T. and Bryant, C.H. (2006) Improving Biological Grammar Acquisition by considering the length of training examples, 16^{th} *International Conference on Inductive Logic Programming - Short Papers* (Mamer & Bryant 2006)

and most of the work and results presented in Chapter 6 have been published in:

> Mamer, T., Bryant, C.H., and McCall, J. (2008), L-Modified ILP Evaluation Functions for positive-only Biological Grammar Learning, *Proceedings of the* 18^{th} *International Conference on Inductive Logic Programming*(Mamer, Bryant & McCall 2008)

1.4 Structure of this Thesis

The structure of this thesis is as follows. The first three chapters set up the theoretical foundations for this work:

- Chapter 2 gives an introduction to molecular biology and bioinformatics. Some of the different grammatical structures are discussed and it is shown how these structures can be abstracted from biological sequences.
- Chapter 3 introduces model selection principles which are based on data compression and information theory. The main focus of this chapter is to introduce the Minimum Description Length principle and explain its relevance to biological grammars.

- Chapter 4 describes machine learning, leading on to Inductive Logic Programming (ILP), one of its sub-disciplines. It introduces some ILP learning tools and how they can be used to learn biological grammars.

The following three chapters describe the novel work that was conducted for this research:

- Chapter 5 describes all the details of the experimental setup that was used throughout this thesis.
- Chapter 6 marks the main contribution of this work, introducing the L-modification, which applies MDL concepts introduced in Chapter 3 to learning biological grammars and compares with the benchmark learning technique.
- Chapter 7 expands on the experiments described in the previous chapter by introducing the notion of noise and investigating the consequences of introducing noise into our dataset.

The last chapter finally offers a discussion of the research conducted:

- Chapter 8 summarises the research conducted, offers some conclusions and discussions and finally gives some suggestions as to how this work can be expanded on, describing some possible future work.

Chapter 2

From Molecular Biology to Biological Grammars

This chapter starts with a very brief overview of the field of Molecular Biology, mainly focusing on its subfields that are relevant to this work. This is followed by a similar introduction into Bioinformatics. Then the concepts of grammars are explained and within this context, we will look at Biological Grammars in particular. At the end of this chapter we will finally take a first look at Biological Grammar Acquisition.

2.1 Molecular Biology

> "Molecular Biology is the branch of biology that deals with the formation, structure, and function of macromolecules essential to life, such as nucleic acids and proteins, including their roles in cell replication and the transmission of genetic information." (AHSD 2010)

2.1.1 Proteins and Nucleic Acids

All living things strongly depend on the activities of proteins. Proteins are an unusual and complex family of molecules. There are many different kinds of proteins and they work together in large groups to carry out almost every biological function. Another unusual and complex family of molecules that are used by all living things are the two distinct

kinds of *nucleic acids*:

- *deoxyribonucleic acid*, usually referred to as *DNA*,
- *ribonucleic acid*, usually referred to as *RNA*.

Both proteins and nucleic acids are molecules that are considerably larger than most other molecules and are therefore referred to as biological macromolecules. Both proteins and nucleic acids are linear polymers, which means that they are molecules made up of long strings of just a few basic components. The components that make up nucleic acids are called nucleotides and the components that make up proteins are called amino acids. There are four different nucleotides: A (adenosine), G (guanine), C (cytosine) and T (thymine); and there are 20 different amino acids: (A, C, D, E, F, G, H, I, K, L, M, N, P, Q, R, S, T, V, W, Y). The relationship between proteins and nucleic acids is explained in what is called the *Central Dogma of Molecular Biology*, which states:

> "DNA molecules contain information about how to create proteins; this information is *transcribed* into RNA molecules which, in turn, directs chemical machinery which *translates* the nucleic acid message into a protein." (Hunter 2004)

DNA and RNA molecules are mostly two different means of storing information, so exchange between them is mere transcription, however proteins are mechanisms for action, so to construct it, information has to be translated into action. Some RNA molecules can also play a direct functional role, in which case they are not translated into a protein.

2.1.2 Structure and Function

Two main aspects of single Bimolecular components are studied deeply: their *structure*, how they describe themselves physically; and their *function*, the role they play in the process of life. There is a clear relationship between the structure and the function of bimolecular components and an important research area of molecular biology aims to discover these relationships between structure and function.

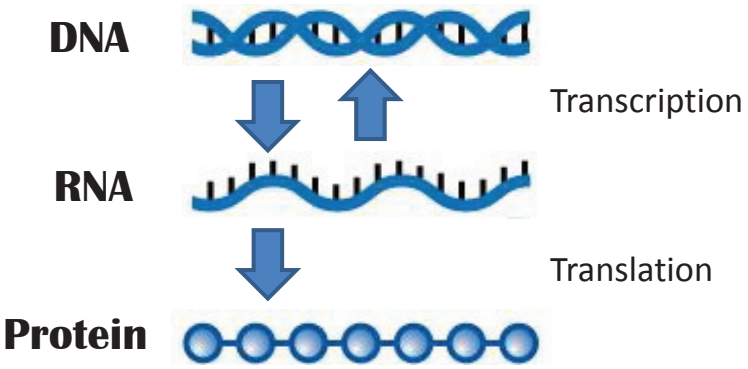

Figure 2.1: In a simplified form, the Central Dogma of Molecular Biology states that DNA is *transcribed* into RNA, and RNA is *translated* into proteins. Although it is unusual, RNA can be transcribed back into DNA (e.g. by retroviruses such as Aids), however once information gets into a protein, it can't flow back to a nucleic acid.

The main purpose of proteins is to realize the complex chemical reactions that are necessary to turn food into offspring. They can (among other things) serve as *enzymes* to enable chemical reactions by providing the required activation energy. This enzymatic function of proteins has three aspects:

1. *activity* (what chemical action an enzyme can do),
2. *specificity* (the ability of an enzyme to recognize and act on a particular material),
3. *regulation* (the ability to turn on or off an enzymes activity)

These three different enzymatic functions, activity, specificity and regulation, are realized by corresponding structural aspects of a protein (Hunter 2004). As mentioned in Section 2.1.1, proteins are linear polymers, made up of a sequence of amino acids. A protein can contain anything from a hundred up until several thousand amino acids, of which there exist 20 different ones. Each amino acid itself has different chemical properties (e.g. charged, heavy, hydrophilic, hydrophobic, etc.), therefore, the way they are lined up to assemble a protein determines its structure and thus, in the case of enzymes, it also defines the activity, specificity and regulation. The complex three-dimensional shapes that proteins assemble themselves into is crucial to their function; missfolded proteins can cause diseases such as the mad cow disease (Taubes 1996). Being able to predict the three dimensional shape of a protein, given its sequence of amino acids is a major aim in the

field of Molecular biology, which so far, is still elusive, even though much progress has already been made on this in recent years.

2.2 Bioinformatics

The NIH (U.S. Department of Health and Human Services National Institute of Health) Biomedical Information Science and Technology Initiative Consortium agreed on the following definition of *Bioinformatics*:

> "**Bioinformatics**: Research, development, or application of computational tools and approaches for expanding the use of biological, medical, behavioural or health data, including those to acquire, store, organize, archive, analyse, or visualize such data." (Huerta 2000)

In essence, this means that in Bioinformatics, biological problems are solved through the use of computing technologies. The term *Bioinformatics* was first used by Paulien Hogeweg (Hogeweg & Hesper 1978). A sub-discipline of bioinformatics is called *Computational Biology* which deals with the actual analysis and interpretation of biological data. Within Computational Biology there are many different types of data that need to be investigated; the area of interest to this thesis is concerned with protein sequence analysis. Many different kinds of structures have been used to represent or classify protein sequences, one such structure are grammars.

2.3 Grammars

A *grammar* is a set of rules that manages the construction of complex structures using small building blocks (Chomsky 2002). When we derive a legal structure using a grammar, we follow a derivation sequence, which is a sequence of legal choices made using the rules of the grammar. Grammar rules usually look like this: $S \Rightarrow xD$, where lower case letters represent terminal symbols (which are part of the provided alphabet) and uppercase letters represent non-terminal symbols (temporary symbols, placeholders, which do not appear

in the provided alphabet). The above rule states that if we are given the non-terminal S, we can replace it by x, followed by D.

As mentioned above, a grammar is a set of rules; consider grammar G, comprising of two rules $G : S \Rightarrow xS,\ S \Rightarrow y$. This grammar can generate any string that starts with any number $(0 - \infty)$ of occurrences of x, followed by exactly one y. *Derivation* is the process of rewriting a given starting symbol, S as often as it takes until the last non-terminal is replaced. E.g.: starting with non-terminal S, we get: xS, xxS, $xxxS$, $xxxxS$ and finally: $xxxxy$. Starting from S we applied the first rule of grammar G four times and the second rule once. After these five steps, we end up with the terminal string $xxxxy$, one out of infinitely many strings that can be derived using grammar G.

There are several classes of formal grammars which can be put into a precise containment hierarchy: the *Chomsky hierarchy*.

2.3.1 The Chomsky Hierarchy

In 1956 Noam Chomsky described a containment hierarchy for classes of formal grammars, known as the *Chomsky hierarchy* (Chomsky 1956). The Chomsky hierarchy includes, from specific to general: *Regular Grammars*, *Context-Free Grammars*, *Context Sensitive Grammars* and *Unrestricted Grammars*. Each of these classes grammars subsumes the previous one. Lets look at each of these grammar classes in more detail.

Regular Grammars

A regular grammar defines a regular language. A regular grammar consists of production rules whose left-hand-sides have one single non-terminal and whose right-hand-sides have one single non-terminal, one single terminal or both: $A \rightarrow t,\ A \rightarrow B$ *or* $A \rightarrow tB$, where $A, B \in N$ (set of all non-terminals), $t \in T$ (set of all terminals). The rule $S \rightarrow \epsilon$ (where ϵ is the empty string) is allowed if S does not appear on the right side of any rule. Regular languages can be recognized by a finite state automaton (See Appendix A.1). Regular languages can also be obtained by regular expressions (concise and flexible means for matching strings of text) and are often used to define the lexical structure of

programming languages.

Context-Free Grammars

A context-free grammar defines a context-free language. Production rules of a context-free grammar are of the form: $A \rightarrow \lambda$, where $A \in N$ (set of all non-terminals) and λ is a string of terminals and non-terminals. Context-free languages can be recognized by a non-deterministic pushdown automaton (See Appendix A.2) and are often used as theoretical basis for the syntax of programming languages.

Context-Sensitive Grammars

A context-sensitive grammar defines a context-sensitive language. The only restriction placed on context-sensitive grammars is that no production rule can map a string to a shorter string. Production rules of a context-sensitive grammar are of the form: $\alpha A \beta \rightarrow \alpha \lambda \beta$ where $A \in N$ (set of all non-terminals) and λ, α and β are strings of terminals and non-terminals. α and β may be empty but λ must not. The rule $S \rightarrow \epsilon$ is allowed if S does not appear on the right side of any rule. Context-sensitive languages can be recognized by a linear bounded automaton (See Appendix A.3)

Unrestricted Grammars

Production rules of unrestricted grammars can be of any form ($e.g. Baa \Rightarrow a$). Unrestricted grammars define *recursively enumerable languages*, which are all the languages that can be recognized by a Turing Machine (See Appendix A.4).

Every *regular* language is context-free, every *context-free* language is context-sensitive and every *context-sensitive* language is *recursively enumerable*. In turn, there exist *recursively enumerable* languages which are not *context-sensitive*, *context-sensitive* languages which are not *context-free* and *context-free* languages which are not *regular*.

2.3.2 Applications of Grammars

The main advantage of grammars is that they provide a single mechanism to describe many complex structures. Grammars can describe many structures, the most obvious are probably natural languages and programming languages, however grammars can also be used to describe graphs, neural networks, mathematical expressions, biological sequences or the manner in which molecules arrange themselves in compounds. Our work deals with grammars that describe biological sequences, more precisely, protein sequences. Such grammars are called *Biological Grammars*.

2.4 Biological Grammars

Since the early 1980s scientists have discovered that applying linguistic approaches to molecular biology yields some promising results. The use of the grammar types contained in the Chomsky hierarchy (See Section 2.3.1) enables us to capture informational aspects as well as structural aspects of macromolecules (Searls 2002). The primary structure of nucleic acids and proteins can be represented as a sequence of letters taken from a well defined chemical alphabet (See Section 2.1.1). When a protein is folded in three dimensional space, regions of the sequence which are linearly widely separated become close specially, therefore long range interactions between different parts of the sequence are important when capturing characteristics of proteins (Muggleton, King & Sternberg 1992a).See Figure 2.2 for an example of a protein. The black arrow points to a situation where two parts of the protein are close to each other and thus could potentially interact with each other. However in the linear sequence of the protein itself, those two parts are very distant from each other. Because of their declarative and hierarchical nature, formal grammars can define such long range dependencies in biological sequences.

2.4.1 Grammars describing Nucleic Acids

The most basic secondary structure element in RNA is called *stem-loop*, where the stem creates a succession of nested dependencies. These stem-loops can be described by a grammar that is at least context-free (See Section 2.3.1). However, there are more complex

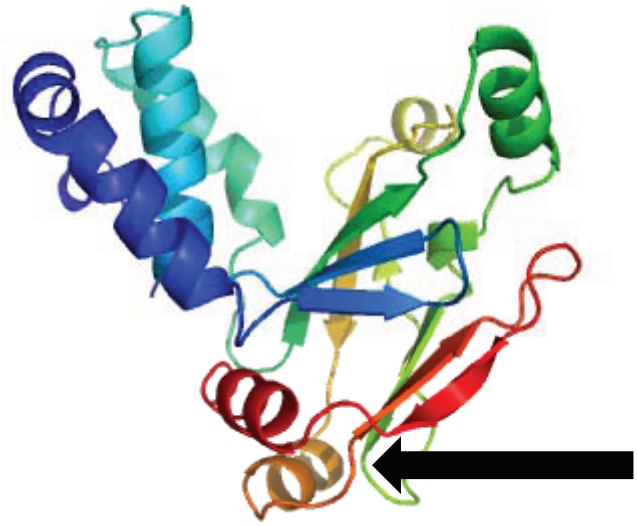

Figure 2.2: A Protein. The black arrow points at one of the locations where *dependencies between distant parts* might be possible.

elements in RNA that raise the grammar requirements above context-free. One group of such elements are called pseudoknots, which are pairs of stem loop elements where one stem can be found within the loop of another. A grammar able to describe pseudoknots needs the ability to define cross-serial dependencies, which entails that at least a context-sensitive grammar is required. A type of grammar that is often used to describe RNA secondary structures are the so called *tree-adjoining grammars* (Vijay-Shankar & Joshi 1986) (See Appendix A.5). In addition to that, new types of grammars have been invented (Searls 1995, Rivas & Eddy 2000) to describe nucleic acids.

2.4.2 Grammars describing Proteins

Generally there has been less work done in modelling proteins than there has been in modelling RNA. However, specific aspects of the structure of proteins have been modelled extensively using formal grammars. For example, stochastic tree grammars (similar to the tree-adjoining grammars in Appendix A.5) have been used to represent hydrogen bonds between strands in a β-sheet (Abe & Mamitsuka 1997).

2.4.3 Advantages of Biological Grammars

A Biological grammar describes specific patterns which are common to sequences of a certain family. Usually, all sequences of a certain family share a specific biological function. There are two main advantages of Biological Grammars compared to other means of describing biological sequences:

1. They can be used to annotate sequences of unknown function, providing molecular biologists with a likely function for such sequences.

2. Because a grammar structure emphasizes common points between sequences of similar functions they can help biologists to understand how such functions are realized.

2.5 Biological Grammar Acquisition

It has been shown that complex biological sequences can indeed be identified using linguistic approaches (Searls 1997, Searls 2002). If biological experts construct such grammars manually, it is very expensive and time consuming. An automatic process that can learn such biological grammars is therefore desirable, especially in sight of the massive amounts of data being accumulated by genome projects.

Biological grammars have already been acquired from biological data in many different forms: *String Variable Grammars* (SVG) (Searls 1993, Searls & Dong 1993), *Patscan patterns* (Dsouza, Larsen & Overbeek 1997) or *Basic Gene Grammars* (Leung, Mellish & Robertson 2001) deal with nucleic acids, while *Prosite patterns* (Jonassen 1997, Falquet, Pagni, Bucher, Hulo, Sigrist, Hogfmann & Bairoch 2002) and *Probabilistic regular* or *context-free grammars* (Bateman, Birney, Cerruti, Durbin, Etwiller, Eddy, Griffiths-Jones, Howe, Marshall & Sonnhammer 2002, Sakakibara, Brown, Hughey, Mian, Sjolander, Underwood & Haussler 1994) deal with protein families. These grammars have been acquired automatically using many different approaches: while Falquet et al (Falquet et al. 2002) have applied semi-automatic learning to obtain sub-regular grammars using alignment methods followed by human examination, others have used parameter optimisation, an approach that learns probabilities of stochastic grammars (See Appendix A.6) with prede-

fined structure. This produces stochastic regular grammars as in Bateman et al. (Bateman et al. 2002) or stochastic context-free grammars (See Appendix A.7) as in Sakakibara et al. (Sakakibara et al. 1994). Last but not least, Machine Learning approaches (See Section 4.1) have been used successfully to learn sub-regular grammars (Dassow 2005) by the pratt software (Jonassen 1997) and by Inductive Logic Programming (See Section 4.9) (Muggleton, Bryant, Srinivasan, Whittaker, Topp & Rawlings 2001).

2.6 Summary

In this chapter, we introduced the field of *Molecular Biology*, we described proteins, nucleic acids and their relationship between each other, which is referred to as the *Central Dogma of Molecular Biology*. We discussed the two main aspects of bimolecular components, namely: their structure (how they describe themselves physically) and their function (their role in the process of life). Furthermore, we introduced the field of *Bioinformatics*, which deals with computational tools that can be used to handle biological data. We gave a detailed description of the most common hierarchy of grammars, the Chomsky hierarchy, and the most common applications of grammars. We explained how grammars are used within Molecular Biology and in Bioinformatics. The last section of this chapter finally touched on the topic of *Biological Grammar Acquisition*, which marks an integral part of this thesis.

The following chapter will discuss model selection principles and why we are so interested in them. Most importantly, the *Minimum Description Length* principle will be described in detail.

Chapter 3

Model Selection Principles

Minimum Description Length (MDL) (Rissanen 1978) is a principle for inductive inference in Information theory. It is often used to decide among several models which all describe the data equally well. This problem is often referred to as the model selection problem (Burnham, Anderson & Burnham 2002). The MDL principle is a formal restatement of Occam's Razor, therefore this chapter starts with introducing Occam's Razor and shows how the MDL principle can be derived from it. Then the MDL principle will be formally defined and finally its relevance to this thesis is illustrated.

3.1 Hypotheses and Models

Throughout this chapter we will make extensive use of the terms: *hypothesis*, *model* and *data*. So before we start, we explain these notations.

The uses of the word *model* are quite diverse. In its most general sense, a model is anything that can be used in any way, to describe something else (Freudenthal 1961). In the context of this chapter: if observations can be made about a certain phenomena, then a model might give details on what caused the things we are observing. This way, a model may even explain the given phenomena itself.

A *hypothesis* is a proposed explanation for an observable phenomenon. Within the context of this work, the key word here is *proposed*. This word marks the key difference between a hypothesis and a model. When we attempt to describe and explain an obser-

vation, we first postulate a hypothesis. Then, if the hypothesis is accepted to be true, it becomes either a model, or part of a model.

When we refer to *data*, we usually mean the observations which we are trying to explain through our hypotheses.

3.2 Occam's Razor

The English philosopher and cleric William of Occam (1285 - 1349) formulated a principle which is widely accepted by scientists and commonly called *Occam's Razor*. In its original form, it says literally:

> "Entities should not be multiplied unnecessarily."

However, nowadays it is usually formulated as (Natarajan 1991):

> "The simplest explanation of the observed phenomena is most likely to be the correct one."

or, in a more logical form (Kirchherr et al. 1997):

> "Among all hypotheses consistent with the facts, choose the simplest."

In science, the primary activity is to formulate theories to explain observations. But for every observation there could be an infinite number of theories that explain that observation. We are then faced with the problem of deciding which theory is the best and most likely to be the true one, given the evidence at hand. Without a method of choosing between several competing and theoretically possible theories, science would cease to function entirely. Occam's Razor is a very effective tool to choose between competing theories, ranking theories by favouring those without redundant elements. A theory that has to make a lot of assumptions to explain the observed data is, as such, quite complex and will always be worse than a simpler theory that manages to explain the data without so many assumptions, which is why the latter is preferred. A simple theory that logically entails the observed data is usually in return similarly entailed by the data.

To illustrate this on an example, let us consider *Newton's Third Law*, which states that (Thomson, Guthrie & Howard 1912):

> "For every action there is an equal and opposite reaction."

This law is widely accepted as the truth; however there are many other theories that can explain the same observations. One such theory could be as follows:

> "For every action there is an opposite action of half intensity, but benevolent undetectable creatures magnify the opposing action with input of their own energy so it appears to be equal. These creatures will all die in the year 2055, and at that point the observable nature of the universe will instantly shift." (example taken from Microsoft Encarta article on Occam's Razor)

Our intuition tells us that the latter theory is unlikely to be the correct one, however it is quite impossible to prove it wrong. Because these benevolent creatures are said to be undetectable, there is no way to prove that they don't exist, nor that they are not doing what the theory claims they do. Furthermore, there can be an infinite number of such theories that explain the same observation. To illustrate this, consider the mention of the year 2055 in the theory. We can easily rewrite the same theory, but using the date 2056 or 2057 and so forth, hence it is easy to comprehend that there are in fact an infinite number of such theories. But as mentioned above, our common sense and intuition tells us that Newton's Third Law is most likely the correct theory. Why is this? The reason is that Newton's Law is simpler. It explains the observations in a credible way, does not make any assumptions that would be hard or impossible to prove and is, as such, less complex.

3.3 Principle of Indifference

In the history of science Occam's razor has seen many uses. It should be mentioned however that there are other principles that do not completely agree with Occam's razor; for example, the British philosopher Bertrand Russel restated the Greek philosopher Empiricus saying (Kirchherr et al. 1997):

> "When there are several possible naturalistic explanations ... there is no point trying to decide between them"

Today this is more known as the **Principle of Indifference** (Jaynes 2003):

> "Keep all hypotheses that are consistent with the facts."

The basis for this principle is that it is possible that more precision may be achieved when using several explanations, that are equally in agreement. Maybe all the evidence should be retained to provide more robust predictions. This suggests that one cannot follow Occam's razor too literally; after all it is more a guideline than a scientific fact. Some experts even suggest that Occam's razor can be *false* because models of real-world situations are usually quite complex, and therefore one should not always favour the simplest models.

