TopLog: ILP using a logic program declarative bias

Transfer Report

José Carlos Almeida Santos

September, 2008

Contents

1 Introduction **3**

1.1 Thesis abstract 3

1.2 Introduction and research goals 3

1.3 Report overview 4

2 Background **7**

2.1 Machine Learning 7

2.1.1 Hypotheses space search 8

2.1.2 Estimating Hypothesis Accuracy 8

2.1.3 State of the art systems and applications 10

2.2 Logic Programming 10

2.2.1 Compilers and applications 13

3 Inductive Logic Programming **15**

3.1 Mode declarations 16

3.2 Comparison with other Machine Learning approaches 17

3.2.1 Limitations of attribute value learning 18

3.2.2 Rook-King vs King legality 19

4 TopLog: A declarative bias ILP system **23**

4.1 Mathematical preliminaries 23

4.1.1 Mode-Directed Inverse Entailment 24

4.1.2 Top-Directed Hypothesis Derivation 25

4.2 TopLog modules 26

4.2.1 From mode declarations to $\top$ theory 26

4.2.2 Hypothesis derivation 27
4.2.3 Learning algorithm 27
4.2.4 Building the final theory 28
4.2.5 Efficient cross-validation 29
4.3 Comparison with other ILP systems 30
4.3.1 Learning the Fibonacci series 31

5 Empirical evaluation of TopLog 33
5.1 Methods 33
5.2 Datasets 34
5.3 Results 35
5.4 Conclusions 36

6 PrioLog: A stochastic TopLog 37
6.1 PrioLog main algorithm 37

7 Conclusions and future work 39
7.1 Upgrade $\top$ theory to an SLP 40
7.2 Parallelization 40
7.3 Relevant applications 40

Bibliography 43

Keywords: TopLog, Inductive Logic Programming, Machine Learning

Chapter 1

Introduction

1.1 Thesis abstract

The aim of my Ph.D. thesis is to create a new framework for describing the hypotheses search space of an Inductive Logic Programming (ILP) system. This new framework allows for a first-order logic description of the search space which creates several opportunities. First, it is both more expressive and elegant than current mode directed ILP systems [Mug95a]. Second, it allows for the hypotheses search space to be more precisely defined thus shortening it. Thirdly, it can be upgraded to a Stochastic Logic Program [Mug95b] which provides a natural and efficient way to sample the hypotheses search space. The combination of these benefits leads to more efficient ILP systems.

1.2 Introduction and research goals

Although Inductive Logic Programming can have other usages such as program synthesis, it is normally used as a logic based supervised machine learning algorithm. The usual setting for its application is, **given:** 1) a set of background knowledge facts B, 2) a set of examples E, **find:** a set of hypotheses H, such that $B, H \models E$. H, the induced model, is a set of Horn rules thus being easily comprehensible by a human.

Inductive Logic Programming has had several successful practical applications specially in the biology domain (e.g. [FMPS98, KMSS96, SKMS97, SMKS96, TMS98]). However, a major practical problem for widespread use is its lack of efficiency. Current ILP systems (e.g. [Mug95a], [Sri07]) take too long to build models for many interesting real world datasets. The main reason for the significant amount of computational time required is the size of the hypotheses search space. For any non trivial dataset, the hypotheses search space is well beyond what can be searched, even incompletely and with heuristics, in a reasonable time.

Our Ph.D. thesis contribution proposes to alleviate this problem in two ways.

Firstly, we change the way the hypotheses search space is defined. In current ILP systems the hypotheses search space is defined through mode declarations, specifying the literals that may appear in the body of any valid hypothesis. This mode declarations are meta-logical and it is purpose is to list the predicates allowed in an hypothesis.

In our setting, the mode declarations are replaced by first-order logic $\top$ theory. The $\top$ theory can be viewed as a form of first-order declarative bias which defines the hypothesis space, since each hypothesized clause must be derivable from $\top$. The use of the $\top$ theory in TopLog is also comparable to grammar-based declarative biases [Coh94]. However, compared with a grammar-based declarative bias, $\top$ has all the expressive power of a logic program, and can be efficiently reasoned with using standard logic programming techniques.

The SPECTRE system [BIA94] employs an approach related to the use of $\top$. SPECTRE also relies on an overly general logic program as a starting point. However, unlike the TopLog system described in this paper, SPECTRE proceeds by successively unfolding clauses in the initial theory. TDHD is also related to Explanation-Based Generalization (EBG) [KCM87]. However, like SPECTRE, EBG does not make the key MDHD distinction between the $\top$ theory and background knowledge. Moreover, EBG is viewed as a form of deductive learning, while the clauses generated by TDHD represent inductive hypotheses.

Having the hypothesis space defined through a first-order logic $\top$ theory allows for an elegant and more specific way of defining the hypotheses search space. This, just by itself, helps decrease the hypotheses search space because the format of an hypothesis can now be more precisely described.

The second way in which we alleviate the hypotheses search space problem is to upgrade the $\top$ theory definition from a logic program to a Stochastic Logic Program (SLP)[Mug95b]. Having the $\top$ theory as a Stochastic Logic Program allows for a stochastic derivation of hypotheses thus providing a natural way for sampling the hypotheses search space. Furthermore, this setting makes it possible to dynamically change the sampling bias by updating the SLP probabilities.

Finally we aim to show how the combination of these improvements are translated into significant benefits when applied to existing machine learning problems.

1.3 Report overview

This report is organized as follows. In chapter 2, background material on machine learning and logic programming is covered. Chapter 3 introduces Inductive Logic

Programming and its benefits over propositional learning approaches. In chapter 4, a new ILP system, TopLog, implementing some of the ideas described in this report is presented. In chapter 5 an empirical evaluation is performed against Aleph [Sri07], a state of the art ILP system. In chapter 6 we present the overall conclusions and a plan for future work.

Chapter 2

Background

In this chapter we introduce some background concepts on the broader areas of Machine Learning and Logic Programming. These two areas are the parents of Inductive Logic Programming (ILP). The aim of this overview is two fold: be self contained by providing the unfamiliar reader with enough background to understand this thesis and to put ILP and our contribution into perspective.

For a detailed overview we recommend [Mit97] for Machine Learning and [Llo87] for Logic Programming.

2.1 Machine Learning

Machine Learning is a broad sub-field of Artificial Intelligence, its aim is to devise algorithms that allow automated learning. Using [Mit97] definition, a computer program is said to **learn** from experience E with respect to some task T and performance measure P, if its performance at task T, as measured by P, improves with experience E.