3.4 Bayes's Rule

Thomas Bayes, an English mathematician came up with what can be considered to be a modified principle of indifference. Rather than just simply keeping all hypotheses that are consistent with the facts, his method allows us to assign probabilities to hypotheses. *Bayes Rule* is as follows:

> "The probability that a hypothesis is true is proportional to the prior probability of the hypothesis multiplied by the probability that the observed data would have occurred assuming that the hypothesis is true(Bayes 1764)."

As we will need to use *Bayes Rule* several times later in this thesis we can take this opportunity to derive its common formula. As a starting point, we have the definition of *conditional probability*, which states: the probability of an event A given an event B is:

$$P(A|B) = \frac{P(A,B)}{P(B)} \tag{3.1}$$

where $P(A,B)$ is the joint probability of A and B and $P(B)$ is the probability of B. At the same time, the probability of an event B given an event A is:

$$P(B|A) = \frac{P(A,B)}{P(A)} \tag{3.2}$$

We can now combine 3.1 and 3.2:

$$P(A|B)P(B) = P(A,B) = P(B|A)P(A) \tag{3.3}$$

Assuming neither P(A) nor P(B) are 0, we can now derive the more common form of *Bayes Rule*:

$$P(A|B) = \frac{P(B|A)P(A)}{P(B)} \tag{3.4}$$

3.5 Complexity

The *principle of indifference* tells us to keep all hypotheses explaining the data, *Bayes Rule* tells us that some are more likely than others and *Occam's Razor* tells us to prefer simple hypotheses and reject the more complex ones. The next step is to look at what exactly is meant by *simple* and *complex* within this context.

Between 1964 and 1969 three men, Ray Solmonoff (Solomonoff 1964), Andrei N. Kolmogorov (Kolmogorov 1965), and Gregory Chaitin (Chaitin 1966) independently from each other developed the concept of *Kolmogorov complexity*. Solmonoff addressed the question of how to assign a priori probabilities to hypotheses, while Kolmogorov and Chaitin were investigating the precise meaning of *randomness*. All three mathematicians came up with equivalent notions, showing that the two problems are fundamentally related and all three used the notion of *computable* to answer their questions.

Consider that a computer program is just a finite set of instructions showing a computer how to calculate a given function. From that we can derive that a computer program is just a computable binary sequence, a *string*. If we consider a particular programming language, like C, there are infinitely many programs that do not take any input, e.g. a program that prints out a binary string and then stops. If we consider all the programs that print out a given string S, we are looking at an infinite number of programs. But, out of all those, there is one or more programs which can be said to have the *shortest length*. This length is called the *complexity* of the output string. Formally:

> "The complexity of a binary string S is the length of the shortest program

which, on no input, prints out S."

Instead of simply looking at the length of **all** computer programs, we could restrict ourselves to just look at the prefix-free programs. Prefix-free in this context means that no program in the set is a prefix of another. This is not hard to achieve, we merely need a programming language using a specific codeword that marks the end of a program and which cannot be used anywhere else in the program. By definition, such a language can only produce prefix-free programs. This gives us the *Kolmogorov Complexity*:

> "The Kolmogorov Complexity K(S) is the length of the shortest program among all prefix-free programs, that prints out S."

The Kolmogorov Complexity itself is incomputable; we can never prove that a given program is the shortest possible program that prints the given data, yet, Kolmogorov Complexity is the key to proving other theorems related to complexity, such as the Minimum Description Length Principle.

3.6 Minimum Description Length (MDL)

Minimum Description Length (MDL) (Rissanen 1978) is a principle for inductive inference in information theory and a formal restatement of Occam's razor (See Section 3.2). MDL can be considered as an extension of Minimum Message Length (MML) (Wallace & Boulton 1968). They are both based on very similar ideas and are not that different from each other.

> "The fundamental idea behind the MDL principle is that any regularity found in the data can be used to compress the data; to describe it using fewer symbols than the number of symbols needed to describe the data literally. The more regularities there are, the more the data can be compressed. " (Grunwald, Myung & Pitt 2005)

When we refer to *describing the data*, we must take into consideration what methods of description are used. The method that is most interesting to us is a computer language.

Any data can be represented by a string of symbols from a finite alphabet. Therefore, for any data, it is possible to create a program that outputs it. The most basic among such programs could be of the form `write S, then halt`, where S is the string containing the data, but there are in fact infinitely many programs that could output S. Hence, for *describing the data* we consider a program that outputs or *prints* the data and then halts. If this program is much smaller than the data itself, then the data can be said to be highly compressible. To compress a sequence, any kind of regularity that is found within that sequence can be used to compress it. For a better understanding, consider the following examples:

- The number π, whose exact value has an infinite decimal expansion: its decimal places never end and have no repetitive pattern. Yet it is reasonably easy to write a program that prints the N first digits of π, and that program would be of a constant size, except for the specification of N. The more digits of π the program prints, the smaller the program itself will be in comparison, thus π is very compressible because of its regularities.

- A natural language, where a grammar can be used to significantly compress a text written in the given language (Grunwald 1996). This makes use of the regularities in the language and the constraints in the order of words. Something that can also be used is the meaning that text contains, e.g.: "Aberdeen" is shorter than "the granite city", which illustrates that meaningfulness and compressibility are closely related. The same idea can be applied to biological sequences (e.g. protein-sequences), which can be compressed by biological grammars (See Section 4.5.1).

As mentioned above, there are, in theory, an infinite number of programs that could print (describe) some given data. As the name of the principle suggests, MDL aims to minimize the length of such a program. More precisely, out of all the programs that exist which can describe the given data, we chose the shortest one. The length of this shortest program (assuming it is written in a prefix-free language) is called Kolmogorov complexity (Kirchherr et al. 1997) (See the end of Section 3.5). The lower the Kolmogorov complexity,

the more regularities are in the data. According to Solmonoff (Solomonoff 1964), the best model for a sequence of data is the shortest program that prints the data. Later these ideas lead to an idealized MDL principle. Idealized because the MDL principle itself is practically not implementable, mainly because the Kolmogorov Complexity is incomputable. As mentioned in Section 3.5, we can never prove that a given program is indeed the shortest possible program that prints the given data. A less important flaw is that the length of such a program also depends on the syntax used by the chosen language, but this only becomes a problem when we are dealing with very short sequences.

3.6.1 Formal definition for MDL

(Kirchherr et al. 1997): Using the Kolmogorov Complexity $K(S)$, the length of the shortest program which, on no input, prints out S (See Section 3.5), we can actually assign *a priori probabilities* to binary strings which is the probability of the string occurring:

$$P(S) \;=\; 2^{-K(S)} \tag{3.5}$$

where $K(S)$ is the Kolmogorov complexity of the string S, $2^{-K(S)}$ is the a *prior probability* of S. The formula (3.5) assigns higher probabilities to strings with shorter Kolmogorov complexity. In other words, higher probabilities are assigned to simpler strings. This reflects Occam's Razor precisely: Simpler things are more likely. The Kolmogorov complexity of a string S is:

$$(3.5) \implies \quad K(S) \;=\; -\,log_2(P(S)) \tag{3.6}$$

As we have discussed previously (See Section 3.5), any computer program is just a computable finite length string. The same can be said for models, which are most often represented by computer programs of some kind. Thus we can conclude that a model can be represented by a string. (Rissanen 1978): The Kolmogorov complexity of a model M,

which is also called its *Information Content* $I(M)$ (Stahl 1996) is:

$$(3.6) \implies K(M) = -log_2(P(M)) = I(M) \tag{3.7}$$

Bayes rule from Section 3.3:

$$(3.4) \implies P(H|D) = \frac{P(H) \cdot P(D|H)}{P(D)} \tag{3.8}$$

where D is the data observed and H is the hypothesis describing the data D. In this work, the data observed is usually a set of examples E, and a hypothesis able to successfully describe those examples is called a model M. (See Section 3.1). Applying this to Equation (3.8):

$$(3.8) \implies P(M|E) = \frac{P(M) \cdot P(E|M)}{P(E)} \tag{3.9}$$

where E is the set of examples and M is the model that explains the examples. $P(E|M)$ is the probability of the examples given the model M and $P(E)$ is the probability of the examples, which is constant, and because we can assume that the examples are correct, equal to 1 (Stahl 1996). The complexity of a string S is:

$$(3.9) \implies P(M|E) = P(M) \cdot P(E|M) \tag{3.10}$$

$$(3.10) \implies -log_2(P(M|E)) = -log_2(P(M)) + (-log_2(P(E|M))) \tag{3.11}$$

$$(3.11) \;\&\; (3.7) \implies I(M|E) = I(M) + I(E|M) \tag{3.12}$$

$I(M|E)$ is the information content (description length) of the model, given the examples. It is the sum of the description lengths of the model M and the examples E encoded with the model M. This is the description that has to be minimized (Stahl 1996) (therefore the name MDL) and as a consequence $P(M, E)$ has to be maximized.

3.6.2 Minimum Message Length principle (MML)

Minimum Message Length principle (MML) is slightly older than MDL. It is based on very similar ideas and draws similar conclusions. It is worth giving a summary of how the MML principle is formally derived:

We know from information theory (Shannon 1948) that a string S with an assigned probability P(S) has a (binary) message length of length:

$$length(S) \;=\; -\,log_2(P(S)) \tag{3.13}$$

We are looking for the shortest description of both the model M and the data encoded with the model $E|M$ together. Deriving Bayes theorem:

$$P(E|M) \;=\; \frac{P(M,E)}{P(M)} \tag{3.14}$$

$$P(M,E) \;=\; P(E|M) \;\cdot\; P(M) \tag{3.15}$$

we find that it is $P(M,E)$ for which we are seeking the shortest description:

$$length(M,E) \;=\; -\,log_2(P(M,E)) \tag{3.16}$$

$$-log_2(P(M,E)) \;=\; -\,log_2(P(E|M) \;\cdot\; P(M)) \tag{3.17}$$

$$-log_2(P(M,E)) \;=\; -\,log_2(P(M)) \;+\; (-log_2(P(E|M))) \tag{3.18}$$

where $-log_2(P(M,E))$ is the length of the model M and the data encoded by the model E; $-log_2(P(M))$ is the length of the model M (model complexity) and $(-log_2(P(E|M)))$ is the length of the data E encoded by the model M (model accuracy). So by minimising $-log_2(P(M,E))$ we are minimising the combination of model complexity and model accuracy, thus MML trades model complexity for model accuracy. A system following the MML principle would therefore never choose a more complicated model over a simpler one unless it pays off when encoding the data.

3.6.3 MDL and MML

After reading Sections 3.6.1 and 3.6.2 one might wonder what the practical differences are between the MML and the MDL principles. Unfortunately the precise differences between MDL and MML are not always obvious. Sometimes it seems from the literature that experts themselves do not really agree on the boundaries between both of them, sometimes they are simply treated as being identical, but in fact, they are not. They do indeed cover the same territory; they are both concerned with the same scenario: we have some data and a sender that has to send this data in a message through a channel in the most efficient way. The aim is to describe the data in the shortest way possible. The receiver of the message takes what comes from the channel and uses it to recreate the original data. Both MML and MDL code the parameters of models and then choose the theory that describes the model and the data with the shortest message.

It is unclear how best to define the differences between MML and MDL. The key difference seems to be that with MDL the sender has to send the parameters of a model together with the examples' codes whereas in MML the sender has to send the encoding of the model together with its parameters and the encoding of the examples using the model. Nowadays the notion of MDL generally seems to be the most popular one.

3.6.4 Overfitting

When the MDL principle is applied to Machine Learning (ML) (See Section 4.1) it has some very desirable advantages. The purpose of learning models in ML is to enable us to predict future data that will come from the same source as the data that we are analysing. If the resulting model is naively using every single available regularity down to the letter to describe the seen data, then its performance on unseen data might not be so good, in fact it's usually worse. This problem is called *overfitting*, the model is too closely specialized for the data that is analysed to be any good for unseen data. Another property of such overfitted models is that they are usually quite complex. In practice, most model-selection methods make some kind of a trade-off between goodness-of-fit and complexity for their models as this leads to much better performance on unseen data. The

advantage of MDL is that this trade-off is done automatically in the theory of MDL. We look at the description length of the data and minimize it (*Minimum Description Length*). However the description of the data often does not only contain the model itself, but also its exceptions, i.e., the errors of the model. In other words, it contains everything that is needed to reproduce the data. This prevents two extreme cases from happening: In one extreme case, where we have a very complex model whose description will be quite long and together with the exceptions even longer, the description of the data is not the smallest one possible and should not be chosen. In another extreme case, where we have a very basic model whose description will be very short, but together with the many exceptions of this model, the description of the data is bound to be very long and should therefore not be chosen either. Because MDL seeks a short description of the data, both of these extreme cases are prevented by choosing a quite, but not too, simple model with a good, but not perfect, fit (Grunwald et al. 2005).

Satisfying Occam's razor

Now that we have introduced the MDL principle, we can revisit the start of this chapter and verify if we did indeed respect Occam's Razor (Section 3.2).

MDL seeks the shortest description length of the data, more precisely, the shortest description length of both the model and the examples encoded with that model. So, in the situation where we have 2 different competing models that both fit the data equally well, the exceptions to both models should be the same. As far as MDL is concerned, it will then choose the model that has the shortest description. Considering that the model with the shortest description is also the least complex and simplest one (according to Section 3.5) MDL is proven to implement the principle of Occam's razor precisely.

3.7 Summary

We started this chapter introducing the necessary foundations, like *Occam's Razor*, the *Principle of Indifference* and *Bayes Rule*. From there, we moved on to discuss what exactly complexity is, leading to the introduction and definition of the *Kolmogorov Complexity*.

Finally we went into explaining the details of *Minimum Description Length (MDL)*, including the formal derivation of this compression principle. We explained how the MDL principle helps to prevent overfitting when applied to *Machine Learning*.

The next chapter will introduce *Machine Learning* and in particular, *Inductive Logic Programming*. Furthermore, Chapter 4 will make the connection between *Machine Learning*, *Molecular Biology* (from Chapter 2) and *Minimum Description Length* (explained in this chapter).

Chapter 4

Machine Learning & Biological Grammar Learning

This chapter introduces Machine Learning, focusing on one of its subfields, Inductive Logic Programming, and all the relevant concepts and tools needed in this work. It also introduces Biological Grammar Learning from a Machine Learning perspective and gives a summary of previous work done, and finally explains the full motivations of this work.

4.1 Machine Learning (ML)

Machine learning (ML) is a sub-discipline of artificial intelligence that is concerned with the design and development of algorithms and techniques that allow computers to *learn*. In other words, ML is trying to answer the question of how to build computer programs that automatically improve their performance at a task through experience (Mitchell 1997). ML has already been used extensively in a variety of application domains. One such domain is *Data Mining*, which is about analysing databases, or large amounts of data and finding useful relationships from implicit regularities. ML in Data Mining can be used to analyse outcomes from medical treatments with the help of patient databases, or helping to make weather predictions using data about weather behaviour in the past. ML is also used in domains where humans do not have the knowledge needed to develop useful algorithms, like face recognition (Swiniarski 2000), or in domains where a machine must

dynamically adapt to changing conditions, like in a manufacturing process with changing supply stocks (Agard & Kusiak 2005). In this work, one specific sub-discipline of ML is of great interest: *Inductive Logic Programming*(See Section 4.3), which is said to lie at the intersection of ML and *logic programming* (See Section 4.2) (Muggleton 1991).

4.2 Logic Programming

In a broad sense, *logic programming* (LP) is considered to be the use of mathematical logic for computer programming. One of the earliest uses of LP in this sense was John McCarthy's advice-taker proposal (McCarthy 1958), which suggested a hypothetical computer program that uses logic to represent data in a computer. In this sense of LP, logic is only used as a declarative representation language and the problem solver takes the form of a theorem-prover. Nowadays however, LP is commonly referred to as the use of first-order logic, not only as declarative representation but also as a procedural one. This is done by applying a backwards reasoning theorem-prover to declarative statements. For example the implication:

$$If\ A\ and\ B,\ then\ C$$

can be treated as a procedure:

$$in\ order\ to\ prove\ C,\ prove\ A\ and\ B$$

4.3 Inductive Logic Programming (ILP)

> "*Inductive Logic Programming* (ILP) is an approach to *Machine Learning* where definitions of relations, hypotheses, are induced from examples. In ILP, Predicate Logic is used as a language for defining hypotheses; therefore the result of learning is usually a formula in predicate logic." (Bratko 2001)

ILP is different from most ML approaches because of its expressive representation language. This enables the experts from any domain of application, who may know little about computer science, to provide the programmer with some relevant, known facts

about their domain. Such facts are called *Background Knowledge* (BK). The programmer can encode this BK in a way that the ILP system can understand, which in most cases means that the BK takes the form of logical predicates. These BK predicates are then made available to the ILP system prior to learning. Another advantage of this expressive representation language is that the same domain experts can make some sense of what has been learned by the ILP-system after induction.

For an ILP system to work, two main components are required; a set of *Background Knowledge* (as described above) and some *Training Data* (TD). The TD defines the problem that is to be solved. The TD consists of a set of *examples* that indicate what the intended program should do (positive examples) and, in most cases, counter examples that indicate what the program should not do (negative examples). The ILP system will then use the BK-predicates, combining them and putting them together to produce a set of rules which form a hypothesis that is able to solve the task defined by the TD. The resulting hypothesis can be considered as a program in itself. Different approaches have been developed specifically for learning tasks where negative examples are unavailable. One such approach, positive-only learning, is described in Section 4.4. Most ILP tasks are so complex that they require several hypotheses to be learned before a task can be solved with them. Such a collection of hypotheses is usually called a *theory* or a *model* (See Section 3.1). See Table 4.1 for a basic example on how ILP works.

4.4 Positive-Only Learning

Most traditional learning problems, for which ILP was designed, deal with positive and negative examples. Herein the aim of the learning tool is to find models that can differentiate between positive and negative examples (e.g.: a binary classification task). This is achieved by trying to induce a model that can describe as many of the positive examples as possible while describing as few of the negative examples as possible. There are however some domains where negative examples are very difficult or impossible to find. The most obvious of such domains is the learning of *natural grammars* in *Natural Language Processing* (NLP). A problem which is common to most NLP learning tasks is that there

Suppose we define the Background Knowledge (BK) to be the following relationships about family relations:

`parent(X,Y)` - X is a parent of Y
`male(X)` - X is male
`female(X)` - X is female

what we want the program to learn is the relationship

`has_daughter(X)`

We will provide the ILP tool with some correct training data defining some family relationships between different people using the BK predicates, then we specify two correct examples of the use of `has_daughter(X)` and two wrong examples. The ILP tool should then induce the following relation:

```
has_daughter(X):-
     parent(X,Y),
     female(Y).
```

Which means: X has a daughter when X is a parent of Y and Y is female.

Table 4.1: An example of how ILP can be used to learn a very simple relationship, taken from (Bratko 2001).

are no explicit negative examples available. In these cases, an ILP system would often induce models that are overly general. A possible way to overcome this problem is through the use of the Closed World Assumption (CWA) (See Appendix A.8) to produce implicit negative examples, but the CWA often does not apply. What is more, when learning natural grammars, the main focus is on positive examples, the legal sentences of the given language; we want our grammars to be able to parse legal sentences correctly. In this sense, NLP grammar learning is not a classification task as a grammar is not concerned with differentiating legal sentences from illegal ones, but instead, the aim is to find the correct parse for a given sentence. The situation that needs to be avoided is not that illegal sentences are wrongly classified as legal ones, but rather from the many possible parses, an incorrect one is chosen.

This problem led to the development of *positive-only learning*. Muggleton (Muggleton 1996) proposes a probabilistic method for learning from positive examples where higher probabilities are assigned to more compact models while still aiming to favour less general models. This method addresses a trade-off between the size and specificity of the induced model.

Another such domain where negatives examples are often unavailable is the learning of biological grammars from certain protein sequences which is the focus of this work (See Section 4.9).

4.5 Clause Evaluation Functions

During the learning process an ILP system searches through a search-space of clauses (logic rules) to find the one that solves the current task best. As the search space is explored, each clause encountered along the way needs to be evaluated. There are usually 3 different actions that can be taken once a clause is encountered. The clause can be determined to be completely unsuitable and as such it is unlikely that further exploration of that particular branch of the search-space leads to any suitable candidates. In this case, the branch is usually pruned from the search-space completely. The clause itself can be determined unsuitable, however there is the possibility that a good clause can be found further along the branch. The usual consequence here is that the clause is ignored and the search moves on. Finally, a clause can be suitable to solve the task. In this case there is a need to give a score to the clause, determining its quality. If the whole search-space is explored, or some other search parameters cause the search to stop, like a maximum number of nodes traversed, then the clause with the best given score is chosen as the outcome of the search. Assigning a score to an accepted clause is a very important step of the search algorithm as the outcome of the search greatly depends on it. There are many different ways to evaluate a clause and calculate a score. For instance, one could simply choose the clause that covers the most positive examples, however that is usually not enough. Equations that are used to calculate a score for a clause are called *clause evaluation functions* (Functions). They are one of the main aspects of this work (See Section 6.1).

In ILP, a number of clause evaluation functions have been used in the past. Suppose P stands for the number of positive examples covered by the clause, N stands for the number of negative examples covered by the clause and L is the number of atoms (predicates) in the clause itself. Some common clause evaluation functions are:

- *coverage*: $P - N$, a simple function which gives higher scores if more positive and less negative examples are covered: maximising P and minimising N.

- *compression*: $P - N - L$, which aims to give higher scores to clauses that are compressive. It maximises P, minimises N, and in cases where two rules cover the same number of P and N, the inclusion of L ensures that the shorter of the two rules is given the highest score. Two examples of related work using this function are: (Turcotte, Muggleton & Sternberg 2001a) and (Cootes, Muggleton & Sternberg 2003).

- *posonly*: $log(P) - log(\frac{R+1}{RSize+2}) - \frac{L+1}{P}$ (Muggleton 1996)

4.5.1 Compression Measures

The literature reviewed in Chapter 3 suggests that a very interesting criteria to consider, when evaluating candidate hypotheses, is compression (Muggleton, Srinivasan & Bain 1992), (Srinivasan, Muggleton & Bain 1994), (Kirchherr et al. 1997), (Stahl 1996). If there are two competing hypotheses, the one that best compresses the data, the training examples, is often the best. Tying in the model selection principles in Chapter 3 into the field of ML, we can use compression to decide between two competing models that have been learned using ILP. Clause evaluation functions that take compression into account are called Compression Measures, which are essentially operationalisations of the Minimum Description Length (MDL) principle (See Section 3.6).

4.6 ILP tools

There are many different ILP tools available, but in relation to this work, two stand out: Progol and Aleph.