For example, a computer program that learns to play chess, might improve its performance *as measured by its ability to win* at the *task of playing chess* through *experience obtained by playing games against itself.* To have a well defined learning problem we need to identify these three components: task, performance measure and training experience. Learning consists on acquiring general concepts from specific training examples. For example, acquiring the concept of bird through observing a small labelled sample of animals.

When learning the target concept, the learning algorithm is presented a set of training examples, each consisting of an instance e from the set of possible examples E, along with its target concept value $c(e)$. The problem the learner solves is to find an hypothesis, h, such that $h(e) = c(e)$. Thus, learning involves search through a space of possible hypotheses to find the hypothesis that best fits the available training examples and other prior knowledge or constraints.

Notice that because the only information available about the target concept c

is its value over the training examples, all learning algorithms have to assume that any hypothesis that approximate the target concept well over a sufficiently large set of training examples will also approximate the target function well over unobserved examples. This is a critical assumption in machine learning.

2.1.1 Hypotheses space search

A well designed machine learning algorithm will eventually find the target concept if *enough error-free training examples are provided and the hypotheses space contains the target concept.* Learning algorithms diverge mostly on how they characterize the hypotheses space and how it is searched. Any learning algorithm has to heavily restrict the hypotheses space in order for the search to be tractable.

To illustrate the problem of the hypotheses space complexity, consider the following argument. If each example has N features and each feature may assume one of K distinct values, then the number of possible unique instances (i.e. examples) is, clearly, N^K. The number of unique concepts that can be defined over this example set is equal to its number of distinct subsets (i.e. the power set of the example set). This is because a concept can be regarded as the set of examples it entails. The cardinality of the power set of a set X is $2^{|X|}$, thus the number of unique concepts is 2^{N^K}.

This value is doubly exponential and is quickly beyond reach even for relatively small values of N and K. It is clear that only a tiny fraction of the hypotheses space can be searched in reasonable time, thus the need to heavily restrict the hypotheses to consider.

Machine learning algorithms thus have to make another assumption: the concept to be learned lies in its restricted hypotheses space. Obviously this is not generally true but, in practice, very often good approximations can be found. For instance, in a decision tree hypotheses can be regarded as disjunctions of conjunctions in propositional logic. In ILP it is similar but with first-order logicİn a neural network, hypotheses are a neuron layer configuration with the respective weights between the neurons and in a support vector machine, hypotheses are a set of support vectors.

2.1.2 Estimating Hypothesis Accuracy

Evaluating the accuracy of hypotheses is fundamental in machine learning. Important questions to answer are: 1) given the observed accuracy of an hypothesis in the training data S, how well does this estimate the accuracy on unseen (i.e. testing) data? 2) how to compare two distinct learning algorithms?

Statistical theory allow us to answer the first question if we assume the examples have been independently sampled and there are at least 30 of them. If the above assumptions hold and no further information is available, then the most probable value for $error_D(h)$ is $error_s(h)$. Furthermore, we can estimate with arbitrary confidence that the true $error_D$ lies in the interval:

$$error_D(h) = error_s(h) \pm Z_N\sqrt{\frac{error_s(h).(1 - error_s(h))}{n}}$$

Z_N is a constant taken from the standard Normal distribution. The area under $[-z, z]$ on the standard Normal distribution is N. This N can be directly translated to a confidence level. For instance, for $Z_N = 1.96$, the area under $[-1.96, 1.96]$ in the standard Normal distribution is 0.95.

E.g., if we use a Z_N value of 1.96[1] we can say with 95% confidence that the true error $error_D$ lies in the interval $[error_s(h) - 1.96\sqrt{\frac{error_s(h).(1-error_s(h))}{n}}, error_s(h) + 1.96\sqrt{\frac{error_s(h).(1-error_s(h))}{n}}]$.

In [Mit97] a handy rule of thumb is presented to check if it is appropriate to use the above approximation is:

$$n.error_s(h).(1 - error_s(h)) \geq 5$$

In order to answer question two, i.e. whether a learning algorithm is more accurate than another for a certain problem, we normally perform a k-fold cross validation. The idea is to divide the data in k disjoint subsets: T_1, T_2, ..., T_k of equal size of at least 30 elements. Then, both h_1 and h_2 are executed k times, using all but one of the k subsets as training set and the remained as test set.

This execution results in k pairs of accuracies which are used to perform a one or two tail, paired t-test. The implicit null hypothesis being tested is either that the hypothesis have identical accuracy (two tailed t-test) or that one hypothesis has not higher accuracy than the other (one tailed t-test).

T-test returns a p-value which is to be interpreted as how likely it is to observe results at least as extreme as the ones we observe were the null hypothesis true. If the p-value is too low, normally below 0.05 or 0.01, we reject the null hypothesis.

For instance, if we have two learning algorithms, e.g. C5.0 and TopLog, solving a particular problem, and performed 3-fold cross validation, the accuracies could look something like, respectively: $\langle 0.655, 0.636, 0.647\rangle$ and $\langle 0.654, 0.634, 0.644\rangle$. The two paired t-test returns a p-value of 0.12 so we would not reject the null hypothesis and would consider the accuracies to be identical.

[1] A few other values for Z_N and N are: $\langle 1.00, 68\%\rangle$, $\langle 1.64, 90\%\rangle$, $\langle 2.58, 99\%\rangle$.

2.1.3 State of the art systems and applications

Standard machine learning algorithms like decision trees, artificial neural networks or, more recently, support vector machines, have been quite successful at solving problems from a broad range of areas such as search engines, medical diagnosis, bioinformatics, credit card fraud, speech and handwriting recognition, drug design, object recognition in computer vision, game playing and robot locomotion [Wik].

Each application may have different requirements for the models (i.e. the recipe to classify unseen examples) the learning algorithms generates. Although often the most important feature a model should have is high accuracy on unseen examples, for some applications like medical diagnosis or drug design it is critical that the models show the reasoning behind the classification.

In the respect of the understandability of the generated models, the learning algorithms can be divided in two major groups. Ones that generate 1) human uncomprehensible (i.e. black box) models and 2) human readable models, often close to natural language. In the first group are, e.g., artificial neural networks and support vector machines. In the second group are decision trees and inductive logic programming.