4.6.1 ILP tool: Progol

Progol (Muggleton 1995) is an Inductive Logic Programming (ILP) tool which uses a covering approach. This means that an example to be generalised is selected and a consistent

clause is found that can cover (parse) the example. Any clauses made redundant by the found clause are removed from the theory. This process will repeat itself until all examples are covered or a given termination condition is reached. Progol uses a general-to-specific search to construct hypotheses. The search strategy is an A*-like algorithm (Hart, Nilsson & Raphael 1968) guided by an approximate compression measure. Each invocation of the search returns a clause which is guaranteed to maximally compress the data. The hypothesis language of Progol is restricted by the means of Background Knowledge and mode declarations provided by the user.

4.6.2 ILP tool: ALEPH

Aleph (**A** **L**earning **E**ngine for **P**roposing **H**ypotheses) (Srinivasan 1993) is another, more recent ILP system. It was initially intended to be a prototype for exploring ideas. In 1993, Ashwin Srinivasan and Rui Camacho at Oxford University produced earlier incarnations of Aleph under the name of P-Progol as part of a project in order to understand ideas of inverse entailment later published in (Muggleton 1995). Since then, the implementation has evolved to emulate some of the functionality of several other ILP systems. For a summary of Aleph's search algorithm see Table 5.6 on page 58.

4.6.3 Progol vs Aleph

This work deals with learning biological grammars (See Section 2.4), which means that it is very likely that a positive-only learning approach is used. The ILP tool used needs to be able to accommodate for this. Progol version 4.2 (Muggleton 1996) can learn from positive-only data. It was used in (Muggleton et al. 2001) which is the paper on which most of the initial work presented in this thesis was based (See Section 4.13.1). However the ILP tool Aleph has features which are very useful for our experiments. Among other things, Aleph provides accessible features that allow the user to implement his own evaluation function to control the search. This enables a user to write his own compression measure (See Section 4.5.1) and make Aleph use it, without needing to rewrite the tool itself. We therefore decided to focus on Aleph as the main ILP tool for our experiments.

4.7 Learning from protein data with ILP

There are several advantages in using ILP to learn from biological data:

- ILP allows a domain expert to guide the search through the search space by providing problem-specific background knowledge (BK),
- logic programs are used as a representation language for examples, BK and hypotheses, enabling users to integrate the learned results into future experiments or applications,
- the learned hypotheses can be made readable, simplifying their interpretation and further use in debates,
- when dealing with protein structures, which are the result of complex interactions between secondary structure elements, the ability to learn relations is crucial.

ILP has been successfully used to learn from protein data in the past. (Muggleton, King & Sternberg 1992a) applied the ILP tool Golem (Muggleton & Feng 1992) to the problem of learning secondary structure prediction rules for a particular type of proteins (α/α domain type proteins). Their findings outperformed the best previous result at the time for this α/α domain type. Later, the protein secondary structure prediction problem was also addressed by (Mozetic 1998) who implemented an ILP system called ALF, a general purpose system designed to learn from sequential data and whose advantage is that it can efficiently handle large quantities of data.

Proteins fold themselves according to certain constraints: either local signatures which depend on particular parts of a sequence or global signatures, which involve features along the entire chain. ILP has been used to discover such signatures for protein folds (Turcotte et al. 2001a), (Turcotte, Muggleton & Sternberg 2001b), (Cootes et al. 2003). (Turcotte et al. 2001a) have explored the use of ILP, through the ILP tool Progol, to search systematically for protein fold signatures and have thus shown that expert type rules can be learned from complex biological data. They also encourage the use of ILP for knowledge discovery in other areas of bioinformatics. (Turcotte et al. 2001b) have applied ILP to

the problem of constructing readable descriptions of protein folds. They also used Progol, but investigated the difference between using two different sets of background knowledge (BK): an attribute-valued set and a relational set. They showed that for learning the construction of protein fold descriptions, the use of the relational BK has demonstrable advantages: it increases predictive accuracy, it will outperform a propositional learning system and more explanatory insight can be achieved. They demonstrated the latter by having a domain expert validate the contents of the rules. (Cootes et al. 2003) have used ILP to automatically learn structural principles from a manually curated database. However they have also encouraged the use of the same approach towards other protein structure classification problems.

ILP has also been applied to functional genomics (a field of molecular biology that attempts to make use of the vast wealth of data produced by genomic projects). (King 2004) applied ILP to predicting gene function and describes several ILP-based approaches to the bioinformatics problem of predicting protein function from amino acid sequences. (King, Karwath, Clare & DeHaspe 2001) have used the ILP algorithm Warmr (Dehaspe, Toivonen & King 1998) in conjunction with other techniques to find frequent patterns in a dataset of biological sequences, aiming to predict the functional class of given proteins.

(King, Whelan, Jones, Reiser, C.H.Bryant, Muggleton, Kell & Oliver 2004) addressed the question whether it was possible to completely automate a scientific process. They implemented a closed-loop robotic system that is able to carry out cycles of scientific experimentation by applying techniques from artificial intelligence. Their system is able to generate hypotheses which explain given observations, devise experiments to test these hypotheses and finally run these proposed experiments using a laboratory robot. Then the system interprets the results, accepts or rejects the hypotheses and if necessary, repeats the whole cycle. There is no human intellectual input in neither the design of the experiments, or the interpretation of the results. Logical inferences were made by the ILP tool ASE-Progol (where ASE stands for Active Selection of Experiments).

The work conducted in this thesis is related to the work discussed in this section. We are also applying ILP to a learning problem concerned with protein data. We use ILP to

learn biological grammars describing neuropeptide precurser proteins, a specific protein family.

4.8 Grammar Learning with ILP

ILP has been used successfully in the past to learn grammars or closely related syntactical structures.

Zelle and Mooney (Zelle & Mooney 1993) have shown that grammar acquisition can be considered as learning control rules for a logic program, therefore the problem can be addressed by ILP. They introduced a new induction algorithm, incorporating constructive induction (which introduces new terms into the learners vocabulary) to learn word classes and semantic relations. Finally, they presented a system which uses ILP to learn a shift-reduce parser (Tomita 1985) . The parsers learned by their system proved to be accurate and generalize well to novel sentences.

Pullman and Cussens (Cussens & Pulman 2000) (Pulman & Cussens 2001) show how ILP can be applied to grammar learning. They used an inductive chart parsing approach to learn missing rules for an incomplete grammar (a grammar is considered incomplete if it is unable to parse the entire provided corpus). However their system was not fully unsupervised and was relying on a linguist to test and further develop the induced rules.

Charniak (Charniak 1996) obtained a tree bank grammar from a tree bank. A tree bank is a text corpus that contains sentences which have been fully annotated with syntactic structure (Wallis 2008). A tree bank grammar does not look at the actual words in the sentences, but rather at the annotations of the syntactic structure. A tree bank grammar is a probabilistic context free grammar, a context free grammar where each rule has an assigned probability based on how often it is used in the training data. Their grammar outperformed other non-word based grammars or parsers. Their results, among other things, motivated De Raedt et al. (De Raedt, Kersting & Torge 2005) to propose a framework for probabilistic inductive logic programming. Their training data consisted of proof trees.

The work conducted in this thesis is related to the work discussed in this section. We

are also applying ILP to a grammar learning problem. Specifically, we are concerned with using ILP to learn biological grammars, in the form of Definite Clause Grammars, from protein sequences.

4.9 Biological Grammar Learning with ILP

As mentioned in Section 2.5, good results have been achieved in the recognition of complex biological signals using linguistic approaches (Searls 1997, Searls 2002). However the necessary biological grammars have to be constructed first. If experts were to do this manually, it would be very expensive and time-consuming, which justifies the need for an automatic process to learn these biological grammars. ILP (See Section 4.3) shows great potential to help automate this grammar learning process. The use of ILP in this application domain has two main advantages: first, the result of an ILP learning task is a set of logic programs which can easily be used to represent grammars, so ILP is able to learn grammars, therefore it is able to learn biological grammars just as well; second, through extensive use of BK an ILP tool can actually take into account knowledge that is already known to the experts in the field (Biologists).

Muggleton et al (Muggleton et al. 2001) first investigated Chomsky-like grammar representations (See Section 2.3.1) for learning cost-effective, comprehensible predictors of members of biological sequence families. They used the ILP tool CProgol (Muggleton 1995) (See Section 4.6.1) to learn grammars describing neuropeptide precursors (NPPs). Their best predictor made the search for novel NPPs more than one hundred times more efficient than randomly selecting proteins for synthesis and testing them for biological activity. Their work was also one of the first real-world scientific applications of using ILP to learn from positive-only examples only.

4.10 Representation of Biological Grammars and Sequences in ILP

All input given to the ILP tools Progol and Aleph conforms to the Prolog syntax. Often, the same also applies to the output. This can be very beneficial as induced models might be used in further tests and experiments using ILP tools. Previous related work in this area (Muggleton et al. 2001, Bryant & Fredouille 2005, Fredouille, Bryant, Jayawickreme, Jupe & Topp 2006, Bryant, Fredouille, Wilson, Jayawickreme, Jupe & Topp 2006) was concerned with learning Biological Grammars in the form of context free grammars. These grammars were represented using a Definite Clause Grammar (DCG) formalism (Pereira & Warren 1986). DCGs require sequences to be represented by a list, where each element in this list represents a letter in the sequence. DCG rules take such a list as input and pass it on to the predicates that make up the rule. Each predicate, starting with the first, then matches one or more elements from the start of the list and returns the rest. This new, shorter list is then in turn given to the next predicate in the rule. If the last predicate returns an empty list, then the whole sequence is matched by the grammar rule and we consider the sequence to be covered by that rule.

4.10.1 Revisiting Kolmogorov Complexity

Suppose we have learned a grammar, consisting of a set of grammar-rules, which all cover one or more biological sequences of the target family. This grammar, combined with a sequential selection of grammar production rules is all that is needed to produce a computer program that generates the initial biological sequences. The shorter this program is, the more compressible the information about the respective family of biological sequences is. Thus we are seeking the shortest such program(s), whose length is the Kolmogorov Complexity of the given biological sequences.

4.11 Neuropeptide Precurser Proteins (NPP)

In most of our experiments, we are learning from biological sequences taken from a specific family: neuropeptide precurser proteins (NPP).

The purpose of most drugs is to modulate the actions of protein molecules, mostly by suppressing undesirable biochemical reactions. This is achieved by interactions between the drug molecules and the target protein molecules, either enzymes or receptors. However, an understanding of the biology of neuropeptides and their interactions with receptors is in many cases still elusive. As a result of this, the pharmaceutical industry is very interested in novel neuropeptides and their receptors. Neuropeptide sequences are subsequences of neuropeptide precursor sequences. A precursor can contain one or more identical or different neuropeptides. If there is more than one neuropeptide in a precursor then they can, but do not necessarily have to, be separated by *filler* peptide. It is assumed, but not proven, that filler peptide merely plays a structural role. Neuropeptide precursors also contain a *signal* peptide, a short prefix of residues roughly 20-30 amino acids in length. The whole precursor peptides can be 70-600 amino acids long. It is this large variation in the length of neuropeptide precursors and their complex internal organization that makes it difficult to use sequence database searching methods like BLAST (Altschul, Gish, Miller, Myers & Lipman 1990) or multiple alignment methods (Carrillo & Lipman 1988) to search for novel neuropeptide precursors. These difficulties further provide a rationale for using ILP to address this task.

4.12 Proposed Experiments

Most clause evaluation functions ignore the lengths of examples and instead only count the number of positive, negative or random examples covered by a hypothesis. This is not a problem in most traditional ILP learning tasks, where the length of all examples is similar, but the lengths of protein sequences are highly variable (e.g. neuropeptide precursor strings vary from 70 to 6000 amino acids (Muggleton et al. 2001)). Thus the main rationale of the practical work in this thesis is the following:

> We suggest that such strong variations should not be ignored by clause evaluation functions when learning biological grammars.

Our work has its roots in the approach of (Muggleton et al. 2001) as we use *definite clause grammars* DCG (Pereira & Warren 1986) (See Section 4.10) and we also use several subsets of the NPP dataset used in (Muggleton et al. 2001)(See Section 5.4). In (Muggleton et al. 2001), the ILP tool that was used was Progol (See Section 4.6.1). The subsequent papers by Bryant et al. (Bryant & Fredouille 2005, Fredouille et al. 2006, Bryant et al. 2006) also take their roots in the approach of (Muggleton et al. 2001). They used the ILP tool Aleph (See Section 4.6.2), which is easier to modify because of its modular system. Our work also uses Aleph as we also need the ability to customize the clause evaluation functions and the clause coverage computation (See Section 4.5).

In our experiments, ILP will be used to learn biological grammars. In ILP, logic programs are often used to represent background knowledge (BK), examples and hypotheses. As grammars can be represented as logic programs, ILP can be used to learn grammars. The BK is encoded expert-knowledge and the examples are biological sequences.

Before we can get into the details of the experiments described in the following chapter of this thesis, some preliminary experiments need to be conducted.

4.13 First Experiments

At the beginning of this work, the task was to gain familiarity with some of the contemporary ILP tools Progol and Aleph. Therefore, before we changed some parts of the systems, we first tried to reproduce some previous experiments.

In the following Sections (4.13.1, 4.13.2 and 4.13.3) some elements of Aleph or our dataset are briefly referred to, as a brief introduction to the practical work. The next chapter of this thesis will go into further detail, describing the datasets, their origin and all relevant parameters used in our work in detail.

```
end(A,B) :- large(A,C), l(C,D), small(D,B).
end(A,B) :- very_hydrophobic(A,C),hydrophobic(C,D),
            positive(D,B).
end(A,B) :- tiny(A,C), hydrophilic(C,D), positive(D,B).
end(A,B) :- pp(A,C), large(C,B).
end(A,B) :- pp(A,C), star(C,B).
end(A,B) :- large(A,C), v(C,D), v(D,B).
end(A,B) :- hydrophilic(A,C), a(C,D), tiny(D,B).
end(A,B) :- hydro_b_don(A,C), s(C,D), hydro_b_acc(D,B).
end(A,B) :- g(A,C), positive(C,D), hydro_b_don(D,B).
end(A,B) :- very_hydrophobic(A,C), n(C,D), t(D,B).
end(A,B) :- t(A,C), q(C,D), t(D,B).
```

Table 4.2: A grammar for NPP end sequences, consisting of 11 Production rules generated by CProgol.

4.13.1 Reproducing Progol Experiments

The first set of experiments consisted of using the CProgol system to learn a subset of the biological grammars that were learned in (Muggleton et al. 2001). We decided to generate the grammars that cover the **end**s of the Neuropeptide Precurser Proteins (NPP). We were provided with some input files for Progol that were related to those that were used in (Muggleton et al. 2001). The aims of our first set of experiments was to get familiar with ILP tools, understand the finer details of the different input files that we have to provide the system with, observe and understand the consequences of tweaking the parameters used, and finally, produce some sensible results. As described in Section 5.5, NPPs consist of, among other things, a start, a middle and an end. As this was our first batch of experiments, we chose the NPP subset that seemed easiest to learn: the **end**s. Every NPP-end consists of exactly 3 amino acids, which means that any grammar rules covering them contain at most 3 predicates, none of which are gaps (See Section 5.2.1). This greatly decreases the work (processing power) required to learn these grammars, in comparison to the NPP middles for example.

We compiled Background Knowledge, positive and random examples (examples where it is unknown if they are positive or negative, see Section 4.4), mode declarations, pruning predicates and type declarations. During these preliminary experiments, a grammar for the ends of NPPs were learned, consisting of 11 clauses, listed in Table 4.2. The specifics

on how to use an ILP system are further explained in Chapter 5.

4.13.2 Translating from Progol into Aleph

After the experiments were successfully reproduced using Progol, the same experiment was also reproduced using Aleph. Aleph provides more options which simplify potential future changes to the system. However, Aleph uses a slightly different structure and syntax for its input files, therefore the first challenge was to translate the Progol input files into Aleph input files.

The output of these experiments proved to be different from the output grammars of the Progol experiments of Section 4.13.1. This was expected. As Aleph follows different learning strategies than Progol does, the outcomes are not identical.

4.13.3 Using Aleph to learn on more complex protein sequences

After successfully translating the task of learning NPP-ends from Progol into Aleph, we can proceed to learning more complex grammars using Aleph. The NPP-ends which were learned from in Sections 4.13.1 and 4.13.2 are just a small part of the entire NPP dataset. They are the shortest sub-sequences in the dataset and thus, the easiest to learn grammars from. The most challenging subset of the NPP sequences are the NPP-middles (See Section 5.5). Therefore the next step is to use Aleph and learn grammars for the NPP-middles using the standard approach. The input files now have to be changed in several ways to incorporate different pruning predicates and add additional Background predicates (like the Gap predicate that is explained in detail in Section 5.2.1). These experiments are described in Chapter 6 and 7.

4.14 Summary

In this chapter, we introduced *Machine Learning* and one of its sub-disciplines, *ILP*. We explained what is meant by *positive-only learning* and why this is relevant to our problem of learning biological grammars. We have explained what a clause evaluation functions

are and how the MDL principle can be applied to them. We have introduced two ILP tools, Progol and Aleph, compared them against each other and gave reasons why we prefer Aleph over Progol in this work. We have given a review on related work dealing with using ILP to learn from protein data, using ILP to learn grammars and finally, using ILP to learn biological grammars. We have introduced the family of proteins that we will be learning from in our experiments, the neuropeptide precurser proteins (NPP). Finally, we have proposed a set of experiments and even ran some preliminary experiments using our chosen ILP tools and datasets.

Now, before we can move on to describing the main experiments in Chapter 6 and 7, we first need to introduce all the materials and performance measures that will be used in this thesis.

Chapter 5

Experimental Setup

This chapter will introduce all the material that is required to run our experiments. The experiments themselves are briefly mentioned at the end of Chapter 4 and explained in further detail in the following Chapter 6. This chapter also serves in further tying in the concepts and literature that were reviewed in Chapters 2, 3 and 4. Finally, this chapter leads on to the main experiments that were conducted in this work, which are described in Chapter 6.

In our experiments we use the ILP tool Aleph (Srinivasan 1993) to learn Biological Grammars (See Section 2.4), therefore we need to provide the system with a number of different materials. This chapter will explain how all these are compiled: Background Knowledge (Section 5.2); Pruning and Constraint predicates which are controlling the search (Section 5.3); and the datasets (Sections 5.4 - 5.6). Our testing strategies are explained in Section 5.7. Also, some further issues are discussed: which grammar evaluation function to use (Section 5.8) and how to deal with Aleph's coverage calculation (Section 5.9)

5.1 Problem statement

Most of the experiments referred to in this work deal with learning context free grammars that describe a specific family of proteins called *Neuropeptide Precurser Proteins* (NPP).

Physicochemical Property	Amino acids
Hydrophobic	H,W,Y,F,M,L,I,V,C,A,G,T,K
Very hydrophobic	A,F,G,I,L,M,V
Hydrophilic	S,E,Q,R,D,N
Electropositive	R,K,H
Electronegative	D,E
Neutral	A,C,F,G,I,L,M,N,P,Q,S,T,V,W,Y
Large	Q,E,R,K,H,W,Y,F,M,L,I
Small	P,V,C,A,G,T,S,N,D
Tiny	A,G,S
Polar	Y,T,S,N,D,E,Q,R,K,H,W
Aliphatic	L,I,V
Aromatic	H,W,Y,F
Hydrogen donor	W,Y,H,T,K,C,S,N,Q,R
Hydrogen acceptor	Y,T,C,S,D,E,N,Q

Table 5.1: Left column: physicochemical properties; right column: all the amino acids that exhibit the physicochemical property in the left column.

We first conduct the basic experiments using an unaltered ILP system and use the results achieved as a benchmark (See Section 4.13). Then we apply some of the principles discussed in Chapter 3 and attempt to improve the results achieved. We aim to learn grammars with better performance than those learned by the benchmark experiment and ideally we would want the process of learning to be more efficient as well.

5.2 Background Knowledge (BK)

We used the same BK in almost all our experiments. It consists of general molecular biology knowledge which can be considered relevant for any protein grammar inference process. The BK contains amino acid letters and their physicochemical properties (as first proposed by (Muggleton, King & Sternberg 1992b), also used in several subsequent works, (Muggleton et al. 2001), (Fredouille et al. 2006) and (Bryant & Fredouille 2005) among others) and *gaps* (See Section 5.2.1 about gaps). Most of the information contained in the BK can be found in Table 5.1. As Aleph expects input in the form of Prolog statements the information in Table 5.1 was rewritten as in the following example:

$$hydrophobic([h|T], T). \tag{5.1}$$

The above predicate logic statement (5.1) tells the system that the first character in the parsed sequence is the amino acid h, which is hydrophobic (row 1, first item in column 2 from Table 5.1). The first argument of the predicate is the given sequence. The notation of $[h|T]$ splits the sequence in two: the first character is matched to h and the rest of the sequence is matched to the variable T. T, as second argument of the predicate, is then processed further. This predicate effectively splits away the first character of a given sequence, **IF** that character happens to be an h, and then passes on the remainder.

5.2.1 Gaps

Protein sequences contain certain parts which are not directly relevant to the function of the protein or which cannot be properly characterized by the predicates making up the BK. These parts however still participate in the overall structure of the molecule (Bryant et al. 2006). *Gaps* are the predicates that are used to match such parts. In this work we consider gaps to be unlimited. This means that one gap can match any sequence of amino acids, independent of the length. The predicates defining gaps, are provided to Aleph as part of the BK and are defined as follows:

$$gap(S, S). \tag{5.2}$$

$$gap([\_|S], T) :- gap(S, T). \tag{5.3}$$

Predicate (5.2) matches the first variable S with the second variable, thus returning exactly the same sequence as was passed to it. This predicate on its own is undesirable, but it works in conjunction with Predicate (5.3).

Predicate (5.3) has as its first argument $[\_|S]$ a sequence of characters, which it then splits in two: " _ " is a wild-card and matches any single character, while the remainder of the sequence is stored in variable S. " $:-$ " denotes an implication from left to right. The predicate to the right of the implication receives S, the remainder of the given sequence and returns the variable T, which is then matched with the corresponding variable T in the head predicate, which marks the output of the predicate. As the name of the predicate

on the right side of the implication is the same as that on the left side, this predicate will call itself recursively, until the predicate 5.2 is successful and stops the process.

5.3 Controlling the Search

In our experiments, we want to learn a context free grammar (See Section 2.3.1). To make sure that any output we get is in this desired form, we can introduce several biases in order to control the search. Using such biases also speeds up the search significantly, as many branches of the search tree, that would never lead to an acceptable result, are not explored at all. Aleph (and many other ILP tools) offers two main features for controlling the search: *pruning predicates* and *integrity constraints*.