In general, for the same problem, black box models yield slightly higher accuracies than understandable models because there are much fewer constraints on what a model may be (e.g. there may be complex combination of weighted attributes). However, as we shall see and explain why in section 3.2, there are problems where ILP, despite still generating understandable models, yields better accuracies than state of the art black box learners.

2.2 Logic Programming

Logic Programming (LP) is a programming paradigm based in first-order logic. In LP, contrary to imperative programming languages, programs are expressed as a set of clauses and known facts. Resolution [Rob65] is then used to automatically prove queries and, as an intermediate result, bind logic variables to values that satisfy the query.

First-order logic is a rich formal deductive system with far more expressive power than propositional logic (which is used, e.g., in classical decision trees). Its added expressiveness allow to distinguish between "things" and assertions about "things", denote the same "thing" (term) and concept (predicate) everywhere by the same symbol.

A subset of first-order logic is at the core of the Prolog programming language.

This first-order logic subset, briefly explained here, allows automated formulae deduction (or automated theorem proving) and, as we will see, is also at the core of any ILP system.

First-order logic has two aspects: syntax and semantics. Syntax concerns well-formed formulas and the semantics is concerned with the meaning attached to the formulas. We start by the essential syntactic definitions and then introduce the semantics of first-order logic.

Definition 2.1. An *alphabet* of first-order logic consists of the following:

1. A set of functions each with an associated non negative arity. In the particular case of arity 0 it is usually called a constant.
2. A set of variables.
3. A non-empty set of *predicate symbols* each with an associated natural number called *arity*.
4. The *connectives* $\neg$, $\wedge$, $\vee$, $\rightarrow$ and $\leftrightarrow$.
5. The *quantifiers* $\forall$ and $\exists$.
6. The *punctuation symbols* '(', ')' and ','.

Definition 2.2. A *term* is defined recursively as either:

1. A variable.
2. A constant (i.e. a 0-ary function symbol).
3. If f is an n-ary ($n > 0$) function symbol and $t_1, \ldots, t_n$ are terms then $f(t_1, \ldots, t_n)$ is a term.

Definition 2.3. A *(well-formed) formula* is defined as follows:

1. $P(t_1, \ldots, t_n)$, where P is a predicate symbol with arity n and $t_1, \ldots, t_n$ are terms.
2. $(\neg F)$, $(F \wedge G)$, $(F \vee G)$, $(F \rightarrow G)$, $(F \leftrightarrow G)$, where F and G are formulas.
3. $\exists x F$ and $\forall x G$, where x is a variable symbol and F is a formula.

Definition 2.4. The *first-order language of a given alphabet* is the set of all well-formed formulas which can be constructed from the symbols of the alphabet.

It is now appropriate to discuss the semantics of first-order logic. However, first-order logic is too expressive being even impossible to know [2], for an arbitrary set

[2]It is an undecidable problem in the general case

of formulas Σ and an arbitrary formula ϕ, if $\Sigma \models \phi$. However, if we restrict all formulas in our language to be horn clauses (i.e. clauses which have at most one positive literal) then, as showed by [Kow74], there are proof procedures like SLD-resolution[3] which are sound and refutation complete. This allows the construction of automated theorem provers, hence permitting (a subset of) first-order logic to be used as a programming language and technology.

A refutation of a goal G within a program P consists in deriving the empty clause $\Box$ as the last goal in the derivation. A refutation fails when a goal that does not unify with the head of any clause in P is derived. By refutation complete it is meant that, if a refutation exists, it will be found[4] in a finite number of steps. If a refutation of a goal G is found, then we can conclude $\neg G$.

Another important concept is *unification*. Given two terms t_1 and t_2 we are interested in finding, if it exists, an assignment of values to variables in t_1 that unifies t_1 and t_2.

Definition 2.5. A *definite program clause* is a clause of the form $A \leftarrow B_1, \ldots, B_n$ which contains precisely one atom (A) in its consequent. A is called the head and $B_1, \ldots, B_n$ is called the body of the program clause.

We will now introduce some logic concepts which are of particular interest for Prolog.

Definition 2.6. Let L be a first-order language. The *Herbrand universe*, U_L, is the set of all ground terms which can be formed out of the function symbols appearing in L. In the particular case L has no constants (i.e. functions with arity 0), an arbitrary constant is added to form ground terms.

Definition 2.7. Let L be a first-order language. The *Herbrand base*, B_L, is the set of all ground atoms which can be formed by using predicate symbols from L with ground terms from the Herbrand universe as arguments.

Definition 2.8. Let L be a first-order language. A *Herbrand interpretation*, is a subset of the B_L, that is, an assignment of truth values to atoms in B_L. An *Herbrand model* is an Herbrand interpretation that makes all formulas (or clauses in a logic program) in L true.

[3]SLD-Resolution is based on the resolution inference rule by [Rob65]

[4]In standard Prolog interpreters this does not hold because of Prolog's depth-first search leftmost computation rule

The least Herbrand model is the set of all ground atomic logical consequences of the program.

Prolog, created in the early 1970s, is a logic based programming language implementing the above mentioned concepts [5]. It is declarative in the sense that the programmer only has to express what he knows about the problem domain (i.e. facts and relationships between terms) and the built-in resolution mechanism takes care of the control part. The solution to the problem is an assignment of values to free variables that satisfy the program clauses. A solution is thus an Herbrand model of the logic program.

2.2.1 Compilers and applications

There are a number of open source and commercial Prolog compilers. The most well known commercial Prolog currently is Sicstus Prolog, the descendant of Quintus Prolog. As for open source implementations, SWI-Prolog is the better known and has, by far, the largest developer community. Though robust, SWI is not an efficient Prolog implementation being several times slower than Sicstus in many benchmark tests.

A less well known, with a very small development team, but nevertheless mature and well documented Prolog compiler is YAP [Cos08]. YAP is the fastest open source implementation, even faster than Sicstus. YAP supports multiple argument indexing [CSL07], even in dynamic predicates [dSC07], which is crucial for top performance in ILP engines.

TopLog supports these three Prolog interpreters and in several ILP benchmarks between the systems, the relative speed was roughly: Sicstus 10x slower than YAP and SWI 10x slower than Sicstus.

Although a Turing complete programming language and general purpose, Prolog is mainly used in artificial intelligence applications. Prolog can be used for knowledge representation, constraint logic programming and natural language processing.

[5] Due to practical reasons Prolog introduces some changes to the first-order logic semantics. It introduces the cut (!) operator, which has no logical semantics, to prevent backtracking and reduce the search space. By default it also eliminates the occurs check in the unification algorithm.