5.3.1 Pruning Predicates

Pruning predicates allow the user to introduce a bias into how the search traverses a search tree. When performing a search for a clause, Aleph allows for two different sorts of pruning. There are built-in pruning predicates, referred to as internal pruning, that perform admissible removal of clauses from a search. But Aleph also allows for user-defined pruning predicates. These are predicate logic statements that the user can provide to the system which specify certain conditions. If we know with certainty that a clause, or any of its further refinements, which satisfies one or more conditions, should never be accepted as a candidate hypothesis, then we can encode these conditions into pruning predicates. Any hypothesis satisfying any of the provided pruning predicates is pruned from the search. This means that the clause is not considered as a solution, nor is it further refined.

In our experiments we provide the Aleph system with three pruning predicates, which are listed in Table 5.2. The purpose of the pruning predicates seen in Table 5.2a is to make sure that the hypotheses form a variable chain from the head.

$$E.g.: middle(A, B) :- yvh(A, C), gnt(C, D), fnt(D, E), gap(E, B).$$

The first variable in the head, A matches the first variable in the first predicate in the body. Then the second variable in the first predicate in the body, C, matches the first variable in the second predicate of the body. This continues all the way until the last predicate in the body, whose last variable matches the last variable in the head, B, in turn. The purpose of the pruning predicate seen in Table 5.2b is to make sure no gap is of the form $gap(A, A)$ (See Chapter 5.2.1 for more information on gaps). If such a gap were allowed, it would not cover anything. Even though gaps can cover any number of letters, we want them to cover at least one. The purpose of the pruning predicate seen in Table 5.2c is to make sure no two gaps follow each other. This would be pointless as a gap is a wildcard and can cover many letters anyway, so two following gaps would be the same as just one gap, therefore one gap would be redundant and would just increase the complexity of a clause unnecessarily.

a) This pruning predicate ensures that the body of the given clause forms a variable chain from the head.

$prune((Head :- Body)) :- Head = ..[\_, U, \_], not(chain(U, Body))$.

b) This pruning predicate ensures that no gap is of the form gap(A,A).

$prune((\_ :- Body)) :- in(gap(A, B), Body), A == B$.

c) This pruning predicate ensures there are no two gaps immediately following each other in the body.

$prune((\_ :- Body)) :- in(gap(\_, B), Body), in(gap(C, \_), Body),\ B == C$.

d) Support predicates: These predicates are needed for the above to work.

$chain(\_, true)$.
$chain(U, A) :- A = ..[\_, V, \_], U == V$.
$chain(U, (A, B)) :- A = ..[\_, V, W], U == V, chain(W, B)$.

$member(E, [E|\_])$.
$member(E, [\_|L]) :- member(E, L)$.

$in(Pred, Term) :- nonvar(Term), Pred = Term$.
$in(Pred, Term) :- nonvar(Term),$
$\quad Term = ..[\_|TArgs],$
$\quad member(A, TArgs),$
$\quad in(Pred, A)$.

Table 5.2: Pruning predicates provided to Aleph in our NPP experiments.

5.3.2 Mode Declarations

Both ILP tools Aleph and Progol use **mode declarations** to define which predicates can be used to construct hypotheses and how exactly they should be used. If one considers the BK predicates to be building blocks for hypotheses, then the mode declarations dictate how these blocks can be put together. For each of the provided predicates, mode declarations define which of its arguments should be used as input, output or if it should contain a constant. Mode declarations also define if a given predicate should be used in the head or in the body of a hypothesis. A declaration defining a predicate to be the head of a hypothesis usually means that this is the hypothesis that should be learned by the ILP tool. Depending on which ILP tool is used, mode declarations can contain even more constraints, like a *recall number* which can be used to bound the number of alternative solutions for instantiating the atom.

5.3.3 Type Declarations

The **type declarations** specify the *type* of the arguments of the clauses that are to be learned. Any predicate can be used as a type. The search does not consider arguments for its hypotheses which do not satisfy the predicates set as argument types.

5.4 NPP Data Set

All experiments described in this chapter use a dataset made up of Neuropeptide Precursor Protein sequences (NPPs). The original NPP dataset was compiled in (Muggleton et al. 2001) and all the sequences used were taken from *SWISS-PROT* (Bairoch & Apweiler 1997), a manually curated biological database of protein sequences. SWISS-PROT aimed to provide reliable protein sequences associated with a high level of annotation (function and structure of a protein, variants etc.), a minimal level of redundancy and high level of integration with other databases. SWISS-PROT is now included, as one out of two sections, in UniProtKB (UniProtKB n.d.). This NPP dataset consists of a set of positive examples and a set of randomly selected sequences. As, at the time of the compilation

of this dataset, there were still many undiscovered NPP proteins, it was not possible to come up with a large enough unbiased set of negative examples.

5.4.1 Positive Examples

The set of NPP-Positives contains all the 44 NPP sequences that were known when the dataset used in (Muggleton et al. 2001) was compiled (spring 1997). 41 of these NPPs are human proteins. The remaining 3 were not human proteins, but it was expected that they were closely related to their (at the time still undiscovered) human counterpart and thus they were included in the dataset as well.

5.4.2 Random Examples

The set of NPP-Randoms contains 3910 human protein sequences. These were all the human sequences that were in SWISS-PROT at the time this dataset was compiled (spring 1997).

5.5 NPP-middles

The NPP dataset described in Section 5.4 contains complete NPP sequences, however in our experiments we chose to focus on a subset of the NPP dataset. As described in Section 4.11, within a NPP there can be one or more neuropeptides. The NPP dataset used in (Muggleton et al. 2001) had neuropeptides divided in 3 parts: *start*, *middle* and *end*. A neuropeptide start is a list of two letters, the first two amino acids in the neuropeptide sequence and a neuropeptide end is a list of three letters, the last three amino acids in the neuropeptide sequence. The middle of a neuropeptide contains everything that can be found between a start and an end. Considering that the start and end have a fixed length, they are of little interest to us and unsuitable for our experiments. The middles however have a large variation in their length, they range from 5 to 95 amino acids, which is the highest variation in the length of sequences that can be found in all the subsets of the NPP dataset. This makes the middles the most challenging to learn and the most

relevant to our work. So finally, the training data we use in our experiments contains 76 examples of NP-middle sequences, which we henceforth denote as *NPP-middles*.

5.6 NPP-middles - Examples grouped by length

In the course of our experiments we hypothesised that the behaviour of the system changes if only examples of a certain length range are provided. For the purpose of investigating this we decided to split up the NPP-middles dataset (See Section 5.5) into several parts based on the length of examples. This allows us to run separate experiments using separate sub-parts of the complete NPP-middles dataset. We took the NPP-middles dataset and split it into three disjoint subsets, based on the length of positive examples, effectively creating three separate datasets. We denote these three subsets as follows: *NPP-middles-short*, *NPP-middles-medium* and *NPP-middles-long*. The intention was to split the set of positive examples into three sets containing more or less the same number of examples. The first subset, NPP-middles-short, contains 24 examples, each of length (number of characters) $l < 13$, the second subset, NPP-middles-medium contains 26 examples with $13 \leq l \leq 29$ and the third subset, NPP-long contains 26 examples with $l > 29$. The NPP-middles-short subset contains only 24 examples (instead of 25 or 26 as seems intuitive, given that we have 76 total positive examples) because we set a length threshold of length 13, in order to prevent examples of the same length being found in two different subsets.

The random examples were split up in the same way according to the length of their examples, using the same cut-off values that were used for the positive examples. This results in NPP-middles-short containing 908 random examples, NPP-middles-medium containing 864 random examples and NPP-middles-long containing 1136 random examples.

The range of the length of examples within the datasets is as follows: NPP-middles-short contains positive examples that are between 4 and 13 characters long, giving us a range of 9 characters. The positive examples within NPP-middles-medium are between 13 and 29 characters long, giving a range of 16 characters. Finally, the NPP-middles-long contains examples that contain from 30, up to 94 characters, thus giving a range of 64. The range of the NPP-middles-long is a lot higher, as the size of each individual example

Dataset	length (l)	positive		random	
		No.	range	No.	range
NPP-middles-short	$l < 13$	24	9	908	9
NPP-middles-medium	$13 \leq l \leq 29$	26	16	866	16
NPP-middles-long	$l > 29$	26	64	1136	65

Table 5.3: Summary of the three datasets described in Section 5.6.

increases drastically once it surpasses 40 characters. The same values apply for the random examples in all three datasets, except that the longest random example is 95 characters long. For easier reference, a summary of the datasets described in this section can be found in Table 5.3.

5.7 Testing

To be able to test the biological grammars that will be learned, we applied a 5-fold stratified cross-validation strategy to each of the NPP-datasets we used. This means that for each set of examples, the positive and random examples, and for each of their subsets, as mentioned in Section 5.6, the given set of examples was split in five partitions of approximately equal size and each is used in turn for testing and the remainder is used for learning. On each turn, four fifths of the data is used for learning and one fifth is used for testing. In the end, every example is used exactly once for testing.

To prevent the case that examples of certain criteria are overrepresented in the test set, there is a need for stratification. This means that the sets of examples should be divided in such a way that it is guaranteed that examples of all different criteria (or at least of those criteria that matter to the task at hand) are properly represented in both training and test examples. In this work, the stratification of the cross validation was based on the length of the examples, ensuring that all training and test sets include a variety of examples of different lengths. The stratification of positive examples and random examples was performed separately, ensuring that both positive and random examples are evenly spread across all five folds.

Let TP stand for true positives, FP for false positives,
TN for true negatives and FN for false negatives.

a) The *accuracy* is the proportion of correctly predicted examples ($TP + TN$) among all predictions:

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

b) *Precision* is the ratio of predicted positive examples ($TP + FP$) that are indeed true positives (TP) and *recall* is the ratio of true positives (TP) among all positives ($TP + FN$):

$$precision = \frac{TP}{TP+FP} \quad ; \quad recall = \frac{TP}{TP+FN}$$

c) *F-measure* is the weighted harmonic mean of precision and recall:

$$F - measure = \frac{2 \cdot precision \cdot recall}{precision + recall}$$

Table 5.4: Definitions of predictive accuracy (a), precision & recall (b) and F-measure.

5.8 Evaluating Grammar Performance

Some tasks in Machine Learning deal with highly unbalanced datasets, which means that the ratio between positive examples and negative (or random) examples is not the same. In most of these special cases, there are only very few positive examples compared to a very high number of negative examples. When dealing with such unbalanced datasets, the traditional way of evaluating the performance of learned models in ILP, which is the *accuracy* (See Table 5.4a) is not appropriate. Three measures which find their root in the field of Information Retrieval (IR) have become popular even beyond their original domain (e.g. information extraction). These are: *precision*, *recall* and their weighted harmonic mean, the *F-measure* (See Table 5.4b and c).

Consider a case where we are trying to find a model for a given class and there are only a small number of positive examples available, but a very large number of negative examples. If we use the accuracy to judge the performance of a model learned from, and tested on such an imbalanced dataset, then we run into some problems. For example, if our machine learning tool learns a model that simply classifies each example encountered in the test set

Model	TP	FN	TN	FP	Acc	Rec	Prec	F
A	30	20	950	50	0.933	0.6	0.375	0.461
B	50	0	900	100	0.905	1	0.333	0.5

(Acc = Accuracy; Rec = Recall, Prec = Precision, F = F-measure)

Table 5.5: Example of two models competing with each other.

as negative, such a model would have a very high accuracy indeed, as most of the examples are classified correctly. However, we are not interested in a model describing the negative examples, but rather in a model that can describe positive examples. What is desired is a model that accepts a high proportion of the positive examples and a low proportion (preferably none) of the negative examples. The emphasis here is on the positive examples. The rejection of negative examples, although still desirable, is only secondary. Considering such unbalanced datasets, the variable that has the largest influence on the outcome of the accuracy formula in Table 5.4a is indeed TN the true negatives. However the equations for precision, recall and F-measure in Table 5.4b and c do not even include TN, so the outcome cannot be biased by TN.

For a better understanding, consider the following example: Suppose we have an imbalanced dataset consisting of 50 positive examples and 1000 negative examples, and suppose we have two models A and B of different coverage as given in the Table 5.5. A decision must be made as to which of these two models is the better. After calculating the accuracy, it would appear that Model A is the best. However, as mentioned above, we are more interested in covering as many of the positive examples as we are in classifying the negative ones, therefore the accuracy is misleading. Model B, which covers a lot more positives than Model A, is closer to what we are looking for than Model A, so we would expect our performance measure to give a higher score to Model B. The recall of Model B is higher than that of model A, and so is its F-measure. The precision is slightly lower, which is a consequence of the FP for Model B being higher.

We conclude that the F-measure is the more appropriate performance measure for a model which is learned on this type of imbalanced dataset. The dataset we use in this work (See Section 5.5) consists of 76 positive examples and 3910 random examples and as such, clearly falls into this category of unbalanced datasets. We have therefore decided to

use the F-measure as the main performance measure in this work. We will also monitor the accuracy as well as recall and precision.

When evaluating results, it should be taken into account that precision and recall are not always equally relevant. Consider that we are dealing with a positive-only dataset and that the fundamental idea therein is that not all the random examples can be said to be negative. In fact, it is assumed that some of those 3910 random examples are in fact unknown positive examples. Therefore, when referring to FPs it should be taken into account that this number might include unknown TPs. As a consequence, recall should carry more weight when evaluating grammars than precision, as precision depends on the number of FPs, whereas recall does not.

Other authors in the ILP community which set out to learn hypotheses from unbalanced datasets have also come to the conclusion that predictive accuracy is not always the best and only performance measure. For example: (Goadrich, Oliphant & Shavlik 2004) used ILP for Information extraction, which is a domain that usually deals with a very large amount of negative examples and a comparably very low amount of positive examples. In that work, precision, recall and F-Measure were used. (Turcotte et al. 2001a) and (Cootes et al. 2003) were applying ILP to learning from protein data and used precision and recall, alongside the predictive accuracy to evaluate performance of their rules.

5.9 Modification of Coverage Computation

As explained in Section 5.1, the main focus of this work is how an ILP system copes with the large difference in length of protein sequences. Aleph, the ILP system we decided to use, does not consider the length of the examples, our protein sequences, at all. However in order to consider the MDL principle, as explained in Section 3.6, our system needs to be aware of the length of the examples. Of course, the reason we decided to use Aleph in the first place was to have a system that is easily modifiable to suit our needs. At this point, this means that we need to modify Aleph in such a way that it is able to take into account the length of each individual example that it encounters. Over the course of this work, several approaches have been considered to enable Aleph to solve this task.

```
1. Select a positive example to be generalized.
    If none exist, stop.
2. Build the most specific clause entailing the example selected
    (the bottom clause)
3. Search; Find a clause more general than the bottom clause.
    (Construct a search tree, each node containing a clause
    consisting of a subset of the literals in the bottom clause.
    Search for the clause with the best score)
4. The clause with the best score is added to the grammar.
5. Remove all examples that were made redundant.
6. Return to Step 1.
```

Table 5.6: A simplification of the basic Aleph algorithm.

The first approach considered was to make Aleph calculate the coverage we required during its learning process. Aleph is keeping track of the coverage of the clauses it builds anyway, through a data structure called *Labels*. A *label* in Aleph is a data structure consisting of nested lists of the form $[P, N, R]$, where P is a list of all ranges of positive examples covered by the clause and N and R do the same for negative and random examples respectively. A *range of examples* contains the clause's coverage thus far, in the form of either ID-numbers of individual examples or of a range of ID-numbers (e.g. *5-8*). However attempting any modification of this data structure and how it is used and modified by Aleph led to some unforeseen consequences during Aleph's runtime. Another downside of this approach was a severe increase in time needed by Aleph to complete a learning process, as the additional functionality included consisted of a number of predicates which were being called extremely often during the learning process.

The second approach that was explored was to just use the clause and calculate our own coverage whenever it is needed. What this means is that whenever Aleph calls the clause evaluation function, our own custom function then calculates the coverage as we need it. Our predicates are provided with the current clause to be evaluated, which is one of the parameters given to the clause evaluation function anyway. Then our predicates extract the given examples of protein sequences from the input files and parse each of them using the clause to be evaluated. We can observe the output and thus, are able to find out which example was covered by the current clause. Once we know that, we can take

into account the length of the covered examples to calculate our own coverage. However, our tests revealed a discrepancy in the number of examples covered by the clause. Our predicates would often cover more examples than Aleph's built-in functions. The origin of this problem lies in the way Aleph was calculating coverage, which we had not considered before: Aleph applied a cover-set search algorithm. See Table 5.6 for a simplified version of Aleph's search algorithm. When using this algorithm, once a clause has been accepted by the search, all examples that are covered by that clause are removed from the set of examples to be learned on. Therefore simply calculating our score on all examples in the given set, without considering that some should have been removed, is not acceptable. We had to find a way to just consider the examples that were still unremoved (not yet redundant).

Further investigation of the Aleph source code finally gave us the answer. There is one global variable inside Aleph which keeps track of all the examples that are still left in the set to be considered:

$$'\$aleph\_global'(atoms\_left, atoms\_left(Type, Left)),$$

which can give us all examples of a certain type (*Type*) that are still left (as *Left*). Using this global variable we were finally able to calculate our own coverage value that still respected the way Aleph was conducting its search. Now that we were able to get a list of all examples that were still left to cover, we could write our own predicate that would parse all remaining examples through the clause that is to be evaluated, and for those that were covered, we could investigate their length and use that information to calculate our own coverage. Some of the predicates we wrote to calculate this modified coverage can be seen in Table 5.7.

5.10 Summary

In this chapter, we have introduced all the items that we need to run our experiments, including our test strategy. We have also outlined how the chosen ILP tool Aleph can be

changed in order to incorporate our proposed ideas, enabling us to conduct the experiments necessary to test our hypotheses. We can now move on to the next chapter, which describes the details of the experiments that were conducted and the results that were achieved.

a) **compute_Lmodcover/3** - This predicate computes the modified coverage of a clause on all remaining examples of a certain type.

```
%used as:compute_Lmodcover(Type,Clause, L-modifiedCoverage)
 compute_Lmodcover(Type,(Head:-Body), Cov) :- !,
    'aleph_global'(atoms_left,atoms_left(Type,Left)),
    compute_Lmodcover(Type, (Head:-Body), Left, 0, Cov).
 compute_Lmodcover(Type, ClauseWithoutBody, Cov) :-
    compute_Lmodcover(Type,(ClauseWithoutBody:-true),Cov).

 compute_Lmodcover(_, _, [], Cov, Cov).
 compute_Lmodcover(Type, Clause, [Inter|Rest], Cov,
                   CovRes) :-
    compute_Lmodcover_interval(Type, Clause, Inter,
                   Cov, Cov1),
    compute_Lmodcover(Type, Clause, Rest,Cov1,CovRes).

 compute_Lmodcover_interval(_, _, Start-Finish, Cov, Cov) :-
    Start > Finish, !.
 compute_Lmodcover_interval(Type,Clause,Start-Finish,
                   Cov,CovRes) :-
    example(Start, Type, Atom),
%get the L-modified coverage SeqLength for this example:
    parse_exple_for_length(Clause, Atom, SeqLength),
%add L-modified coverage for this example to the total:
    Cov1 is Cov+SeqLength,
    Start1 is Start+1,
    compute_Lmodcover_interval(Type, Clause, Start1-Finish,
                   Cov1, CovRes).
```

b) **parse_exple_for_length/3** - This predicate parses an example and returns the length of the sequence if it was parsed successfully. If it cannot be parsed, 0 is returned.

```
%used as:  parse_exple_for_length(Clause,Example,
                   SequenceLength)
 parse_exple_for_length((Head:-Body),Example,SeqLength) :-
  % the +(+()) are needed to ensure Head and Example
  % do not stay unified after the call
    +((+((Example=Head, call(Body))))),
    Example=middle(MidSeq,[]),
    length(MidSeq,SeqLength), !.
 parse_exple_for_length(_,_,0).
```

Table 5.7: Predicates calculating the modified coverage based on the length of training examples.

Chapter 6

L-Modification

This chapter builds on the concepts and literature that were reviewed in Chapters 2, 3 and 4. This chapter will finally introduce the specifics of our proposed L-modification. First, the benchmark clause evaluation function will be introduced, and then we will apply L-modification and partial L-modification to the benchmark. A series of experiments will be run using each of these clause evaluation functions.

Section 6.2 describes our first set of experiments, which investigates the general effect of L-modification and compares the results with the benchmark. Section 6.3 descries a set of experiments which further investigates the effect of the length of the examples in the training data as well as the influence of the ranges of the lengths of examples. Section 6.4 investigates the computational cost which is involved to learn from examples, and how much the lengths of examples can influence that cost. Section 6.5 investigates the relationship between the time needed to learn a good clause, compared to the length of the example it is learned from. Finally, Section 6.6 makes an analysis of the grammars which were induced in our experiments, Section 6.7 looks into the concept of partially L-modified clause evaluation functions and Section 6.8 investigates if any one term of our L-modified function might carry more weight than others.

All experiments in this chapter are reported using the following structure:

- Motivation
- Methodology
- Results

- Evaluation and Discussion

6.1 L-Modification of Clause Evaluation Functions

As described in Section 5.1 the aim of this work is to apply the principles described in Chapter 3 to Biological Grammar Learning using ILP, focusing on the main difference between this task and the more common learning tasks: the significant variation in lengths of the training examples. We decided to modify existing ILP clause evaluation functions (See Section 4.5) in such a way that they satisfy the MDL principle and consider the differences in the length of training examples. We will name this modification of the clause evaluation functions: *L-modification*.

Note here that a *clause evaluation function* (Section 4.5) and a *grammar performance measure* (Section 5.8) are two distinct things. While a clause evaluation function is used to grade each individual clause during learning, in the form of a node in the search tree, a grammar performance measure is only used to estimate the future performance of an entire grammar (a set of clauses) once learning is completed successfully.

Before devising L-modification, let us introduce the standard clause evaluation function used in this domain.

6.1.1 Standard positive-only evaluation function

As described in Section 5.4 we have positive and random examples, but no negative examples. Therefore, a positive-only learning approach (See Section 4.4) is applied in this work. The standard evaluation score used in ILP for positive-only learning was devised by Muggleton (Muggleton 1996):

$$Score = log\,(P) - log(\frac{(R+1)}{(Rsize+2)}) - \frac{L}{P} \tag{6.1}$$

Where P is the number of positives covered; R is the number of random examples covered; $Rsize$ is the total number of random examples and L is the number of literals in the hypothesis. This evaluation function, which we will refer to using the notation *Function 6.1* for the remainder of this thesis, will serves as our benchmark evaluation function. The aim of the following experiments is to compare the performance of L-Modified evaluation

functions against this benchmark.

6.1.2 L-modification of the standard positive-only evaluation function

We intend our clause evaluation function to consider the length of training examples, but Function 6.1 above does not. Therefore we decided to modify it accordingly. What we propose is to replace any quantity appearing in Function 6.1 that represents a numerical count of examples with a modified quantity which represents the length of examples.

More precisely, this means that instead of feeding P and R to the evaluation function, we replaced all their occurrences in Function 6.1 with *LModPos* and *LModRan* respectively. LModPos is the sum of the lengths of all covered positive examples and LModRan is the sum of the lengths of all covered random examples. These changes combine the idea of the existing pos-only evaluation function with the idea of considering the length of training examples. Henceforth we denote such a change of variables as *L-modification* (*LModPos* being L-Modified positive coverage and *LModRan* being L-Modified random coverage). In addition to P and R, the variable *Rsize* in Function 6.1 also refers to a number of examples: the total number of random examples in the training data. As the name of the variable suggests, this number intends to represent the size of the set of randoms. Therefore, within the context of this experiment, it makes sense that we also apply our L-modification to this variable. Therefore we replaced *Rsize* with *LmodRsize*, which is the sum of the lengths of all random examples in the training data. Applying these L-modifications to Function 6.1 gives us the following Function 6.2:

$$Score = log(LModPos) - log(\frac{LModRan + 1}{LmodRsize + 2}) - \frac{L}{LModPos} \tag{6.2}$$

6.1.3 Partially L-modified evaluation functions

Although our experiments will focus on Functions 6.1 and 6.2, we also ran our experiments on two other partially modified versions of Function 6.1. The motivation for this was to investigate whether partial L-modification, where we apply L-modification only to some of the variables, might be a useful concept.

The first partially L-modified Function 6.3 replaces P and R with *LModPos* and *LMod-*

6.1	Score =	$log(P)$	$-log(\frac{R+1}{Rsize+2})$	$-\frac{L}{P}$
6.2	Score =	$log(LModPos)$	$-log(\frac{LModRan+1}{LmodRsize+2})$	$-\frac{L}{LModPos}$
6.3	Score =	$log(LModPos)$	$-log(\frac{LModRan+1}{Rsize+2})$	$-\frac{L}{LModPos}$
6.4	Score =	$log(LModPos)$	$-log(\frac{LModRan+1}{Rsize+2})$	$-L$

Table 6.1: List of all Functions used in this work.

Ran respectively:

$$Score = log(LModPos) - log(\frac{LModRan + 1}{Rsize + 2}) - \frac{L}{LModPos} \tag{6.3}$$

The second partially L-modified Function 6.4 also takes into account that the value of LModPos is 10 to 60 times larger than P, therefore the last term of Function 6.3, $(\frac{L}{LModPos})$, possibly leads to the clause length L greatly losing influence on the score. Therefore we tried giving L more weight by not dividing by $LModPos$:

$$Score \ = \ log(LModPos) - log(\frac{LModRan + 1}{Rsize + 2}) - L \tag{6.4}$$

6.2 Experimenting with L-Modification

6.2.1 Motivation

The aim of the first experiment is to test the following **null hypothesis**:

> *Our proposed L-Modification of ILP clause evaluation functions, implemented into the generic ILP system Aleph, makes no difference to the predictive performance of learned definite clause grammars describing proteins of the NPP family.*

The rejection of this null hypothesis entails the acceptance of the alternative hypothesis:

> *Our proposed L-Modification of ILP clause evaluation functions, implemented into the generic ILP system Aleph*, makes a notable difference *to the predic-*

tive performance of learned definite clause grammars describing proteins of the NPP family.

6.2.2 Methodology

The way we go about putting this aforementioned hypothesis to the test is to run the following experiments: we run the ILP tool Aleph using the benchmark and the L-modified clause evaluation functions. As input for Aleph, we provide:

- the Background Knowledge given in Section 5.2,
- the pruning predicates, mode declarations and type declarations discussed in Section 5.3,
- the **NPP-middles** dataset described in Section 5.5, containing positive and random examples.

After learning, we test the resulting biological grammars following the 5-fold stratified cross validation testing strategy that was explained in Section 5.7. Using the results of these tests, we finally evaluate the performance of our biological grammars using precision, recall and F-measure as described in Section 5.8.

6.2.3 Results

Table 6.2 shows the results obtained from running Functions *6.1* and 6.2 (See Table 6.1) on the NPP-middle dataset. From the results of the 5-fold cross validation we get the means and standard deviations of *true positives* (TP), *false positives* (FP), *true randoms* (TR) and *false randoms* (FR).

6.2.4 Evaluation and Discussion

From the results of the 5-fold cross validation (summarized in Table 6.2), we can calculate accuracy, precision, recall and F-measure (See Section 5.8). Their means are given in Table 6.3.

As Function 6.1 serves as our benchmark we need to compare our L-modified clause evaluation function 6.2 against it.

Function 6.1

	TP	FP	TR	FR
Mean	10	130.4	451.2	5.2
St.deviation	2.12	72.44	72.28	2.39

Function 6.2

	TP	FP	TR	FR
Mean	7.8	5.8	575.8	7.4
St.deviation	2.39	1.92	1.79	2.61

Table 6.2: Results: TP: true positives, FP: false positives, TR: true randoms and FR: false randoms.

	Function 6.1	Function 6.2
Mean Accuracy	0.77	0.98
Mean Precision	0.08	0.57
Mean Recall	0.66	0.52
Mean F-measure	0.14	0.54

Table 6.3: Evaluation of the results, derived from the values given in Table 6.2.

Analysing Table 6.2, containing the test results, we can observe:

- Using our L-modified Function 6.2 has markedly decreased the FP rate down to 5.8, compared to our benchmark of 130.4.
- TPs have slightly decreased from 10 to 7.8.

Observations and conclusions drawn from Table 6.3, which contain the evaluation of the test results:

- The accuracy, seen in Table 6.3 was increased from 0.77 to 0.98. This is a consequence of the decrease of FPs. An accuracy of 0.98 seems very good, but as mentioned in Section 5.8, such a high accuracy could be misleading in this domain. Therefore we should rather look at the other values in Table 6.3.
- The precision has been considerably increased; from 0.08 to 0.57. The benchmark has a very low precision, as models learned with it accept 130 random examples on average, which is over 25% of the random examples provided in each fold. By comparison, models learned with the L-modified Function accept only 5.8 random examples, which is slightly over 1%.

- The recall decreased from 0.66 to 0.52. This is a consequence of the aforementioned decrease of the TPs.

- The weighted harmonic mean of precision and recall, the F-measure, which is our main means of evaluating the performance of the induced grammars has been increased, from 0.14 to 0.54. We consider this a significant improvement and therefore conclude that our L-modified clause evaluation function has outperformed our benchmark.

This last point clearly rejects our null hypothesis and thus we have no choice but to accept our alternative hypothesis:

> *Our proposed L-Modification of ILP clause evaluation functions, implemented into the generic ILP system Aleph*, makes a notable difference *to the predictive performance of learned definite clause grammars describing proteins of the NPP family.*

6.3 Grouping Training Examples by Length

6.3.1 Motivation

We were interested in investigating the influence of the length of the provided protein sequences on biological grammar learning. There are a few questions that we wanted to have answered:

- Considering that we allow the gap predicate (See Section 5.2.1) to cover any amount of amino acids, what is the impact of the different lengths of examples on the learning process?

- Is it easier to learn grammar rules that cover shorter examples?

- What is the impact of the *range* of the lengths of the examples given as training data?

- With respect to the range of the lengths of the examples, how do our L-modified Functions compare with the benchmark?

Generally, we set out to investigate the influence of the length of protein sequences, and their range, on learning biological grammars.

6.3.2 Methodology

We run Aleph, using both the benchmark Function 6.1 and all three L-modified clause evaluation Functions 6.2, 6.3 and 6.4 to learn biological grammars from each of the three sub-datasets. As input for Aleph, we provide:

- the Background Knowledge given in Section 5.2,
- the pruning predicates, mode declarations and type declarations discussed in Section 5.3,
- the three NPP sub-datasets: *NPP-middles-short*, *NPP-middles-medium* and *NPP-middles-large* (See Section 5.6 for a description how these datasets were compiled, see Table 5.3, page 54 for a summary).

After learning, we test the resulting biological grammars following the 5-fold stratified cross validation testing strategy that was explained in Section 5.7. Note that each of the three different sub-datasets has been individually prepared for 5-fold cross validation. Using the results of these tests, we finally evaluate the performance of our biological grammars using precision, recall and F-measure as described in Section 5.8.

6.3.3 Results

Table 6.4 shows the results obtained from testing the grammars learned using Functions *6.1* and 6.2 on the *NPP-middles-short*, *NPP-middles-medium* and *NPP-middles-long* datasets.

6.3.4 Evaluation and Discussion

From the results presented in Table 6.4 we can calculate the performance of the grammars through accuracy, precision, recall and F-measure (See Section 5.8). Their means are given in Table 6.5.

Analysing Table 6.5, containing the evaluation of the test results, we can observe:

Dataset	Function		TP	FP	TR	FR
NPP-middles-short	6.1	Mean	4	12.4	169.2	0.8
		St.deviation	1.00	9.13	8.93	1.10
	6.2	Mean	3	1	180.6	1.8
		St.deviation	0.71	1.00	1.34	1.10
NPP-middles-medium	6.1	Mean	2.6	15.2	157.6	2.6
		St.deviation	2.30	10.83	10.83	2.61
	6.2	Mean	2.8	1.8	171	2.4
		St.deviation	1.64	1.92	2.35	1.82
NPP-middles-long	6.1	Mean	1.2	34	193.2	4
		St.deviation	0.84	26.45	26.48	0.71
	6.2	Mean	0.4	3.8	223.4	4.8
		St.deviation	0.55	2.68	2.88	0.8

Table 6.4: Summary of the results from learning on each of the three datasets. TP: true positives, FP: false positives, TR: true randoms and FR: false randoms.

Dataset: NPP-middles-	Function	mean Accuracy	mean Precision	mean Recall	F-measure
a) short	6.1	0.93	0.24	0.83	0.38
	6.2	0.98	0.75	0.62	0.68
b) medium	6.1	0.90	0.15	0.50	0.23
	6.2	0.98	0.61	0.54	0.57
c) long	6.1	0.84	0.03	0.23	0.06
	6.2	0.96	0.10	0.08	0.09

Table 6.5: Evaluation of the results from learning on each of the three datasets.

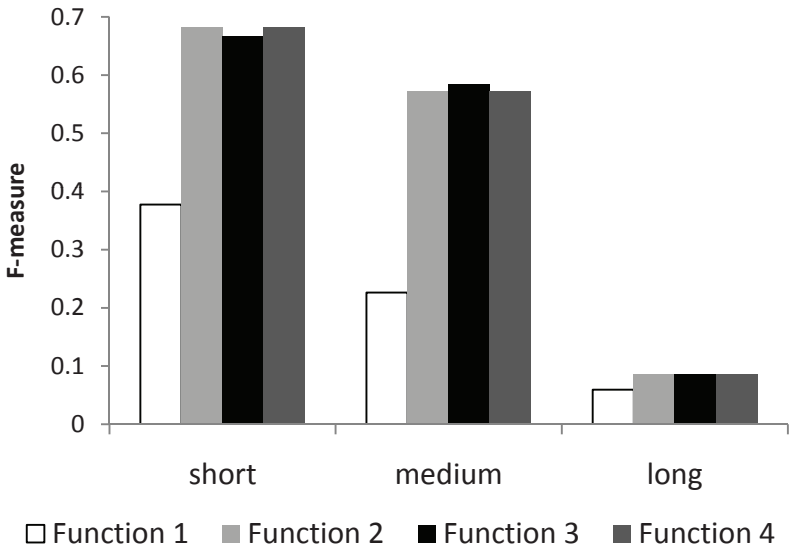

Figure 6.1: For all three parts of dataset NPP-middles-(short, medium and long) the performance for Functions 6.1, 6.2, 6.3 and 6.4 is plotted.

- L-modification has increased the accuracy, compared to the benchmark, on all three sub-datasets. Again, this can be credited to the decrease of FPs.
- The precision has also been increased compared to the benchmark on all three datasets.
- The recall has decreased compared to the benchmark on two of the three datasets. However on the NPP-middles-medium dataset, recall has only been increased by 0.04.
- Finally, the F-measure has been increased compared to the benchmark on all three datasets.

Figure 6.1 shows the F-measures plotted for all clause evaluation functions on all three datasets NPP-middles-short, NPP-middles-medium and NPP-middles-long. Looking at Figure 6.1, we can make the following observations:

- Our L-modified clause evaluation functions always achieve a better F-measure than the benchmark.
- The F-measure is highest when learning on NPP-middles-short, slightly lower when

learning on NPP-middles-medium, and a lot lower when learning on NPP-middles-long.

It is not surprising that the performance is lower on datasets that contain larger examples in their training data. Our intuition suggests that it should be easier to learn grammar rules that can successfully cover short sequences than it is to learn rules covering larger sequences.

Another reason for the decrease in performance when learning from the NPP-middles-long dataset could lie in the range of the lengths of the examples contained within the dataset. The shortest sequence in the NPP-middles-long dataset is 30 amino acids long, while the longest sequence contains 94. This gives us a range of 64, which is a lot higher than NPP-middles-medium, with a range of 16, and NPP-middles-short, with a range of 9. This indicates that a higher range of the lengths of the examples in a dataset could make it harder to learn a good grammar. Note that the range of the length of examples was not considered when compiling these datasets.

Furthermore, if we compare the results reported in this section, summarized in Table 6.5 on page 70 with the results of our first experiments, reported in Section 6.2 and summarized in Table 6.3 on page 67, an interesting observation can be made:

- Grammars learned from datasets NPP-middle-short and NPP-middle-medium have higher performance than those learned from the whole NPP-middles dataset.

This observation is interesting because it goes against our intuition. Generally, in Machine Learning, the more examples we have to learn on, the larger the dataset is, the better the resulting model will be. However, in this case, NPP-middle-short and NPP-middle-medium are subsets roughly one third of the size of the NPP-middles. Once again, the factor that stands out is the range of the length of examples in the training data. The range of the lengths of examples of the whole NPP-middles datasets is 79 while that of NPP-middles-medium is 16, and that NPP-middles-short is 9. Once more, we conclude that it is possible that the range of the lengths of examples has an impact on the performance of the learned grammars: a larger range might make it harder to learn a good model, with or without L-modification.

Note here that the resulting grammars learned in this section are effectively incomplete, as they were learned only from subsets of the complete dataset. We are not suggesting to learn from smaller datasets in order to seek a lower range of length of examples.

6.4 Computational Cost

6.4.1 Motivation

While conducting the experiments discussed earlier in this chapter (Section 6.2 and 6.3) we observed that it takes Aleph longer to learn models using our L-modified evaluation functions than it took using the benchmark evaluation function. We decided to investigate the cause of the increase in time.

The first factor which causes this increase in time is that our L-Modified Functions require more computation. Aleph keeps track of the number of positive and negative examples covered by the clause to be evaluated and passes these values on to the predicate that is to evaluate the given clause. The benchmark function only uses these values to compute a score. However our L-modified Function requires L-modified coverage to calculate the score and does not use the values passed on by Aleph. Instead it calls our special predicates (See Section 5.9) to calculate its own L-modified coverage.

Each time Aleph uses the user defined Function to evaluate a clause, it will make two calls to the predicate **compute_Lmodcover/3**, (See Table 5.7 page 61); once for positive and once for random examples. Calling **compute_Lmodcover/3** involves, among other things, the parsing of several examples through the predicate **parse_exple_for_length/3** to compute the L-modified coverage.

This extra computational work, which is necessary for the computation of L-modified Functions, accounts for some of the increased time needed when comparing with the benchmark. However, we also decided to investigate the possibility that more nodes are constructed using our L-modified evaluation function. Given that our Function results in grammars with better performance, we expect it to be more selective when it comes to accepting clauses, thus more clauses might be constructed before Aleph is satisfied with the learning process.

6.4.2 Methodology

Just as in Section 6.2 we run Aleph using the benchmark Function 6.1 and all three L-modified clause evaluation functions 6.2, 6.3 and 6.4. However, this time we specify several different values for maximum number of nodes in the input settings file. Aleph uses an input parameter, called `setNodes`, to limit how many nodes can be constructed during each search. The standard value for `setNodes`, which we used in all our other experiments is 100000. For this experiment however, we decrease that value several times and observe the performance of the resulting model. The different values for `setNodes` are: 100000, 50000, 10000, 5000, 1000.

As input for Aleph, we provide:

- the Background Knowledge given in Section 5.2,
- the pruning predicates, mode declarations and type declarations discussed in Section 5.3,
- the **NPP-middles** dataset described in Section 5.5, containing positive and random examples.

Whenever Aleph has finished learning and comes up with a theory, it returns the total amount of nodes constructed during the search. This number indicates the total number of nodes that were constructed in order to learn the resulting grammar, meaning: the sum of the number of nodes that were constructed in order to learn each individual clause of that grammar. We record this number for each experiment.

After learning, we test the resulting biological grammars following the 5-fold stratified cross validation testing strategy that was explained in Section 5.7. Using the results of these tests, we finally evaluate the performance of our biological grammars using precision, recall and F-measure as described in Section 5.8.

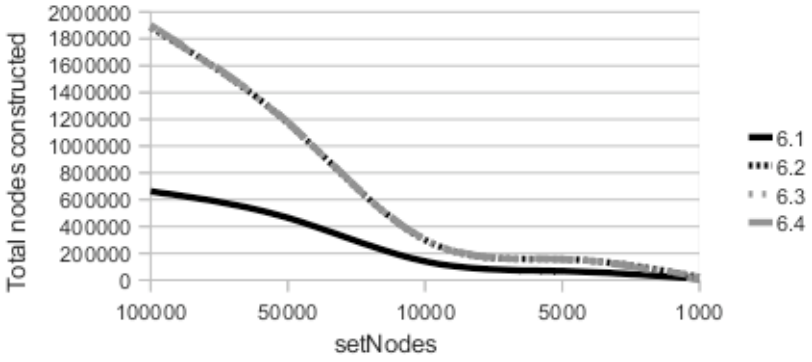

Figure 6.2: The *setNodes* parameter plotted against the total nodes constructed during learning of a model for all four clause evaluation functions.

6.4.3 Results

Total nodes constructed

Table 6.6 contains the total nodes constructed for each setNodes and each evaluation function. Figure 6.2 shows the data in Table 6.6 plotted on a graph which shows the total number of nodes constructed against the setNodes parameter for each evaluation function.

	Functions			
setNodes	6.1	6.2	6.3	6.4
100000	662299	1885723	1885723	1903029
50000	464489	1172399	1172399	1179706
10000	139747	299747	299747	305747
5000	67000	156000	156000	156000
1000	10000	24000	24000	2000

Table 6.6: Mean total number of nodes constructed to learn a grammar.

Performance

Table 6.7 shows the means and standard derivations of True Positives, False Positives, True Randoms and False Randoms that were covered by the learned theories on the test sets.

6.4.4 Evaluation and Discussion

Table 6.8 gives the performance of the grammars learned in each experiment. The values in the table are derived from the results given in Table 6.7. Figure 6.3 shows the performance using each of the clause evaluation functions against the decreasing setNodes.

SetNodes: 100000								
Function	TP	sd.TP	FP	sd.FP	TN	sd.TN	FN	sd.FN
6.1	10	2.12	130.4	72.44	451.2	72.28	5.2	2.39
6.2	7.8	2.39	5.8	1.93	575.8	1.79	7.4	2.61
6.3	7.8	2.39	5.8	1.93	575.8	1.79	7.4	2.61
6.4	7.8	2.39	5.8	2.39	575.8	2.289	7.4	2.61
SetNodes: 50000								
6.1	10	2.12	129	73.72	452.6	73.54	5.2	2.39
6.2	7.8	2.39	6.4	2.7	575.2	2.59	7.4	2.61
6.3	7.8	2.39	6.4	2.7	575.2	2.59	7.4	2.61
6.4	7.8	2.39	6.4	3.21	575.2	3.11	7.4	2.61
SetNodes: 10000								
6.1	10.2	1.79	214.4	75.14	367.2	75.19	5	2.12
6.2	9.2	2.59	48.6	14.15	533	13.77	6	2.74
6.3	9.2	2.59	48.6	14.15	533	13.77	6	2.74
6.4	9.2	2.59	47.6	16.79	534	16.42	6	2.74
SetNodes: 5000								
6.1	10	1.87	229.6	62.92	352	62.96	5.2	2.17
6.2	9.2	1.92	79.8	14.62	501.8	14.08	6	2.12
6.3	9.2	1.92	79.8	14.62	501.8	14.08	6	2.12
6.4	9.4	2.3	87	16.94	494.6	16.46	5.8	2.49
SetNodes: 1000								
6.1	13.6	1.52	358.8	69.37	222.8	69.13	1.6	1.52
6.2	12.2	2.28	344	82.46	237.6	82.54	3	2.45
6.3	12.2	2.28	344	82.46	237.6	82.54	3	2.45
6.4	15.2	0.45	581.6	0.55	0	0	0	0

Table 6.7: TP: Mean True Positives, FP: Mean False Positives, TR: Mean True Randoms and FR: Mean False Randoms for each setNodes parameter and each clause evaluation function.

Figure 6.3 shows the graph plotting the performance of the evaluation functions against the total number of nodes constructed. We can see that when more nodes are constructed, then grammars with better performance are learned.

We have observed in all our experiments that, when using the L-modified Functions 6.2, 6.3 and 6.4, the induction process takes longer than when using the benchmark Function 6.1. From the data given in Table 6.6 we can see that a learning process using the default setNodes value and the benchmark Function needs to construct around 660,000 nodes in total to learn a theory, while around 1,900,000 were constructed using the L-modified Functions. A similar trend is observed if the setNodes parameter is reduced. In Figure 6.2 we can see that the experiment using the benchmark Function 6.1 constructs far less

SetNodes	Function	Accuracy	Precision	Recall	F-measure
100000	6.1	0.77	0.07	0.66	0.13
	6.2	0.98	0.57	0.51	0.54
	6.3	0.98	0.57	0.51	0.54
	6.4	0.98	0.57	0.51	0.54
50000	6.1	0.78	0.07	0.66	0.13
	6.2	0.98	0.55	0.51	0.53
	6.3	0.98	0.55	0.51	0.53
	6.4	0.98	0.55	0.51	0.53
10000	6.1	0.63	0.05	0.67	0.09
	6.2	0.91	0.16	0.61	0.25
	6.3	0.91	0.16	0.61	0.25
	6.4	0.91	0.16	0.61	0.26
5000	6.1	0.61	0.04	0.66	0.08
	6.2	0.86	0.1	0.61	0.18
	6.3	0.86	0.1	0.61	0.18
	6.4	0.84	0.1	0.62	0.17
1000	6.1	0.4	0.04	0.89	0.07
	6.2	0.42	0.03	0.80	0.07
	6.3	0.42	0.03	0.8	0.07
	6.4	0.03	0.03	1	0.05

Table 6.8: Performance of each evaluation function with relation to the setNodes parameter.