Chapter 3

Inductive Logic Programming

In this chapter we cover basic concepts on ILP. A thorough reference is [NCdW97].

The purpose of Inductive Logic Programming (ILP) is, in its simplest form, to induce (i.e. discover) the definition of a predicate by observing examples of its input/output behavior. Normally the predicate being induced interacts with other predicates present in the logic program (i.e. the background knowledge). The observed examples are also often of two distinct types: positives and negatives. Positive examples should be entailed by the induced predicate and negative examples should not.

More formally the usual ILP setting is denoted as:

Given: a finite set of clauses B and two additional sets of clauses E^+ and E^-

Find: a theory Σ such that $\Sigma \cup B$ is correct with respect to E^+ and E^-.

A theory which is both complete and consistent is called correct. The induced theory is complete if it implies all the positive examples and consistent if no negative examples are implied. Inducing a correct theory is the ultimate purpose of an ILP system but in real world problems this is often not possible due to noise in the observations or errors in the background knowledge. Note that even with error free background knowledge and examples it is not always possible to build a correct theory (theorem 9.9 of [NCdW97]).

Also notice that the trivial theory $\Sigma = E^+$ would always be correct (and lengthy) but is of zero predictive power since no concept has been learned and thus any unseen example will always be classified as negative. The purpose is to find the minimum description length (i.e. as small as possible) which is as close as possible to being a correct theory.

To achieve this ILP systems evaluate the merit of a theory by a compression measure which takes into account the number of positive and negative examples covered and the length of its description. A compressive theory is one that covers more positive examples then the sum of the negatives covered plus the length of the theory itself.

There are two main approaches to find the correct theory Σ: top-down and bottom-up. A top-down approach starts with an overly general theory and specializes it to cover fewer examples. With bottom-up approaches the starting theory is overly specific and is generalized to cover more examples. In both cases an important concept is the one of refinement operator which performs a small step (e.g. add or remove an atom from the body) of a clause in order to generalize or specialize it.

There are several ILP systems. Two of the widely used systems are Progol [Mug95a] and Aleph [Sri07]. Aleph tries to integrate ideas from many systems and, with the proper settings, can emulate Progol. Aleph's main algorithm, which is the same as Progol, is:

1. Randomly select a positive example to be generalized
2. Build most-specific-clause, $\bot$, for the example
3. Heuristic top-down search to find a clause more general than $\bot$ but bounded by it
4. Remove examples covered by the clause found in step 3 (clause now belongs to theory)
5. If more positive examples exist go to step 1 otherwise stop outputting the theory

This relatively simple algorithm implements well the normal ILP setting but the search lattice defined by the $\bot$ clause may be too large thus reducing the efficiency. Also it is sensitive to example order and takes too greedy decisions as to which clauses should belong to the induced theory.

As we will see in the next chapters, some of TopLog's advantages are: irrelevance of example order, global optimization of the theory after all hypothesis are available (possible leading to higher accuracy), efficient cross-validation. The main disadvantage is that the hypothesis derivation step inside the $\top$ theory is not heuristically guided which may lead to many non interesting hypothesis generated (decreased efficiency) or exhaust the search resources without finding good hypotheses (decreased accuracy).

3.1 Mode declarations

Mode declarations purpose in a ILP system is to delimit the hypotheses search space. They characterize the format of a valid hypothesis. There are two types of mode

declarations: head and body. Mode head declarations, *modeh*, inform what is the target predicate the ILP system is supposed to induce (i.e. the head of a valid hypothesis) and mode body declarations, *modeb*, inform what literals may appear in the body of such hypotheses. Normally mode body declarations refer to predicates defined in the background knowledge but, in the case of recursive theories, they can also refer to the target predicate being induced.

Mode declarations also inform the ILP system the types, IO modes and recall of the predicate arguments. The predicate recall is the maximum number of times the predicate may succeed and thus possibly yielding alternative values in its output variables and constants.

For instance: *modeh(1, active(+molecule)), modeb(5, atom(+molecule, -atomid, #atomtype)*, means that 1) target predicate is *active/1* and it is called with an instantiated variable of type *molecule*; 2) *atom/3* may appear in the body of an hypothesis, has an input variable of type *molecule*, an output variable of type *atomid* and a constant of type *atomtype. atom/3* may succeed up to 5 times.

TopLog supports mode declarations for compatibility with MDIE ILP systems but does not require them. Instead the user is encouraged to describe the hypotheses search space in a grammar like way.

3.2 Comparison with other Machine Learning approaches

Although ILP is a general purpose machine learning algorithm it is not well suited for all machine learning problems. More specifically, if the problem to be solved is mainly propositional (i.e. an instance is represented by a vector of independent attributes) then, very likely, a decision tree could be applied with results identical to an ILP algorithm with the significant advantage of requiring orders of magnitude less computing power. Other type of problems were ILP is particularly ill-suited is when there is no requirement for the model to be understandable and the features are mostly real numbers. In this situation a Support Vector Machine would likely yield better accuracy and require less computing power as well.

This being said, it is now time to illustrate where the power of ILP lies. We will do so by showing the limitations of attribute value learning and compare it with relational learning. Finally we will solve the “Rook-King vs King legality” classification problem with ILP and other two machine learning approaches: decision trees and support vector machines. The classification problem presented in 4.3.1 also highlights the problems of attribute value learning.

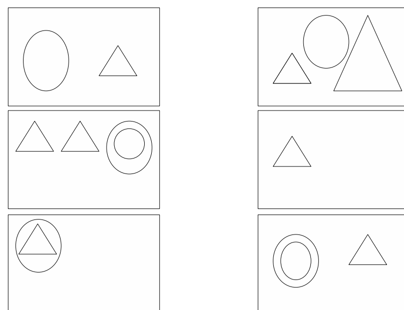

Figure 3.1: Bongard figures

The machine learning systems we use for these problems are C5.0 one of the most mature commercial decision trees. C5.0 is the successor of the well known C4.5 [Qui93], both by Ross Quinlan. For support vector machines we have used LibSVM [CL01], also a very known SVM package. As for the inductive logic programming system we have used TopLog, which is presented in the next chapter of this thesis. Other ILP systems such as Aleph or Progol would have yielded identical results.