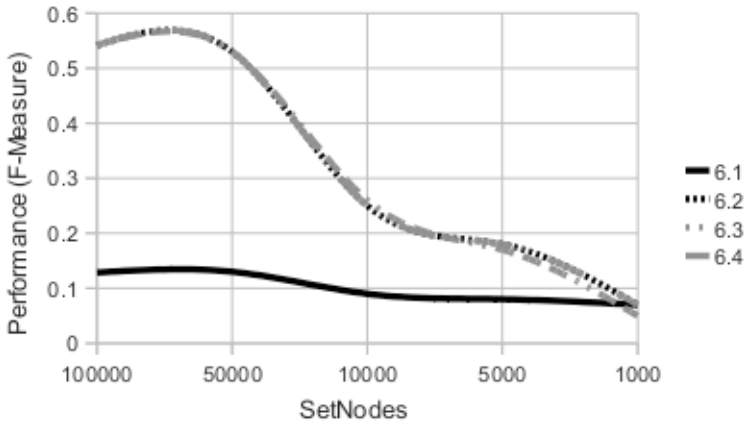

Figure 6.3: For each evaluation function the performance is plotted against the setNodes parameter.

nodes that the experiment using the L-modified Functions. It is not surprising that with the decrease of the setNodes parameter comes a decrease of the nodes constructed. We can observe a similar behaviour for the performance of the learned theories in Figure 6.3.

Figure 6.4 shows the performance plotted against the total number of nodes constructed

for Functions 6.1 and 6.2. The R-squared values derived from the linear regression lines (which are not shown in Figure 6.4) are given for each Function: R-squared = 0.862 for Function 6.1 and R-squared = 0.925 for Function 6.2. Both these values indicate that there is a correlation between performance and total nodes constructed, which is consistent with our hypothesis.

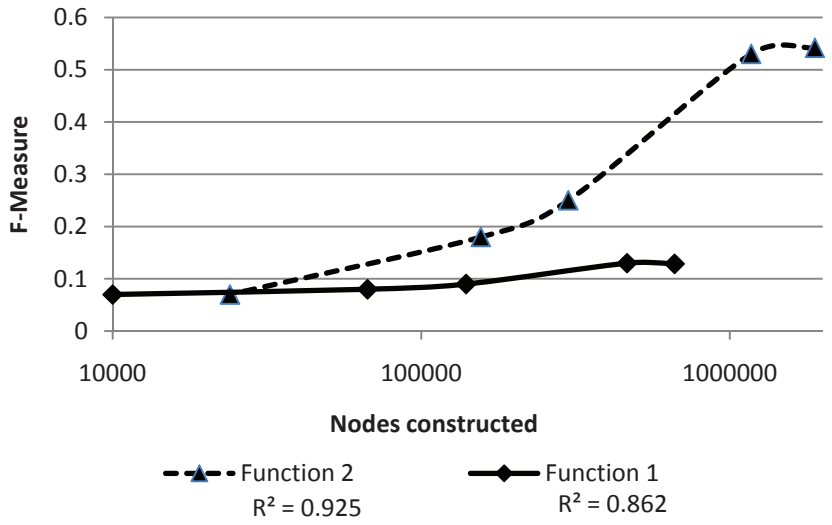

Figure 6.4: Performance plotted against the total number of nodes constructed during the search. The R-squared values are given for each Function. X-axis is in logarithmic scale.

Finally, inspecting Table 6.6 and subsequently Figure 6.2, we can observe that the total nodes constructed using the benchmark and L-modified Functions seem to be correlated. To confirm this we plotted the total nodes constructed for both Functions 6.1 and 6.2 against each other for each setNodes parameter. Figure 6.5 shows this graph. The linear regression line gives us an R-squared value of 0.993, therefore we can conclude that the total nodes constructed for both Functions are correlated. This indicates that the setNodes parameter has equal weight on both Functions.

6.5 Search Time vs. Length of Examples

6.5.1 Motivation

While evaluating the results of the previous experiments in this chapter, we postulated whether there was a relationship between the time needed to learn from an example and the length of that example. Our intuition tells us that more time should be needed to

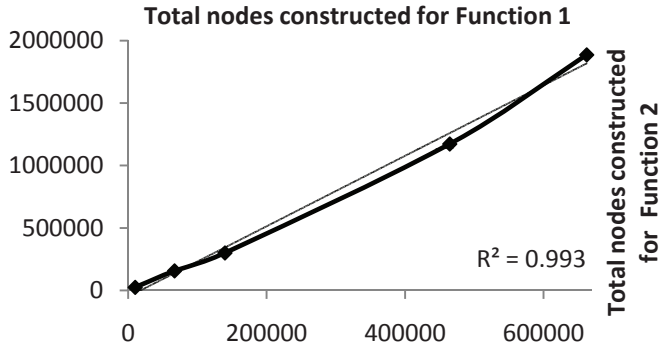

Figure 6.5: Total nodes constructed for both Functions against each other for each setNodes parameter.

learn from larger examples than from smaller ones.

6.5.2 Methodology

In order to confirm the hypothesis introduced in the previous subsection, that more time is needed to learn from larger examples, we analysed the output which was produced by Aleph during our experiments on term domination (described later in this chapter in Section 6.8). During learning, Aleph aims to find the best clause that is able to cover the current example and adds that clause to the grammar. Whenever it has determined which clause encountered is the best, it displays the clause itself and some information about it, including the time needed to learn that clause and the example that the clause was learned from. From that output, we can therefore extract the length of the example and the time needed to learn for each clause.

6.5.3 Results

We collected the times and example lengths for each of the clauses that were added to the grammar learned in the experiment in Section 6.8 and plotted them against the size of the respective examples they were learned from. Figure 6.6 shows this plot for the L-modified evaluation function, learning on the whole NPP-middle dataset, without cross validation. Note that this graph only includes those examples that the search decided to learn on. Whenever the search finds an acceptable rule and adds it to the grammar, it removes all the examples that are covered by this rule from the pool of examples to be learned on

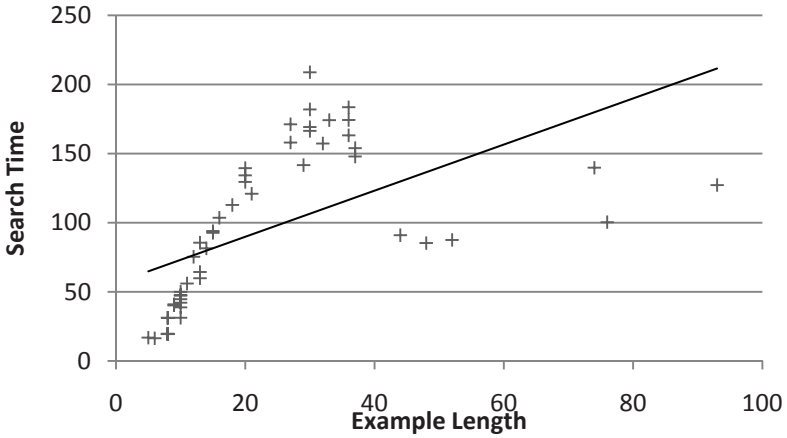

Figure 6.6: This graph plots the length of the examples in the training data against the time needed (in seconds) to search a rule that covers each example. It also contains a linear regression line found using the method of least squares.

(Step 5. *Remove Redundant* in the algorithm in Table 5.6 page 58). Therefore, we have no data on the search-time for the removed examples.

6.5.4 Evaluation and Discussion

We can see from Figure 6.6 that the very small examples need very little time to be learned from. As the size of the examples increases, the time needed to learn from them increases as well. This trend holds up until the examples reach lengths of over 40 amino acids and from there onwards the search time is considerably lower. This observation was unexpected.

Figure 6.6 also contains the linear regression line derived from all the data points plotted, using the least squares method. The equation for that line is:

$$y = 1.667x + 56.50 \tag{6.5}$$

with the Pearson product-moment correlation coefficient:

$$r \;=\; 0.559 \quad and\; R-squared \;=\; r^2 \;=\; 0.312 \tag{6.6}$$

Such a low value for R-squared would suggest that example length and search time are not correlated. However taking into account the unusual behaviour of the last 6 points, representing the examples with over 40 amino acids, we decided to draw the same graph

again, ignoring those 6 data points. This graph is presented in Figure 6.7. This graph also contains the linear regression line derived from all the data points plotted. The equation for that line is:

$$y = 5.433x + 4.093 \tag{6.7}$$

with the Pearson product-moment correlation coefficient:

$$r \;=\; 0.943 \quad and\; R-squared \;=\; r^2 \;=\; 0.889 \tag{6.8}$$

This value for R-squared (0.889) does indeed suggest a correlation between example length and search time.

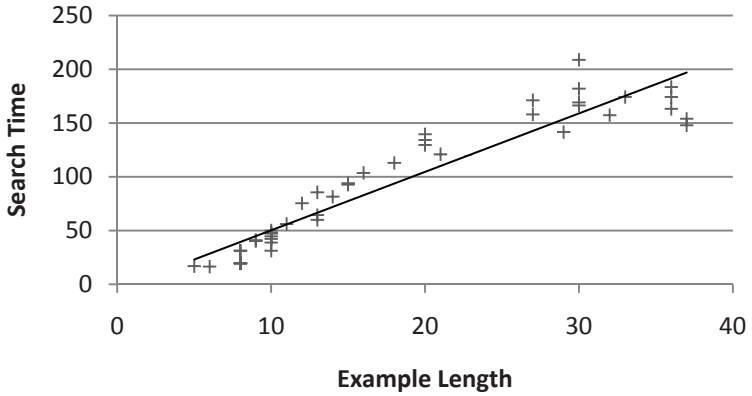

Figure 6.7: This graph plots the length of the examples in the training data against the time needed (in seconds) to search a rule that covers each example. It also contains a linear regression line found using the method of least squares. This graph omits all examples with more than 40 amino acids.

To find out if this behaviour is unique to the experiments reported in this section, we looked at the data we received from some of our previous experiments in this chapter:

- using Function 6.2, learning on the whole NPP-middle dataset, applying 5 fold cross validation,
- using Function 6.1, learning on the whole NPP-middle dataset, applying 5 fold cross validation,
- using Function 6.1, learning on the whole NPP-middle dataset, applying no cross validation.

We then followed the same procedure: we plotted the length of examples against the time needed to learn them. We observed the same behaviour: learning very small examples needs very little time, as the length of examples increases, so does the time, and once the length of examples rises above 40 amino acids, the time decreases. Omitting the examples with less than 40 amino acids also gives us R-square values above 0.8 for each experiment. Note: as all these graphs looked more or less identical to Figure 6.6 and 6.7, we therefore decided to not present them here.

As the same behaviour was observed in all the above experiments, with or without 5 fold cross validation, independent of the Function used, we can conclude that neither of those factors is responsible for the decrease of learning-time for the larger examples. In fact, the only distinctive factor seems to be the length of the examples, as learning time only decreases if examples contain more than 40 amino acids. We are confident that this does not represent a shortcoming of our L-Modification.

Plotting a linear regression line based on the length of the examples and the time it took to learn clauses that cover them and calculating an R-squared value of 0.889 based on it gives a clear indication that both values are correlated. We conclude that our initial hypothesis is confirmed: more time is indeed needed to learn from larger examples.

The behaviour exhibited by examples consisting of more than 40 amino acids potentially discredits the use of Aleph when learning from examples exceeding such length. As part of the future work, the effect of learning from examples that contain more than 40 characters could be investigated (See Section 8.3).

6.6 Analysis of induced grammars

We have already noted that using our L-modified clause evaluation function increases the accuracy and F-measure of the learned biological grammars. However, it was also observed that applying L-modification to the evaluation function increases the number of rules in the learned grammars. See Table 6.9 for an analysis of the grammars which were learned during our experiments described in Section 6.2. Columns 2 and 3 show that grammars learned using L-modified Functions contain a lot more rules than those which

were learned using the benchmark. Furthermore, column 4 shows that grammars learned using L-modified Functions contain a lot more rules that cover only one single example compared to those which were learned using the benchmark.

Function	mean # of rules per fold	# of rules in whole dataset	# of rules covering 1 example
6.1	15	14	3
6.2	30.6	35	22

Table 6.9: Details on the number of rules learned for Functions 6.1 and 6.2.

Therefore the grammars which are better at describing the positive examples (cover a lot less random examples), are more complex as well.

Taking into account that the number of rules in the induced grammars is different, we decided to readdress the results we presented in Section 6.4. Instead of looking at the total number of nodes that were constructed to learn the grammar, we focused on the average number of nodes that were constructed to learn one rule. In essence we took the total number of nodes constructed to learn each grammar and divided them by the number of rules present in that grammar. This gives us the mean number of nodes constructed per rule. We then plotted this number against the performance of the respective grammar for each setNodes parameter. The values themselves can be seen in Table 6.10 and the plotted graph can be seen in Figure 6.8. In this figure, each datapoint represents the observations for one setNodes parameter.

Looking at Figure 6.8 we can see that the performance of the rules learned using Function 6.2 is always higher than the performance of the rules learned using Function 6.1, for a comparable amount of mean nodes constructed per rule. The exception would

	Function 6.1		Function 6.2	
SetNodes	Mean number of nodes constructed	F-Measure	Mean number of nodes constructed	F-Measure
100000	47307	0.13	53878	0.54
50000	33178	0.13	33497	0.53
10000	9982	0.09	8564	0.25
5000	4786	0.08	4457	0.18
1000	714	0.07	686	0.07

Table 6.10: Mean number of nodes constructed per rule and the respective performance of the complete grammar.

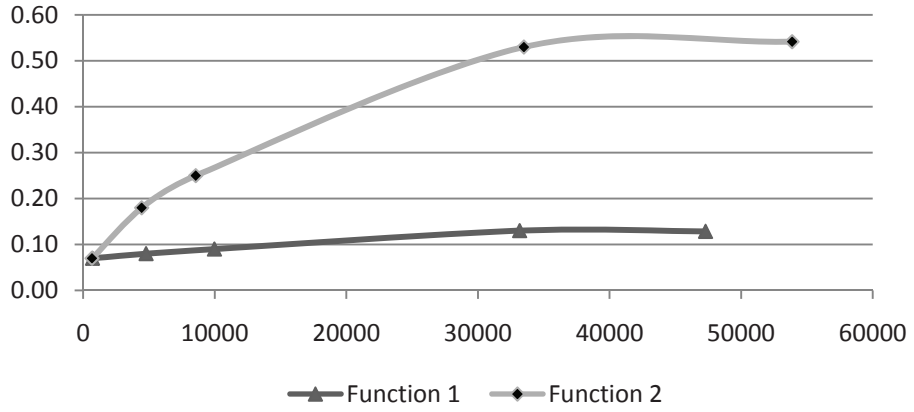

Figure 6.8: Mean number of nodes constructed per rule, plotted against the performance of the complete grammar, for each setNodes parameter.

the lowest setting of setNodes parameter, at which point the performance is the same.

6.7 Partially L-modified Functions

All experiments in this chapter were run using the benchmark Function 6.1, our L-modified Function 6.2 and two partially L-modified Functions 6.3 and 6.4. See Table 6.1 page 65 for a summary. However, throughout the chapter, we often only reported results using Functions 6.1 and 6.2. The reason for this is that in all our experiments, there was only little difference between the resulting grammars and their performances when comparing L-modified Function 6.2 and partially L-modified Functions 6.3 and 6.4. In fact, for some of the experiments we ran, the outcomes were identical. Therefore there is no notable difference between partial and full L-modification.

If we learn on the entire NPP-middle dataset, without any cross validation, then the behaviour of Aleph is identical for all 3 L-modified Functions; the same numbers of nodes are constructed and the same grammar is learned. The only thing that differs between the three Functions is the time is needed to learn the grammar, which can be explained by different computation times required to use each of the Functions.

In conclusion, there is a significant difference in performance when introducing the length of examples into the clause evaluation function, but there is no further advantage to be gained by considering the differences between L-modified Function 6.2 and partially L-modified Functions 6.3 and 6.4. Partial L-modification does not seem to be a useful

concept.

A potential explanation for this behaviour could be that the first term of these L-modified equations, $log(LModPos)$, is dominating the Function, which would hardly be surprising considering the large values of $LModPos$ we use in our experiments. Therefore, it might be beneficial to investigate the impact of each of the terms on the final score.

6.8 Term domination in L-Modified Clause Evaluation Functions

6.8.1 Motivation

The findings presented in Section 6.7 were unexpected, therefore we decided to investigate if all terms in the equation for our L-Modified Function 6.2 have equal weight on the outcome. If the terms should not have equal weight then we wanted to know which is the dominant term and why.

6.8.2 Methodology

We ran Aleph, providing the following input:

- the Background Knowledge given in Section 5.2,
- the pruning predicates, mode declarations and type declarations discussed in Section 5.3,
- the **NPP-middles** dataset described in Section 5.5.

We did not run any tests, so a cross-validation was not performed.

We changed the user-defined predicates which Aleph uses to calculate our L-modified coverage (See table 5.7) in a way that all the variables needed to calculate our L-modified score are not only used, but also saved in a separate output as well. Each time the evaluation function was successfully called, the following variables were recorded:

- *P, R* (# of positive and random examples covered),
- *L* (# terms in the clause),
- *LModPos, LmodRan* (L-modified Positive and Random coverage),

- *Score* (the value produced by the equation).

This way, we have all the necessary information to calculate the individual terms of the Function used. In this experiment, we used Function 6.2:

$$Score = log(LModPos) - log(\frac{LmodRan + 1}{78496 + 2}) - \frac{L}{LModPos}$$

and thus we kept track of the values for the following terms:

$$term1 = log(LModPos)$$

$$term2 = log(\frac{LmodRan + 1}{78496 + 2})$$

$$term3 = \frac{L}{LModPos}$$

Where

$$Score = term1 - term2 - term3$$

During learning, whenever Aleph finds a *good clause* it will add its details onto the output produced. What constitutes a *good clause* is dictated by various Aleph parameters (Srinivasan 1993). We therefore have access to all the *good clauses* Aleph comes across during induction, not only those that were actually added to the grammar. This information, together with the additional outputs that give us the contents of the needed variables, now enables us to analyse the three terms of all good clauses.

6.8.3 Results and Discussion

During this experiment, the function was run a total of 970,380 times. Out of those, 63,643 times, a good clause was evaluated. Table 6.11 gives the summary of the results. The details of all three terms are listed. For easier comparison between them we listed term2, which is always negative, by its absolute value.

The last row of Table 6.11 contains the absolute value of the mean of $|\frac{Score}{term}|$ for each term. This ratio between the final score and term aims to demonstrate the impact that each individual term can have on the final score. If all three terms had a comparable impact on the final score, we would expect the values of these ratios to be close to each other.

	term1	$\|term2\|$	term3
Mean	5.13	6.12	0.07
Standard deviation	1.32	1.3	0.12
Median	5.4	5.86	0.02
Minimum	1.79	3.77	0
Maximum	7.5	9.32	0.83
$\|\frac{Score}{term}\|$	2.37	1.91	768.23

Table 6.11: Summary of the results.

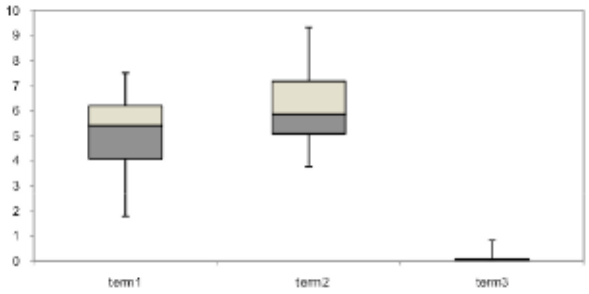

Figure 6.9: Based on the values from the 63643 cases we plotted a box-and-whisker plot, containing all values of the three terms

We can however observe that $|\frac{Score}{term3}|$ is extremely large (768.23), compared to the other two (2.37 & 1.91). This indicates that term3 has only very little impact on the final score. Consider that term3 fits into Score 768 times. The same observation can be made based on Figure 6.9 which shows a box-and-whisker plot containing all values of the three terms. We can clearly see that term1 and term2 have a similar weight on the outcome, however term3 is barely even visible on the graph, thus, it has a very small, if not negligible effect on the end result.

We conclude that term1 and term2 do influence the score in similar ways, while term3 seems to be negligible. The consequence of term3 ($term3 = \frac{L}{LModPos}$) having no impact on the outcome of the score is that L, the length of the clause has no impact on the score.

Normally this would reduce the usability of the Minimum Description Principle, which is concerned with the length of a clause. However, if we compare the large length of the examples (and their variations) with the rather short length of the clauses (and their variations), it becomes apparent that L would not have that much impact anyway. This is further demonstrated by the fact that Function 6.4, which only uses L as its term3, behaves

similarly to Function 6.2. From this we conclude that in this domain, when learning NPP grammars, term3 is not significant. If L-modification, and in particular this L-modified evaluation function, is applied to other domains in the future, then the usability of term3, within these domains, should be further investigated.

6.9 Conclusions

We have shown that when learning on the NPP-middle dataset, the L-modifications we propose do improve the performance of the induced grammars, both in terms of accuracy and F-measure.

The precision is always higher using L-modified functions than using the benchmark. The recall however was decreased in some experiments. This is a consequence of a strong decrease of false positives. A potential consequence of this decrease could be that fewer undiscovered true positives (which are hidden in the random examples) are now covered by our grammars. However this could also mean that the learning is now more selective in which randoms it accepts and thus, is more likely to accept only the hidden undiscovered positives.

Splitting the NPP-middle dataset in 3 disjoint subsets and learning on those, we have shown that our L-modification improves performance of induced grammars for short, medium and long examples. Within this context, we have also shown that it is generally harder to learn rules covering longer examples than shorter ones. By observing the outcomes of learning on NPP-middle dataset and comparing them with those that we can observe when learning on each of its subsets (NPP-middle-short, -medium and -large) we conclude that it is harder for the ILP tool used in this work (Aleph) to learn from a set of examples whose lengths have a larger range.

Setting different limits for the number of nodes allowed to be constructed during learning, we have shown that for larger search spaces, using the L-modified clause evaluation functions results in grammars with a higher F-measure. We have also noticed that more nodes are constructed during the learning process, and thus, more time is needed.

Plotting the length of examples in the training data against the time it took to learn

clauses that cover them, we discovered that there is a correlation between the two factors, as long as the examples contain less than 40 characters. For examples containing more than 40 characters, this observation does not hold up. This gives cause to potential future work investigating why the behaviour changes for examples reaching beyond this length.

Our analysis of the grammars which were induced in our experiments shows that through L-modification, learned grammars have a better performance on unseen data, but they are a little more complex compared to grammars learned through the benchmark. We also showed here that for a comparable number of nodes constructed per rule, the performance of the grammars learned using the L-modified Function 6.2 is always higher.

When comparing L-modified Function 6.2 and partially L-modified Functions 6.3 and 6.4, we found that their behaviour is actually very similar. This led to the conclusion that at least one of the three terms in the clause evaluation function has a much lower impact than the other two.