3.2.1 Limitations of attribute value learning

Standard machine learning algorithms (e.g. decision trees, neural networks, support vector machines) require examples to be described as a vector of attributes. This simplistic approach can lead to serious difficulties in example representation as is clearly demonstrated by the class of Bongard problems [Bon70].

The goal of a Bongard problem is to classify two-dimensional scenes consisting of sets of nested polygons (e.g. triangles, rectangles, ellipses). Figure 3.1 illustrates the problem.

If the relative order, type and inside status, of the polygons inside the main rectangle is important then it would be extremely cumbersome to describe these six examples using simply a vector of attributes. The first problem is that, since examples have different number of polygons, one would have to consider the vector of attributes for all examples to be of the size of the largest example. For the smaller examples, the extra attributes would be left empty or with a dummy value. A more serious problem is that for any arbitrary property (e.g. is polygon A inside polygon B?) considered relevant there is an explosion of attributes, most of them with dummy values as well.

In contrast, with a first-order logic representation it is very natural to represent an arbitrary example. Eg. the upper left example could be represented by the following Prolog program: *rectangle(pol1). circle(pol2). triangle(pol3). contains(pol1,*

pol2). contains(pol3, pol1).

3.2.2 Rook-King vs King legality

Consider all chess boards where there the only pieces on board are a white rook, a white king and a black king and is white's turn to play. Some of these boards represent legal chess positions and some do not. For instance, any board where the black player is in check is illegal because after its move a player cannot remain in check. Also, any board where more than one piece share the same square is also illegal.

The Rook-King vs King problem [BM94] is to, given a chess board with only these three pieces determine if it is a legal or illegal one. A piece position is described by two integers (row and column) in the range of 1..8. The board is then described by six integers in any pre-arranged order [1].

This toy problem has the advantage of having a well defined example distribution, being possible to build all the training examples and be noise free. There are exactly $(8 * 8)^3 = 262,144$ unique chess boards for our problem and we can easily determine for any one of them what is its legality status. Of these 262,144 boards, 175,392 (negative examples) are legal and 86,752 are illegal (positive examples). We consider the target concept to be *illegal* rather than *legal* because it is easier to model illegality.

Despite the apparent simplicity, the correct target definition for this problem (i.e. board illegality) is not trivial as it may look at first. There are difficult to represent cases such as: *A chess board where the white rook and black king are on the same line (i.e. row or column) may be legal if the white king is in between, blocking the rook.*

The purpose of the learning algorithm in this problem is to, given chess board representations with the three pieces mentioned (i.e. 6 integers), its legality status (i.e. the target concept), build a classification model to predict the legality of unseen chess boards. No further attributes were given to the classifiers nor any background knowledge was provided to TopLog.

Each classifier was executed with default settings and three models were built with increasing number of training examples: 0.1% (262 instances), 1% (2,621 instances) and 10% (26,214 instances) of the example space. Ten fold cross validation was performed with the training data. The reported results are the average accuracy on the ten folds and the respective standard deviation. Accuracy is defined as

[1] We will use the order: white rook row, column, white king row, white king column, black king row, black king column.

the number of correct predictions over the total number of predictions. The default classifier, which says that all boards are legal, has an accuracy of 66.91%. Table 3.1 presents the results.

Algorithm	Accuracy A	Accuracy B	Accuracy C
C5.0	67.5% ± 2.9%	70.7% ± 0.8%	85.4% ± 0.9%
LibSVM	67.8% ± 3.7%	74.9% ± 0.8%	82.6% ± 0.3%
TopLog	91.0% ± 7.7%	91.9% ± 1.1%	91.8% ± 0.5%

Table 3.1: Rook-King vs King legality results

The reason ILP does not get 100% correct the legality status concept is because we provided no extra background knowledge making it impossible for it to find the subtler cases we mentioned above. Nevertheless, even with few training examples it built already the most accurate model possible given no background knowledge was provided. Figure 3.2 shows the four rules TopLog discovered.

The first rule says that whenever the white rook and the black king are in the same row it is an illegal board. Rule two is similar but for the column. Rule three says that whenever the white pieces are in the same square the board is illegal. Rule four says that whenever the rooks are in the same square the board is illegal.

When we contrast the simplicity of these rules with rules the decision tree generates, the advantage of ILP is clear (even not taking accuracy into account). C5.0 generates on average 75 rules (!) and the rules give almost no insight of what is the underlying concept. For instance, one of the rules C5.0 generated was *if (white rook column>2) and (white king row>6) and (white king column>2) and (black king row>6) and (black king column>3) then illegal board.*

The problem with propositional decision tree rules is that they cannot capture the concept that two features should have the same value. This leads to an explosion of rules and poor generalization. If new examples are taken from a 16x16 board, the decision tree rules would behave quite poorly whereas the ILP rules would be as accurate as in the 8x8 board.

Support vector machines, without a specialized kernel[2], have the same issue as decision trees. That is, they generate too many support vectors (i.e. the SVM equivalent of a rule), and do not generalize well in this problem. Furthermore, the model an SVM generates is not human comprehensible.

[2] An SVM kernel is a similarity function between 2 instances

```
Rule #1, PosScore=2898 (2898 new), NegScore=50 (50 new)
Accuracy = 98.3% Coverage = 36.8%

illegal(A,_,_,_,A,_).

Rule #2, PosScore=2877 (2524 new), NegScore=60 (60 new)
Accuracy = 98.0% Coverage = 36.6%

illegal(_,A,_,_,_,A).

Rule #3, PosScore=399 (301 new), NegScore=0 (0 new)
Accuracy = 100% Coverage = 5.1%

illegal(A,B,A,B,_,_).

Rule #4, PosScore=389 (301 new), NegScore=0 (0 new)
Accuracy = 100% Coverage = 4.9%

illegal(_,_,A,B,A,B).
```

Figure 3.2: TopLog model for Rook-King vs King problem

Chapter 4

TopLog: A declarative bias ILP system

This chapter introduces TopLog, a new prototype ILP system being developed for my Ph.D. thesis. TopLog incorporates new ideas on hypotheses generation techniques (i.e. Top Directed Hypothesis Derivation) which, coupled with other techniques described below, allow a significant efficiency improvement in the state of the art ILP technology.

The TopLog ILP system (source code, examples and manual) is freely available at http://www.doc.ic.ac.uk/∼jcs06/TopLog for academic use.

4.1 Mathematical preliminaries

In this section we survey Mode Directed Inverse Entailment (MDIE) and then introduce the Top Directed Hypothesis Derivation (TDHD) framework. We start by introducing the notation used in this section.