When analysing the impact of the three different terms, which Function 6.2 consists of, onto the final score, we discovered that as a consequence of L-modification, one term lost its weight on the final score. This term represents the length of the clause that is evaluated. However considering the huge differences between the range of the lengths of clauses and that of examples, it makes little difference in our experiments, but if L-modification was to be used in other domains, caution is advised.

Considering all the above, we can finally conclude that the L-modification proposed in this chapter does indeed improve the performance Function 6.1.

Chapter 7

L-Modification and Noise

7.1 Motivation

We showed in Chapter 6 that the L-modification of ILP clause evaluation functions does improve the performance of an induced biological grammar. However biological data sets are far from perfect and it might be of interest to investigate the robustness of clause evaluation functions in the presence of noise. Therefore we decided to introduce several levels of noise into our dataset and investigate how the performance of induced biological grammars degrades as the level of noise introduced into the training data increases.

Within this context, we also compared the behaviour of L-modified Functions with that of the benchmark Function, just as we did in the previous experiments in Chapter 6.

7.2 Noise

There are two types of possible noise that were considered:

1. Transcription Error (TE) - an error in the transcription from DNA into RNA. Such an error subsequently results in errors in the amino acid sequences which are translated from the RNA sequences. The consequence of a transcription error in a sequence is that there is a chance of one or more amino acid characters in the protein sequence being wrong.

2. Classification Error (CE) - an error in the class label of a sequence. This error can occur when sequences are wrongly labelled or classified. The consequence of a

classification error is that a protein sequence is labelled to be member of a given family when in fact, it is not.

For the experiments reported in this chapter, we only consider TEs. As we are applying a positive only approach (See Section 4.4) CEs in the random examples have already been accounted for anyway.

In order to introduce noise simulating TEs, a given percentage of the amino acid characters in the protein sequences which make up the positive training set need to be changed. There are several options of how such a character could be changed:

1. a character is replaced by another, different character,
2. a character is deleted,
3. a character is added.

For simplicity we decided to only use the first of the options above: replace the character that is to be changed by a different character taken from the available alphabet of amino acids.

So how do we determine which characters in a sequence are to be changed and by which other characters are they to be replaced with? From a biological point of view it is unlikely that every amino acid in a given sequence has the same chance of being erroneously transcribed. Furthermore, for each amino acid that has been erroneously transcribed, there are probably some amino acids that are more likely than others to take the place of the correct amino acid. Unfortunately, details on the actual probabilities of transcription errors are hard to come by, so we decided to run with the assumption that there is a uniform distribution on both accounts. This means that we choose entirely at random which amino acid character in all the given sequences is to be changed and we also choose entirely at random what character we replace it with, as long as it is not the same character. For the task at hand, it does not make a lot of difference: the aim of this experiment is not to evaluate the transcription of proteins, but to create an artificial dataset which we can use to investigate the behaviour related to our L-modification of the clause evaluation functions.

7.2.1 Introducing Noise

As in most previous experiments, we used the **NPP-middles** dataset described in Section 5.5, containing positive and random examples. Noise was introduced only to the positive examples, according to the algorithm in Table 7.1. This algorithm chooses a number of random characters from the whole set of positive examples and changes them. We decided on this strategy, as opposed to choosing a number of random characters in a single sequence and repeating the same process for each sequence. There are several advantages in applying this strategy:

- It is ensured that noise is rarely introduced into every single sequence in the set.
- Larger sequences have a greater chance of having characters chosen to be changed. This is mirroring the real life situation; it is more likely that one or more TEs occur inside a larger sequence (say 60 characters) than inside a shorter sequence (say 10 characters).
- As our algorithm is calculating the number of characters only once for the whole dataset, as opposed to once for each sequence, rounding errors are minimized. As we are unable to change a fraction of a character, the number of characters to change needs to be rounded down. If this rounding were to happen for each of the 75 sequences, it could entail a lot of unintroduced noise. Using our algorithm, such rounding will occur only once for the whole dataset.

The algorithm also makes sure each character to be replaced is indeed replaced by a *different* character.

Noise is only introduced into the positive examples of the training data. Introducing noise into random examples does not make much sense as it is very unlikely to have an impact on the outcome and it also would not give us any useful information. Introducing noise into the test set also makes no sense, as this would decrease the usefulness of the evaluation procedure. Furthermore, every time a learning process is started, noise is introduced from scratch. This means that for each fold, and for each run, noise is introduced into the clean dataset just prior to learning.

1.		for all positive sequences in Dataset D
	1.1	extract all occurrences of amino acid characters and collect them in one String CD
2.		count TC, the total number of characters in CD
3.		determine X, the number of characters to be changed, based on the given noise level
4.		find X individual random characters in CD
5.		forall X characters found:
	5.1.	find a random character C in CD
	5.2.	generate a random amino acid character CR
	5.3.	if CR != C then: replace C in CD with CR
	5.4.	else: return to 5.2.

Table 7.1: The algorithm that was used to introduce noise into the dataset.

7.3 Methodology

We run the ILP tool Aleph using the benchmark Function 6.1 and the L-modified Function 6.2 (See Table 6.1 page 65), thus applying the positive-only learning approach we used in the experiments described in Chapter 6. As input for Aleph, we provide:

- the Background Knowledge given in Section 5.2,
- the pruning predicates, mode declarations and type declarations which were discussed in Section 5.3,
- the **NPP-middles** dataset described in Section 5.5, containing positive and random examples, after noise was introduced (as described in Section 7.2.1).

After learning, we test the resulting biological grammars following the 5-fold stratified cross validation testing strategy that was explained in Section 5.7. Using the results of these tests, we finally evaluate the performance of our biological grammars using precision, recall and F-measure as described in Section 5.8.

7.3.1 Levels of Noise

To see how the performance behaves in relation to the noise introduced, we ran a number of experiments, each time gradually increasing the level of noise introduced. Our first experiment introduces 8% of noise, while each subsequent experiment increases the noise by 2% until a level of 30% is reached. From there onwards the level of noise is increased by

5% in each subsequent experiment until 90% noise is reached. This gives us the following noise levels (in %):

$$8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30, 35, 40, 45, 50, 55, 60, 65, 70, 75, 80, 85, 90$$

7.3.2 Statistical Significance

To achieve statistically significant results we ran each set of experiments twenty times under the exact same conditions, using the exact same parameters. For each of those twenty experiments the level of noise was the same, however each time we started with the clean dataset and reintroduced the noise. This allows us to reduce the impact of outliers such as special cases where the introduction of noise either has a very little or an extremely large effect on the outcome.

7.4 Results

A total of 4800 experiments were run. Using all the results available we monitored:

- $noise$,
- $fold$,
- $ClauseEvaluationFunction$,
- $run(1-20)$,
- $TruePositives$ (TP),
- $FalsePositives$ (FP),
- $TrueRandoms$ (TR),
- $FalseRandoms$ (FR).

We used TP, FP, TN and FN to calculate $Precision$, $Recall$ and ultimately the $F-measure$. Then we calculated the mean F-measure among all the 20 runs. From there we calculated the mean F-measure among the 5 folds. The resulting mean F-measures for

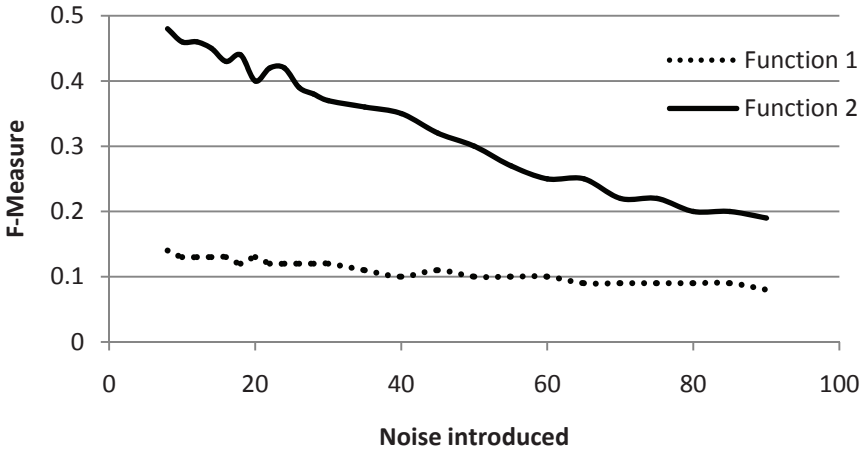

Figure 7.1: The mean performance across all 20 runs plotted against the level of noise introduced.

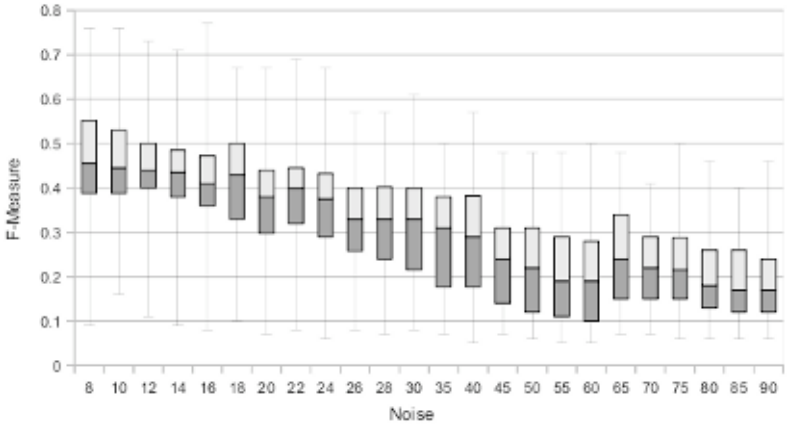

Figure 7.2: Box-Whisker plot of the performances achieved using Function 6.2 plotted against the level of noise introduced.

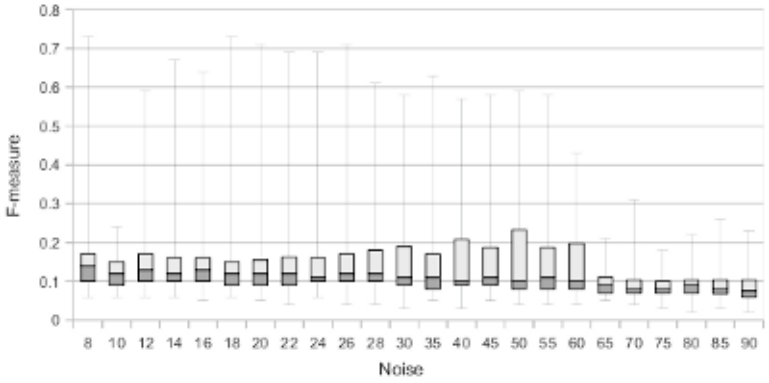

Figure 7.3: Box-Whisker plot of the performances achieved using Function 6.1 plotted against the level of noise introduced.

Noise	Performance (F-measure)	
	Function 6.1	Function 6.2
8	0.14	0.48
10	0.13	0.46
12	0.13	0.46
14	0.13	0.45
16	0.13	0.43
18	0.12	0.44
20	0.13	0.40
22	0.12	0.42
24	0.12	0.42
26	0.12	0.39
28	0.12	0.38
30	0.12	0.37
35	0.11	0.36
40	0.10	0.35
45	0.11	0.32
50	0.10	0.30
55	0.10	0.27
60	0.10	0.25
65	0.09	0.25
70	0.09	0.22
75	0.09	0.22
80	0.09	0.20
85	0.09	0.20
90	0.08	0.19

Table 7.2: The mean performance of the resulting grammars against the level of noise introduced.

Functions 6.1 and 6.2 for each level of noise are given in Table 7.2. A graph plotting the values from Table 7.2 is seen in Figure 7.1

In order to consider all results of all experiments and not just the mean values, we drew box-whisker plots based on all the performances our learned grammars achieved. Figure 7.2 shows the box-whisker plot drawn from the results of the experiments using Function 6.2 and Figure 7.3 shows the box-whisker plot drawn from the results of the experiments using Function 6.1

To investigate whether the splitting for 5-fold cross validation might have some influence on the result, we decided to present the separate results for each fold. Therefore we plotted the mean performance over each of the five folds for each noise-levels and both Functions 6.1 and 6.2. That graph can be seen in Figure 7.4. We can see two groups of

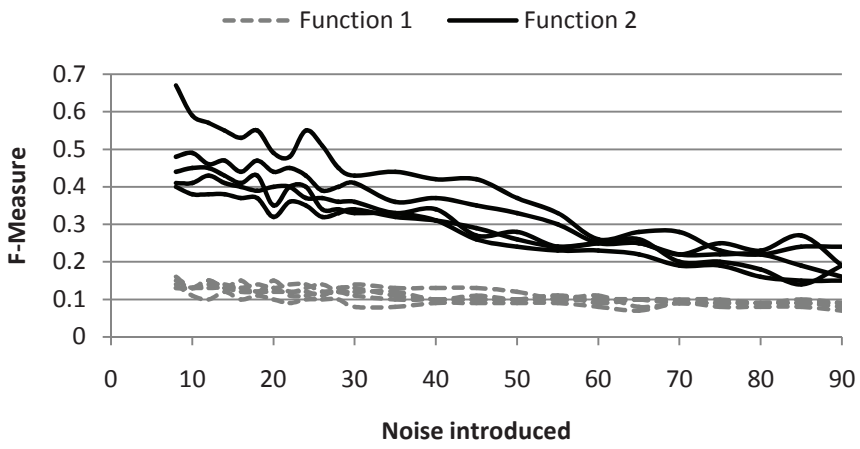

Figure 7.4: The mean Performance for each fold and each Function plotted against the level of noise introduced.

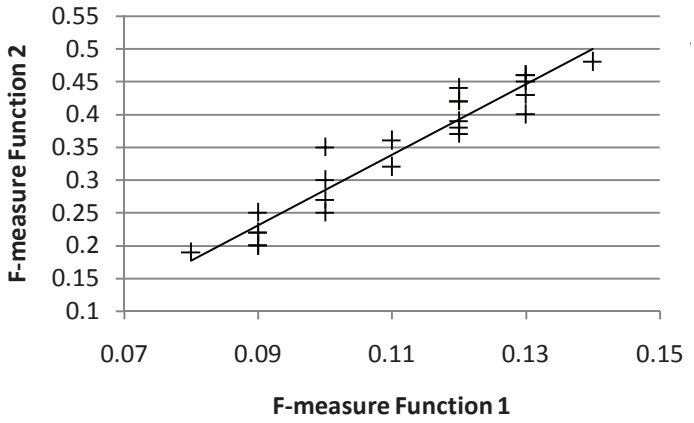

Figure 7.5: The mean Performance using Function 6.1 against the mean Performance using Function 6.2. It also contains a linear regression line found using the method of least squares. The equation for this line is: $y = 5.387x - 0.254$ and R-squared = 0.920.

lines: the five thicker lines represent the performance using L-modified Function 6.2 on all five folds, the five thinner lines represent the performance using the benchmark Function 6.1.

In order to determine if there might be a correlation between the decrease of performance using Function 1 and the decrease of performance using Function 2, we decided to plot the F-measures for both Functions against each other, with respect to the noise introduced. That graph can be seen in Figure 7.5.

7.5 Evaluation and Discussion

From the performance values given in Table 7.2 and from both Figure 7.1 and Figure 7.4, we can observe that at no given point are the results using the L-modified Function 6.2 lower than those of the benchmark Function 6.1. In fact, no matter how much noise we introduce, the performance using the L-modified Function 6.2 is never as low as the performance using the benchmark Function 6.1 without any noise.

Figures 7.1 and 7.4 suggest that the performance using Function 6.1 does not decrease by much, no matter how much noise is introduced, whereas the performance using Function 6.2 notably decreases steadily as more noise is introduced. The behaviour we expected to see while conducting this experiment is a reduction of performance using both Functions based on the levels of noise introduced. Function 6.2 does behave as expected while Function 6.1 seemingly does not. However if we focus our attention to Figure 7.5 which plots both sets of performances against each other, we can see that both performances are correlated (with an R-squared value of 0.920): As the performance of Function 1 decreases, the performance of Function 2 decreases as well. It is merely harder to notice in Figures 7.1 and 7.4 due to the low starting performance using Function 1, when compared with Function 2.

Figure 7.4 shows that the behaviour shown in Figure 7.1 holds true for each of the five folds. We can observe:

- For or each fold, the performance is always higher using Function 6.2.
- For each fold, the performance using Function 6.2 decreases as more noise is introduced while using Function 6.1 the performance does not seem to change.
- The performance does not change much between learning on the five different folds. There is one fold for which Function 6.2 produces slightly higher performance, but this does not seem significant.

Looking at the two box-whisker plots in Figures 7.2 and 7.3, we can further confirm the previous observations:

- No matter how much noise we introduce, the performance of Function 6.2 is always higher than that using Function 6.1.
- When using Figures 7.2, the performance decreases steadily as the level of noise is increased.
- When using Function 6.1, the performance does not seem to change, no matter how much noise is introduced.

We can also see from the box-whisker plots that a lot of the outliers are quite far away from the mean performances using both, Functions 1 and 2.

7.6 Conclusion

From the observations made from Figure 7.4, we can conclude that the process that is used in this work to split the dataset into the different folds has a very small effect on the resulting performance.

Considering the wide spread of values seen in the box-whisker plots in Figures 7.2 and 7.3 we can conclude that the algorithm in Table 7.1 which we used to insert noise into the dataset does so appropriately.

When introducing noise into a dataset, we would expect a decline of the performance over time. Looking at Figures 7.1, 7.2 or 7.4, we can observe that this expectation holds up when we are looking at Function 6.2, whose performance declines gradually when noise is introduced. There is no obvious tipping point where the performance just decreases drastically.

Looking at Figures 7.3 and 7.4 one might come to the conclusion that the performance using Function 6.1 does not really decline noticeably. In fact, it starts with an F-measure of 0.14 and it declines to an F-measure of 0.08. This decline is minimal, but considering that we start with such a low F-measure, the performance cannot decline by much from there. However we have shown in Figure 7.5 that the decline of performance using both Functions is in fact correlated. This shows that the performance declines gradually using both Functions 6.1 and 6.2.

In all our experiments, the performance achieved using the L-modified Function 6.2 was significantly higher than when using benchmark Function 6.1. This further supports the claim that L-Modification improves performance.

Based on all observations done during the course of the experiments reported in this chapter, we conclude that our L-modified Function 6.2 can cope well with noise. We see the results of this work as further confirmation and extension to the results presented in Chapter 6.

Chapter 8

Discussion and Future Work

8.1 Thesis Summary

In Chapter 2 we have introduced the field of Molecular Biology, gave an insight on some of the grammatical structures that can be used and finally explained the advantages of using grammars to describe biological sequences. Chapter 3 discussed model selection principles, in particular those related to the Minimum Description Length Principle. Then, in Chapter 4 we have introduced Machine Learning, in particular Inductive Logic Programming and why it is useful in the Molecular Biology domain. We have described how Inductive Logic Programming can be used to learn biological grammars from protein sequences. Finally we have identified a shortcoming of the traditional way this learning task was addressed: namely that it does not respect the length of the biological examples given in the training data adequately. A series of experiments was proposed to address this shortcoming, and Chapter 5 introduces all the materials that are required to undertake these experiments.

Then, at the start of Chapter 6 we have formalised our proposed way to overcome this shortcoming: *L-Modification*. We went on to undertake an empirical study to determine the potential of the proposed L-modification. The remainder of Chapter 6 presented the experiments that were conducted in this study and analysed their results. Chapter 7 then further extended these experiments and investigates the impact of L-modification when dealing with noisy biological data.

8.2 Discussion

The results presented in Chapters 6 and 7 indicate that L-modification is indeed a promising concept. Our L-modified ILP clause evaluation function has outperformed our benchmark in every experiment we conducted. Throughout our experiments we have always ensured that the predictive performance is measured on unseen test data. We did this by applying a 5-fold stratified cross-validation testing strategy. We have explained why predictive accuracy is not an adequate performance measure for induced models when learning from highly imbalanced datasets. Instead we have used precision, recall and the F-measure as our chosen performance measures.

A number of important conclusions were drawn in this thesis:

- Applying L-modification on our benchmark function and using the resulting L-modified function to learn biological grammars describing protein sequences of the neuropeptide precurser family always results in a significantly better performance that using the unmodified benchmark function.

- By grouping examples from our dataset, based on their length, into 3 disjoint subsets, we have shown a number of things:

    1. L-modification has increased performance on all 3 subsets. This effectively shows that L-modification improves performance when learning on short examples as well as for large examples.

    2. Achieved predictive performance is generally better when learning from shorter examples.

    3. It is possible that the range of the lengths of examples has an influence on the quality of the grammars learned. A higher range of lengths of examples could make it harder to learn grammars from those examples.

- When using L-modified evaluation functions, it takes Aleph longer to learn and the process is more computationally expensive when compared with the benchmark.

The improvements in the performance of the grammars therefore come at a price in computation.

- It is generally easier to learn from smaller examples. There is a correlation between the length of an example and the time and effort needed to learn from it.
- We have shown that as a consequence of L-modification, it is possible that some terms in a clause evaluation function might loose their influence on the result. This was not the case in the experiments conducted in this work, but when applying L-modification to other clause evaluation functions, this should be considered.
- Finally, we have shown that our L-modified Function copes quite well with Noisy data. We gradually inserted more and more noise into the dataset, but there was no tipping point where the performance plummeted. Instead the performance also decreased gradually.

Summing up these results we come to the final conclusion that a sequence-length sensitive approach to learning biological grammars using inductive logic programming is a useful and promising concept.

8.3 Future Work

This section presents a few suggestions on how the work presented in this thesis might be extended. Some of the ideas presented here could not be done because the author did not have the required access to resources or data, but most of the ideas presented here were left unaddressed simply because of a lack of time available for the project.

Apply L-modification to other ILP clause evaluation functions

It should be interesting to apply L-modification to other ILP clause evaluation functions which only use the count of examples to score the quality of learned hypotheses. For domains in which examples in the training data are of variable length, we would expect similar results to those presented in this work.

Apply L-modification to positive and negative learning

The logical next step for this work is to apply L-modification to clause evaluation functions and datasets that are less concerned with positive only learning but rather use positive and negative examples. Learning biological grammars using clause evaluation functions tailored for positive and negative learning and comparing their performance with that of their L-modified counterparts should give interesting results.

One potential dataset, which containins positive and negative examples, is be the GPCR protein sequences which were used in (Bryant et al. 2006).

Investigate real world transcription errors

The author is aware that the transcription errors that were introduced into the dataset in Section 7.2 do not precisely reflect the transcription errors that would be most likely to occur in real biological data. As part of the potential future work there could be an investigation into the sort of transcription errors that might occur in reality and then test our L-modification on such data.

Investigate the effect of noise caused by classification errors

The effect of noise caused by classification errors (See Section 7.2) could be investigated as well. In this work we have only considered transcription errors. As we were dealing with positive only learning, we already accommodated for the presence misclassified (or undiscovered) positive examples anyway. If this work could be expanded towards a positive and negative learning approach, using the respective clause evaluation functions and their L-modified counterparts, the investigation into classification errors might provide interesting results.

Other protein families

This work dealt only with neuropeptide precursor proteins. The author would like to see the results of applying the ideas presented in this work onto datasets which consist of other protein families.

As mentioned before, the GPCR protein sequences which were used in (Bryant et al.

2006) might be a potential candidate. Alternatively, the uniprot database (UniProtKB n.d.) could contain some protein sequences which could be compiled for future experiments. A *Protein Secondary Structure* sequential dataset can also be found in the *UC Irvine Machine Learning Repository* (UCI n.d.).

L-modification and learning grammars describing nucleic acid sequences

Furthermore, this work was concerned with learning grammars which can parse protein sequences, which are described through an alphabet consisting of 20 amino acids. As it was mentioned in Chapter 2, grammars have also been shown to be useful to describe nucleic acids; DNA and RNA sequences. These are described through an alphabet consisting of only four nucleotides. The author would like to see L-modification applied to learning grammars describing nucleic acids.

Potential sources for datasets containing DNA sequences could be the *UC Irvine Machine Learning Repository* (UCI n.d.) where *Promoter Gene Sequences* and *Splice-junction Gene Sequences* can be found. Both of these datasets have been used in the past to learn neural networks by the KBANN system (Towell, Shavlik & Noordewier 1990).

Evaluate the learned biological grammars

Finally we would like to investigate the usefulness of grammars learned using L-modified clause evaluation functions in the biological domain. We would like to have experienced biologists evaluate the grammars that were learned using our L-modified clause evaluation functions.

Learning from very long examples

As we have noticed during the experiments in Chapter 6.5, some unintuitive behaviour was noticed when learning from examples that contained more that 40 amino acids (characters). As of yet, we have no solid suggestions to explain why these examples actually needed less time to be learned on than some of their shorter counterparts. It might be that this is unique to Alpeh, or it might very well be that other ILP tools show the same behaviour. This could be investigated as part of the future work. Also, as we have not actually searched the literature with the aim of seeking to explain this, therefore, having a closer

look at some related work might give explanations. For example, we have noticed that (Charniak 1996) have omitted from their test set any sentences of length greater than 40, but have given no justification for doing so. This could be related. Either way, the ramifications of learning from very long examples should be investigated.

8.4 Epilogue

This thesis has presented a novel concept of modifying ILP clause evaluation functions in order to better deal with datasets which contain examples of highly variable lengths. It is our belief that the length of examples in the training data which is used to learn grammars, contains information which should not be ignored. After all, the aim of a learning system is to learn as much as possible from the data provided, therefore, ignoring such a crucial property of biological data seems inappropriate.

In this work, we had a particular task in mind: the learning of biological grammars from protein sequences. It is a common feature of protein sequences to display variation in their lengths. Within this domain we showed that our proposed L-modification proves to be a promising concept. However the ideas presented within this work focus first and foremost on the variable length of examples and their implication on induced grammars, and not on their biological applications. We expect the idea of L-modification to prove itself useful not just in learning biological grammars from protein sequences but also in other areas concerned with learning linguistic structures from examples of highly variable length.

Bibliography

Abe, N. & Mamitsuka, H. (1997). Predicting protein secondary structure using stochastic tree grammars, *Journal of Machine Learning* **29**(2-3): 275–301.

Agard, B. & Kusiak, A. (2005). Data mining for selection of manufacturing processes, *in* O. Maimon & L. Rokach (eds), *Data Mining and Knowledge Discovery Handbook*, Springer US, pp. 1159–1166.

AHSD (2010). *The American Heritage Science Dictionary*, Houghton Mifflin Harcourt Publishing Company.

Altschul, S. F., Gish, W., Miller, W., Myers, E. W. & Lipman, D. J. (1990). Volume 215, issue 3, 5 october 1990, pages 403-410, *Journal of Molecular Biology* **215**: 403–410.

Bairoch, A. & Apweiler, R. (1997). The SWISS-PROT protein sequence data bank and its supplement trEMBL, *Nucleic Acids Research* **25**(1): 31–36.

Bateman, A., Birney, E., Cerruti, L., Durbin, R., Etwiller, L., Eddy, S. R., Griffiths-Jones, S., Howe, K. L., Marshall, M. & Sonnhammer, E. L. (2002). The pfam protein families database., *Nucleic Acids Research* **30**(1): 276–280.

Bayes, T. (1764). An essay towards solving a problem in the doctrine of chances, *Philosophical Transactions of the Royal Society of London* **53**: 376–398.

Bratko, I. (2001). *Prolog (3rd ed.): programming for artificial intelligence, Chapter 19*, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

Bryant, C. H. & Fredouille, D. (2005). A parser for the efficient induction of biological grammars, *in* S. Kramer & B. Pfahringer (eds), 15^{th} *International Conference on*

Inductive Logic Programming: late-breaking paper track., University of Bonn, Bonn, Germany, pp. 3–8.

Bryant, C. H., Fredouille, D., Wilson, A., Jayawickreme, C., Jupe, S. & Topp, S. (2006). Pertinent background knowledge for learning protein grammars, *in* J. Furnkranz, T. Scheffer & M. Spiliopoulou (eds), *Proceedings of the 17th European Conference on Machine Learning*, number 4212 in *Lecture Notes in Artificial Intelligence*, Springer-Verlag, Berlin, pp. 54–65.

Burnham, K. P., Anderson, D. R. & Burnham, K. P. (2002). *Model selection and multimodel inference: A practical information-theoretic approach*, 2nd edn, Springer.

Carrillo, H. & Lipman, D. (1988). The multiple sequence alignment problem in biology, *SIAM J. Appl. Math.* **48**(5): 1073–1082.

Chaitin, G. J. (1966). On the length of programs for computing finite binary sequences, *J. ACM* **13**(4): 547–569.

Charniak, E. (1996). Tree bank grammars, *Proceedings of Thirteenth National Conference on Artificial Intelligence*, Vol. 2, American Association for Artificial Intelligence, Portland, Oregon, USA, pp. 1031–1036.

Chomsky, N. (1956). Three models for the description of language, *IRE Transactions on Information Theory* **2**: 113–124.

Chomsky, N. (2002). *Syntactic Structures*, 2nd edn, de Gruyter Mouton.

Cootes, A., Muggleton, S. & Sternberg, M. (2003). The automatic discovery of structural principles describing protein fold space, *Journal of Molecular Biology* **330**(4): 839–850.

Cussens, J. & Pulman, S. (2000). Experiments in inductive chart parsing, *in* J. Cussens & S. Dzeroski (eds), *Learning Language in Logic*, Vol. 1925 of *Lecture Notes in Computer Science*, Springer Verlag, Berlin, pp. 143–156.

Dassow, J. (2005). Contextual grammars with subregular choice., *Fundam. Inform.* pp. 109–118.

De Raedt, L., Kersting, K. & Torge, S. (2005). Towards learning stochastic logic programs from proof-banks, *in* M. Veloso & S. Kambhampati (eds), *Proceedings of Twentieth National Conference on Artificial Intelligence*, American Association for Artificial Intelligence, Pittsburgh, Pennsylvania, USA, pp. 752–757.

Dehaspe, L., Toivonen, H. & King, R. D. (1998). Finding frequent substructures in chemical compounds, *in* R. Agrawal, P. Stolorz & G. Piatetsky-Shapiro (eds), *4th International Conference on Knowledge Discovery and Data Mining*, AAAI Press., pp. 30–36.

Dsouza, M., Larsen, N. & Overbeek, R. (1997). Searching for patterns in genomic data, *Trends in Genetics* **13**(12): 497–498.

Falquet, L., Pagni, M., Bucher, P., Hulo, N., Sigrist, C. J., Hogfmann, K. & Bairoch, A. (2002). Protein data bank, *Nucleic Acid Research* **30**: 235–238.

Fredouille, D., Bryant, C. H., Jayawickreme, C. K., Jupe, S. & Topp, S. (2006). An ILP refinement operator for biological grammar learning, *in* S. Muggleton (ed.), 16^{th} *International Conference on Inductive Logic Programming*.

Freudenthal, H. (1961). The concept and the role of the model in mathematics and natural and social sciences, *Proceedings of the colloquium sponsored by the Division of Philosophy of Sciences of the International Union of History and Philosophy of Sciences*, D. Reidel Pub. Co.; Gordon and Breach Dordrecht, New York,, p. 194 p.

Goadrich, M., Oliphant, L. & Shavlik, J. (2004). Learning ensembles of first-order clauses for recall-precision curves: A case study in biomedical information extraction, *Proceedings of the 14th International Conference on Inductive Logic Programming (ILP*, pp. 98–115.

Grunwald, P. (1996). A minimum description length approach to grammar inference, *Springer Lecture Notes in Artifcial Intelligence* **1040**: 203–216.

Grunwald, P. D., Myung, I. J. & Pitt, M. (2005). *Advances in Minimum Description Length: Theory and Applications*, MIT Press.

Hart, P., Nilsson, N. & Raphael, B. (1968). A Formal Basis for the Heuristic Determination of Minimum Cost Paths, *IEEE Transactions on Systems Science and Cybernetics* **4**(2): 100–107.

Hogeweg, P. & Hesper, B. (1978). Simulating the growth of cellular forms. simulation 31. interactive instruction on population interactions, *Comput Biol Med* **8**: 319–27.

Huerta, M. (2000). NIH working definition of Bioinformatics and Computational Biology. http://www.bisti.nih.gov/CompuBioDef.pdf.

Hunter, L. (2004). Life and its molecules: A brief introduction, *AI Mag.* **25**(1): 9–22.

Jaynes, E. (2003). *Probability Theory*, Cambridge University Press.

Jonassen, I. (1997). Efficient discovery of conserved patterns using a pattern graph., *Computer applications in the biosciences: CABIOS* **13**(5): 509–522.

King, R. (2004). Applying inductive logic programming to predicting gene function, *AI Magazine* **25**(1): 57–68.

King, R., Karwath, A., Clare, A. & DeHaspe, L. (2001). The utility of different representations of protein sequence for predicting functional class, *Bioinformatics* **17**(5): 445–454.

King, R., Whelan, K., Jones, F., Reiser, P., C.H.Bryant, Muggleton, S., Kell, D. & Oliver, S. (2004). Functional genomic hypothesis generation and experimentation by a robot scientist, *Nature* **427**(6971): 247–252.

Kirchherr, Li & Vitanyi (1997). The miraculous universal distribution, *MATHINT: The Mathematical Intelligencer* **19**(4): 7–15.

Kolmogorov, A. N. (1965). Three approaches to the quantitative definition of information, *Problems of Information Transmission* **1**: 1–7.

Leung, S. W., Mellish, C. & Robertson, D. (2001). Basic gene grammars and DNA-chartparser for language processing of escherichia coli promoter DNA sequences, *Bioinformatics* **17**(3): 226–236.

Mamer, T. & Bryant, C. (2006). Improving biological grammar acquisition by considering the length of training examples, *16th International Conference on Inductive Logic Programming - Short Papers*, UDC Press service, Coruna, Spain., pp. 140–142.

Mamer, T., Bryant, C. & McCall, J. (2008). L-modified ILP evaluation functions for positive-only biological grammar learning, *in* F. Zelezny & N. Lavrac (eds), *Proceedings of the* 18^{th} *International Conference on Inductive Logic Programming*, number 5194 in *Lecture Notes in Artificial Intelligence*, Springer-Verlag, Berlin, pp. 176–191.

McCarthy, J. (1958). Programs with common sense, *Symposium on Mechanization of Thought Processes. National Physical Laboratory. Teddington, England*.

Mitchell, T. M. (1997). *Machine Learning*, McGraw Hill.

Mozetic, I. (1998). Secondary structure prediction by inductive logic programming, *Proc. 3rd Meeting on the Critical Assessment of Techniques for Protein Structure Prediction, CASP3*, pp. A–26.

Muggleton, S. (1991). Inductive logic programming, *New Gen. Comput.* **8**(4): 295–318.

Muggleton, S. & Feng, C. (1992). Efficient induction in logic programs, *in* S. Muggleton (ed.), *Inductive Logic Programming*, Academic Press, pp. 281–298.

Muggleton, S. H. (1995). Inverse entailment and Progol, *New Generation Computing* **13**: 245–286.

Muggleton, S. H. (1996). Learning from positive data, *in* S. H. Muggleton (ed.), *Proceedings of the 6th International Workshop on Indictive Logic Programming*, Vol. 1314, Springer Verlag, pp. 358–376.

Muggleton, S. H., Bryant, C. H., Srinivasan, A., Whittaker, A., Topp, S. & Rawlings, C.

(2001). Are grammatical representations useful for learning from biological sequence data? - a case study, *Journal of Computational Biology* **8**(5): 493–522.

Muggleton, S., King, R. & Sternberg, M. (1992a). Protein secondary structure prediction using logic-based machine learning, *Protein Engineering* **5**(7): 647–657.

Muggleton, S., King, R. & Sternberg, M. J. E. (1992b). Protein secondary structure prediction using logic-based machine learning, *Protein Engineering -Oxford-* **5**(7): 647.

Muggleton, S., Srinivasan, A. & Bain, M. (1992). Compression, significance and accuracy, *in* D. Sleeman & P. Edwards (eds), *Proceedings of the Ninth International Machine Learning Conference*, Morgan-Kaufmann, pp. 338–347.

Natarajan, B. (1991). *Machine Learning*, Springer.

Pereira, F. & Warren, D. (1986). Definite clause grammars for language analysis, *Readings in natural language processing* **1**: 101–124.

Pulman, S. & Cussens, J. (2001). Grammar learning using inductive logic programming, *Oxford University Working Papers in Linguistics, Philology and Phonetics*, Vol. 6, Oxford University, pp. 31–45.

Rissanen, J. J. (1978). Modeling by shortest data description, *Automatica* **14**: 465–471.

Rivas, E. & Eddy, S. R. (2000). The language of RNA: a formal grammar that includes pseudoknots., *Bioinformatics* **16**(4): 334–340.

Sakakibara, Y., Brown, M., Hughey, R., Mian, I. S., Sjolander, K., Underwood, R. C. & Haussler, D. (1994). Stochastic context-free grammars for tRNA modeling., *Nucleic Acids Res* **22**(23): 5112–5120.

Searls, D. B. (1993). String variable grammar: A logic grammar formalism for the biological language of DNA, *The Journal of Logic Programming* **24**(1-2): 73–102.

Searls, D. B. (1995). String variable grammar: A logic grammar formalism for the biological language of DNA, *The Journal of Logic Programming* **24**(1-2): 73–102.

Searls, D. B. (1997). Linguistic approaches to biological sequences., *Computer Applications in the Biosciences* **13**(4): 333–344.

Searls, D. B. (2002). The language of genes, *Nature* **420**: 211–217.

Searls, D. B. & Dong, S. (1993). A syntactic pattern recognition system for DNA sequences, *2nd International Conference on Bioinformatics, Supercomputing and Complex Genome Analysis*.

Shannon, C. E. (1948). The mathematic theory of communication, *Bell System Technical Journal* **27**: 379–423, 623–656.

Solomonoff, R. (1964). A formal theory of inductive inference, part 1 and part 2, *Information and Control* **7**: 1–22, 224–254.

Srinivasan, A. (1993). A learning engine for proposing hypotheses (Aleph). http://web.comlab.ox.ac.uk/oucl/research/areas/machlearn/Aleph.

Srinivasan, A., Muggleton, S. & Bain, M. (1994). The justification of logical theories based on data compression, *Machine Intelligence* **13**: 91–125.

Stahl, I. (1996). Compression measures in ILP, *Advances in Inductive Logic Programming*, IOS Press, pp. 295–307.

Swiniarski, R. (2000). Data mining methods in face recognition, *in* N. M. Nasrabadi & A. K. Katsaggelos (ed.), *Society of Photo-Optical Instrumentation Engineers (SPIE) Conference Series*, Vol. 3962 of *Society of Photo-Optical Instrumentation Engineers (SPIE) Conference Series*, pp. 52–59.

Taubes, G. (1996). Misfolding The Way to Disease, *Science* **271**: 1493–95.

Thomson, K. W., Guthrie, T. P. & Howard, D. G. (1912). *Treatise on natural philosophy*, Vol. 1, Cambridge, University Press.

Tomita, M. (1985). *Efficient Parsing for Natural Language: A Fast Algorithm for Practical Systems*, Kluwer Academic Publishers, Norwell, MA, USA.

Towell, G. G., Shavlik, J. W. & Noordewier, M. O. (1990). Refinement of approximate domain theories by knowledge-based neural networks, *Proceedings of the eighth National conference on Artificial intelligence - Volume 2*, AAAI'90, AAAI Press, pp. 861–866.

Turcotte, M., Muggleton, S. & Sternberg, M. (2001a). Automated discovery of structural signatures of protein fold and function, *Journal of Molecular Biology* **306**(3): 591–605.

Turcotte, M., Muggleton, S. & Sternberg, M. (2001b). The effect of relational background knowledge on learning of protein three-dimensional fold signatures, *Machine Learning* **1**(2): 81–96.

UCI (n.d.). UC Irvine Machine Learning Repository.
http://archive.ics.uci.edu/ml/datasets/.

UniProtKB (n.d.). Universal Protein Resource.
http://www.uniprot.org/.

Vijay-Shankar, K. & Joshi, A. K. (1986). Some computational properties of tree adjoining grammars, *Proceedings of the workshop on Strategic computing natural language*, HLT '86, Association for Computational Linguistics, Stroudsburg, PA, USA, pp. 212–223.

Wallace, C. S. & Boulton, D. M. (1968). An information measure for classification, *Computer Journal* **11**(2): 185–194.

Wallis, S. (2008). Searching treebanks and other structured corpora, *in* A. Ldeling & M. Kyt (eds), *Corpus Linguistics*, Vol. 48, Berlin, New York (Mouton de Gruyter), p. 738759.

Zelle, J. & Mooney, R. (1993). Learning semantic grammars with constructive inductive logic programming, *Proceedings of the Eleventh National Conference of the American Association for Artificial Intelligence (AAAI-93)*, American Association for Artificial Intelligence, Morgan Kaufmann, San Mateo, CA, pp. 817–822.

Appendix A

Concepts that are only briefly mentioned in this thesis

A.1 Finite State Automaton

A *finite state automaton* (FSA) or *finite state machine* (FSM) or simply a *state machine*, is a model of behaviour composed of a finite number of states, transitions between those states, and actions. A FSA can be in one of a finite number of states and when certain conditions are met (depending on the FSA and the input), it can transition into another state. When a FSA starts working it is in one, of possibly many, initial states. When a FSA stops working it can be in one, of possibly many, final states; if this is the case, then the FSA is said to recognize or accept the input (See Figure A.1). A finite state machine is an abstract model of a machine with a primitive internal memory.

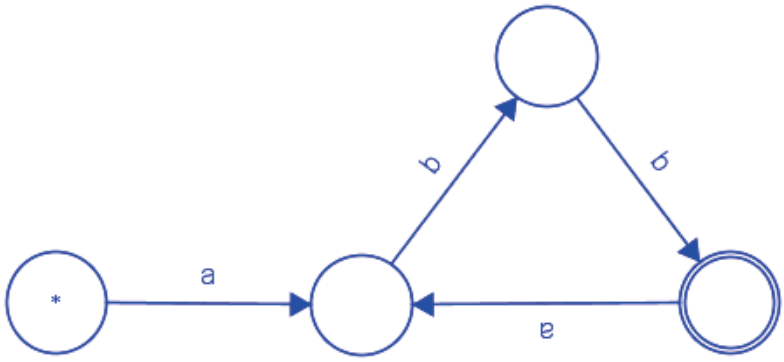

Figure A.1: This Finite State Automaton has an initial state (on the very left) and a final state (on the very right). Input given should correspond to the symbols on the arrow labels and follow the flow until the final state is reached. This particular FSA accepts any string that contains one or more iterations of `abb`.

A.2 Pushdown Automaton

A *pushdown automaton* (PDA) is a finite state automaton (FSA) (See Appendix A.1) that can make use of a stack. PDA and FSA differ in two ways:

1. A PDA can use the top of the stack to choose which state to transition to.

2. A PDA can manipulate the stack as part of performing a transition.

PDAs choose their transition path by considering a given input signal, current state, **and** the symbol at the top of the stack.

A.3 Linear Bounded Automaton

A linear bounded automaton (LBA) is a restricted form of a Turing machine (TM). Like the TM (See Appendix A.4) it possesses a tape made up of cells containing symbols from a finite alphabet, a head that can read from or write to one cell on the tape at a time and can be moved. The difference between a LBA and a TM is that using a LBA, only a limited contiguous portion of the input tape can be accessed by the tape-head, where using a TM, there are no such limitations.

A.4 Turing Machine

A turing machine (TM) is an abstract symbol-manipulating device. The concept of a TM is quite simple, still, they can be adapted to simulate the logic of any computer algorithm. TMs were first described by Alan Turing in 1936. TMs are a thought experiment about the limitations of mechanical computing and not actually a practical computing technology, therefore TMs were not actually built. Alone the study of their abstract properties gives a lot of insight into computer science and complexity theory.

A TM mathematically models a mechanical machine that possesses a tape made up of cells containing symbols from a finite alphabet. A TM has a tape-head that can read from or write to a cell on the tape at a time and can be moved.

A.5 Tree-adjoining Grammars

A Tree-adjoining grammar (TAG) is a specific grammar formalism which was defined by Aravind Joshi (Vijay-Shankar & Joshi 1986). Tree-adjoining grammars are similar to context-free grammars (See Section 2.3.1), but instead of rewriting strings, made up of symbols, TAGs rewrite the nodes of trees as other trees. TAGs are used in graph theory.

A.6 Stochastic grammar

A stochastic grammar is a type of grammar that incorporates the notion of probability into its rules. A stochastic grammar is sometimes also called a statistical grammar. In natural language processing stochastic grammars are especially useful to deal with longer sentences that can be highly ambiguous. Stochastic grammars are usually learned by machine learning and data mining tools which learn from large amounts of data.

A.7 Stochastic context-free grammar

A stochastic context-free grammar (SCFG), sometimes also called probabilistic context-free grammar, is a context-free grammar in which each production rule contains a given probability. This way, the problem of ambiguity can be solved by assigning a probability to a derivation by calculating the product of the probabilities contained in all the production rules used. SCFGs are used in Natural language processing and the study of RNA molecules.

A.8 Closed World Assumption

What is commonly referred to as the *closed world assumption* is a presumption that everything that is not known to be true is false. The opposite of the closed world assumption is called the open world assumption and states that just because we don't know something to be true, we cannot just assume that it is false.

The closed world assumption supports the *negation by failure* inference rule, stating that any predicate that cannot be proven to be true must be false.