We will use standard notation from Inductive Logic Programming [NCdW97] to describe general clauses, Horn clauses, definite clauses, entailment, resolution and subsumption. Special notation will be used for SLD derivations and refutations as follows.

Definition 4.1. SLD derivation. A series of leftmost selection resolution steps involving ordered Horn clauses is referred to as an SLD derivation. An SLD derivation involving n clauses will be denoted $R = \langle C_1, \ldots, C_n \rangle$. We say that R derives the Horn clause D in the case that D is the final resolvent of R.

SLD derivation is exemplified below.

Example 4.2. Example of SLD derivation. Figure 4.1 shows the SLD derivation $R = \langle C_1, C_2, C_3 \rangle$. Clauses C_1 and C_2 are first resolved on their leftmost respective literals to give the resolvent R_{12}. Similarly in the second resolution C_3 is resolved with R_{12} to give the final resolvent R_{123}. Thus R derives R_{123}.

We now define SLD refutation as a special case of SLD derivation.

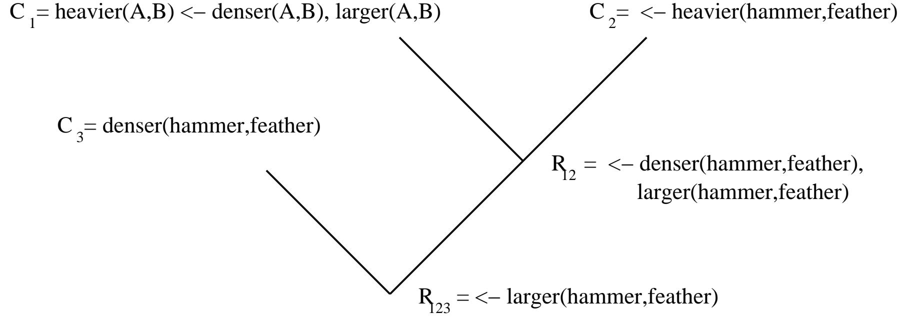

Figure 4.1: Example of SLD derivation

Definition 4.3. SLD refutation. An SLD refutation is an SLD derivation which derives the empty clause, $\square$.

4.1.1 Mode-Directed Inverse Entailment

Mode-Direct Inverse Entailment (MDIE) was introduced in [Mug95a] as the basis for the ILP system Progol. The input to an MDIE system is the vector $S_{MDIE} = \langle M, B, E \rangle$ where M is a set of mode statements M, B is a logic program representing the background knowledge and E is is set of examples. M can be viewed as a set of metalogical statements used to define the hypothesis language $\mathcal{L}_M$. The aim of the system is to find a set of consistent hypothesized clauses H such that for each clause $h \in H$ there is at least one positive example $e \in E$ such that the following holds.

$$B, h \models e$$

For any B, h, e this is equivalent to the following.

$$B, \neg e \models \neg h$$

This form allows hypotheses to be derived from B and e using standard Prolog theorem proving techniques. Since $\neg h$ takes the form of a ground conjunction of literals, for any finitely bound hypothesis language $\mathcal{L}_M$ there is a maximal ground conjunction $\bot_e$ for which the following holds.

$$B, \neg e \models \neg \bot_e \models \neg h$$

Having selected an example e and constructed $\bot_e$ Progol conducts a refinement graph search which considers hypotheses h in the interval

$$\square \succ h \succeq \bot_e$$

where "$\succeq$" denotes θ-subsumption.

4.1.2 Top-Directed Hypothesis Derivation

The input to an TDHD system is the vector $S_{TDHD} = \langle NT, \top, B, E \rangle$ where NT is a set of "non-terminal" predicate symbols (prefixed with a $ sign), $\top$ is a logic program representing the declarative bias over the hypothesis space, B is a logic programming representing the background knowledge and E is a set of examples. Each clause in $\top$ must contain at least one occurrence of an element of NT while clauses in B and E must not contain any occurrences of elements of NT. The aim of a TDHD system is to find a set of consistent hypothesized clauses H, containing no occurrence of NT, such that for each clause $h \in H$ there is at least one positive example $e \in E$ such that the following hold.

$$\top \models h \tag{4.1}$$

$$B, h \models e \tag{4.2}$$

Given the assumptions above we can now show the following lemma.

Lemma 4.4. Example derivability. Given $S_{TDHD} = \langle NT, \top, B, E \rangle$ assumptions (4.1) and (4.2) hold only if for each $e \in E$ there exists an SLD refutation R of $\neg e$ from $\top, B$.

Proof. *Assuming false there must exist e for which there does not exist an SLD refutation of* $\neg e$ *from* $\top, B$. *However, given* S_{TDHD} *we can show from (4.1) that for each* h

$$\top, B \models B, h. \tag{4.3}$$

From (4.3) and (4.2) it follows that for each $e \in E$

$$\top, B \models e. \tag{4.4}$$

From (4.4) and the completeness of SLD derivation [Llo87] it follows that there exists an SLD refutation R *of* $\neg e$ *from* $\top, B$. *This contradicts the assumption and completes the proof.*

According to the following theorem, the results of the preceding lemma can be used to extract implicit hypotheses from the refutations of an example $e \in E$.

Theorem 4.5. Given $S_{TDHD} = \langle NT, \top, B, E \rangle$ assumptions (1) and (2) hold only if for each positive example $e \in E$ there exists an SLD refutation R of $\neg e$ from $\top, B$, such that R can be re-ordered to give $R' = D_h R_e$ where D_h is an SLD derivation of a hypothesis h for which (1) and (2) hold.

According to Theorem 4.5, implicit hypotheses can be extracted from the refutations of a positive example $e \in E$. Let us now consider a simple example.

Example 4.6. Let $S_{TDHD} = \langle NT, \top, B, E \rangle$ where NT, B , e and $\top$ are as follows:

$$NT = \{\$body\} \quad B = b_1 = \text{pet(lassy)} \leftarrow \quad e = \text{nice(lassy)} \leftarrow$$

$$\top = \begin{cases} \top_1 : \text{nice}(X) \leftarrow \$\text{body}(X) \\ \top_2 : \$\text{body}(X) \leftarrow \text{pet}(X) \\ \top_3 : \$\text{body}(X) \leftarrow \text{friend}(X) \end{cases}$$

Given the linear refutation $R = \langle \neg e, \top_1, \top_2, b_1 \rangle$, we now construct the re-ordered refutation $R' = D_h R_e$ where $D_h = \langle \top_1, \top_2 \rangle$ derives the clause $h = \text{nice(X)} \leftarrow \text{pet(X)}$ for which (1) and (2) hold.

4.2 TopLog modules

This section describes the main TopLog modules and how they relate to implement the TopLog system.

4.2.1 From mode declarations to $\top$ theory

As the user of TopLog may not be familiar with specifying a search bias in the form of a logic program, TopLog has a module to build a general $\top$ theory automatically from user specified mode declarations. In this way input compatibility is ensured with existing ILP systems. Figure 4.2 is a simplified example of user specified mode declarations and the automatically constructed $\top$ theory.

modeh(mammal(+animal)).
modeb(has_milk(+animal)).
modeb(has_eggs(+animal)).

$$\top = \begin{cases} \top_1 : \text{mammal}(X) \leftarrow \$\text{body}(X). \\ \top_2 : \$\text{body}(X) \leftarrow .\%\mathit{emptybody} \\ \top_3 : \$\text{body}(X) \leftarrow \text{has_milk}(X), \$\text{body}(X). \\ \top_4 : \$\text{body}(X) \leftarrow \text{has_eggs}(X), \$\text{body}(X). \end{cases}$$

Figure 4.2: Mode declarations and a $\top$ theory automatically constructed from it

The above illustrated $\top$ theory is extremely simplified. The actual implementation has stricter control rules like: variables may only bind with others of the same type, a newly added literal must have its input variables already bound.

It is worth pointing out that the user could directly write a $\top$ theory specific for the problem, potentially restricting the search better than the generic $\top$ theory built automatically from the mode declarations.

4.2.2 Hypothesis derivation

In TopLog hypotheses are generated in a novel way. There is no construction of the bottom clause but rather an example guided generalization, deriving all hypotheses that entail the example with respect to the background knowledge.

The hypothesis derivation procedure is composed of two distinct steps. In the first step an example is proved from the background knowledge and the $\top$ theory. That is, the $\top$ theory is executed having the example as its start clause. This execution yields a proof consisting of a sequence of clauses from the $\top$ theory and background knowledge.

For instance, using the $\top$ theory from figure 4.2 and $B = b_1 = \text{has_milk(dog)}$ to derive refutations for example $e = \text{mammal(dog)}$, the following two refutations would be yielded: $r_1 = \langle \neg e, \top_1, \top_2 \rangle$ and $r_2 = \langle \neg e, \top_1, \top_3, b_1, \top_2 \rangle$.

In the second step, Theorem 4.5 is applied to r_1 and r_2 deriving, respectively, the clauses $h_1 = mammal(X)$ from $\langle \top_1, \top_2 \rangle$ and $h_2 = mammal(X) \leftarrow has_milk(X)$ from $\langle \top_1, \top_3, \top_2 \rangle$.

4.2.3 Learning algorithm

The table below shows the pseudo code for the TopLog learning algorithm. Notice that the hypothesis derivation procedure described in 4.2.2 corresponds to step 2.1 below.

1.	Let H be an empty set of pairs (hypothesis, examples that generated it)
2.	For each positive example e in E do
	2.1. Generate all hypotheses, H_e that are generalizations of e
	2.2 Merge H_e into H
3.	Compute the coverage of each hypothesis in H
4.	Build final theory, T, by choosing a subset of hypothesis in H

Table 4.1: Main TopLog algorithm

In step 1, H is a set of pairs: $\langle h, Eg_h \rangle$, where Eg_h is the set of (positive) examples that generated h. Step 2.2, adds more pairs to the set in case the hypothesis was not generated by any example before or updates existing pairs adding the current example as another generator of a previously found hypothesis. For efficiency we assume two hypotheses are equivalent if their derivations are identical.

The third step of the algorithm, computing the coverage of each hypothesis, would not be needed if the user program is a pure logic program (i.e. no cut symbol,

nor usage of Prolog built in operators) and no negative examples exist. This is because, by construction, the TDHD algorithm generates all hypotheses that entail a given example with respect to the user supplied mode declarations. This implies that the coverage of an hypothesis is exactly the set of examples that have it as their generalization.

However, this coverage computation step is needed for the negative examples, as they were not used to build the hypothesis set, and we also apply it to the positive examples because of two practical reasons. First, it is often the case that the user program is not a pure logic program (e.g. uses the cut symbol or built in comparison operators like $<$ and $>$). Second, it is not guaranteed that all hypotheses generalizing an example are generated from it in the case not enough resources (e.g. maximum proof depth, maximum hypotheses per example) are given to the program.

Notice that, contrary to the default setting in Aleph and Progol, no examples are retracted during the hypothesis construction stage (possibly changing the hypotheses that may be generated later) and thus the final theory is not influenced by the relative example order in the user program.

4.2.4 Building the final theory

The data available for building the final theory is a set of hypotheses, H, where each hypothesis has associated the set of examples from which it was derived, Eg_h, and the set of examples which it entails, Ec_h. As we have discussed above, for pure logic programs and when the given search resources are enough, these two sets would be equal but in general $Ec_h \neq Eg_h$.

The final theory, T, is a subset H' of H that maximizes a given score function. A score function is normally defined as a function of two metrics of T: 1) the set of examples T covers, E_{c_T}, and 2) total number of literals in T.

The set of examples a theory covers, E_{c_T}, is the union of the examples covered by its individual hypotheses. The total number of literals in a theory is the sum of all literals present in each of its hypotheses. The number of literals present in an hypothesis h is denoted by $|h|$.

The ultimate goal of a theory is to, while keeping good comprehensibility, achieve the highest accuracy on unseen data. In order to fulfill that goal, TopLog's default compression-based evaluation function for theories is:

$$\sum_{e \in E_{c_T}} weight(e) - \sum_{h \in T} |h| \tag{4.5}$$

The weight associated to an example, $weight(e)$, is user defined with positive ex-

amples having weight 1 and negative examples weight -1 by default. Weights greater than zero mean the example is positive, smaller than zero means it is negative.

The rationale behind this score function is to, assuming each example is worth one literal, measure by how many literals a theory compresses a dataset. Notice that this score function is similar to Progol's and Aleph's compression measure.

The problem of building a final theory, in our setting, is similar to the set covering problem, which is known to be NP-Complete and has plenty of literature on other related theoretical results [THCS01].

As there is no known efficient algorithm to generate the optimal T, an approximation will have to suffice. Natural choices for generating T are optimization approaches like simulated annealing or genetic algorithms, however we opted for a much simpler greedy algorithm which has several advantages. It is deterministic, has no parameters to tune, is efficient, simple to implement in Prolog and, for the similar case of the set covering problem, there are theoretical results proving that a greedy algorithm is the best-possible polynomial time approximation [LY94].

1. Let H be the unique set of hypotheses generated from the examples
2. Let T, the final theory, be an empty set of hypotheses
3. While there exists an H_i in H that may increase T's score
   3.1. Pick the H_i that, when added to T, most increases T's score
   3.2. Remove H_i from H
4 Output all the hypotheses on T as the final theory

Table 4.2: Greedy final theory generation algorithm

This greedy approach is computationally efficient and returns reasonably good solutions.

4.2.5 Efficient cross-validation

When the user provides examples they may also indicate which fold each example belongs to or let TopLog randomly and uniformly assign examples to folds. TopLog builds all hypotheses that are generated from each example in a first stage. In a second stage the final theory is chosen, allowing a efficient cross-validation implementation.

In the first stage all hypotheses are generated for each example regardless of the fold it belongs to, thus allowing TopLog to have associated with each hypothesis two sets of examples: 1) the examples that generated it, 2) the examples it covers (a

superset of the first). Only the theory generation algorithm, detailed in Table 4.2, needs to take folds into account.

In step 1 the hypotheses considered are only the ones generated by at least one example from a training fold (i.e. hypotheses generated exclusively from examples from the current test fold are removed). In step 3, the computation of T's score by adding a new hypothesis to it now also has to take folds into account. For the purpose of scoring the merit of adding a H_i to T, any positive or negative examples H_i covers are ignored, thus we only consider H_i's coverage on the training folds.

The generation and coverage computation dominates execution times. This is done only once, which minimizes the cost of the greedy algorithm and its application in cross-validation. Notice that this efficient cross-validation, which is internal to TopLog is in sharp contrast with the one performed in most other ILP systems (e.g. Aleph, Progol) where the time consuming hypothesis generation step needs to take place for each fold of the dataset. These ILP systems typically do not support directly cross-validation but rather rely on external scripts to separate the data into folds and run the system fold times in each of them.

4.3 Comparison with other ILP systems

There are many existing ILP systems. Two of the widely used systems are Progol [Mug95a] and Aleph [Sri07]. Aleph tries to integrate ideas from many systems and, with the proper settings, can emulate Progol. Aleph's main algorithm is: 1) Select an example to be generalized, 2) Build most-specific-clause, $\perp$, for the example, 3) Heuristic search to find a clause more general than $\perp$ but bounded by it, 4) Remove examples covered by the clause found in step 3, 5). If more positive examples exist go to step 1 otherwise stop.

Some of TopLog's advantages are: irrelevance of example order, no need for a bottom clause, global optimization of the theory after all hypothesis are available (possible leading to higher accuracy), efficient built-in cross-validation. The main disadvantage is that the hypothesis derivation procedure is not heuristically guided which may lead to many non interesting hypothesis being generated leading to decreased efficiency.

Requiring a bottom clause, $\perp$, constrains the problems an ILP may solve because the size of $\perp$ grows exponentially with respect to the layers of new variables needed to be added (the i setting in Progol and Aleph). For most problems this may not be a major issue but some it is.

Below we present a problem which highlights the advantage of not requiring a

bottom clause. This problem is also a good example of the broad problems ILP systems can solve. Notice that a feature based machine learning algorithm could not solve it. That is partly because the lack of expressiveness but also because it has a recursive solution.

4.3.1 Learning the Fibonacci series

The well known Fibonacci series is *0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, ...*. Each element in the series, apart from $F_0 = 0$ and $F_1 = 1$, is given by the sum of the previous two elements. That is $F_n = F_{n-1} + F_{n-2}$ (e.g. $F_5 = F_4 + F_3, 5 = 3 + 2$).

The problem for the ILP system to solve is, given a minimal background knowledge and a small set of positive examples (i.e. pairs $\langle n, F_n \rangle$), induce the rule that, given an index n, yields $F_(n)$ (e.g. $F_9 = 34$).

The figure below shows the background knowledge provided to TopLog in order to learn the Fibonacci series.

```
pred(N, M):- M is N-1.
plus(A, B, C):- C is A + B.
fib(0, 0).
fib(1, 1).
```

Figure 4.3: Background knowledge for learning Fibonacci series

The background knowledge only provides the predecessor, addition and the Fibonacci sequence values for the first two indexes. This is all that is needed and the definition is self explanatory. The figure below shows the mode declarations.

```
:-modeh(fib(+int/N,-int),[N>1]).
:-modeb(2, pred(+int,-int)).
:-modeb(2, fib(+int,-int)).
:-modeb(plus(+int,+int,-int), commutative).
:-set(maximum_literals_in_hypothesis, 6).
```

Figure 4.4: Mode declarations and settings

The mode header declaration states that the predicate to be induced is *fib/2*. It has an input argument N of type integer and an output argument of type integer as well. In square brackets is an optional parameter specifying the pre-conditions that the input arguments must verify in the induced rules. In this case it is being greater than 1.

The mode body declarations specify which literals may appear in the body of a valid hypothesis and their argument types and IO modes. The optional integer argument in the mode body declarations specifies up to how many times the literal may appear in the body of an hypothesis (by default it is one). The optional "commutative" argument after the predicate tells TopLog that the input order for this predicate is commutative (i.e. the order of the input variables does not affect the output value).

The *maximum_literals_in_hypothesis* is the TopLog equivalent setting to Aleph's *clauselength*. It specifies how many literals are allowed in the body of a hypothesis (by default it is 3, the same as Aleph). The value had to be increased to 6 to be able to learn Fibonacci.

```
example(fib(5,5), 10).
example(fib(7,13), 10).
```

Figure 4.5: Fibonacci Examples

Figure 4.5 has all examples used. The "10" after each example is its weight. Notice that only two examples were needed and there are no negative examples. TopLog is capable of doing positive only learning when the target predicate has both input and output variables. Finally, after executing TopLog with the background knowledge, settings and examples it induces the following model:

```
fib(A,B) :- A>1, pred(A,C), pred(C,D), fib(D,E), fib(C,F), plus(F,E,B).
```

Figure 4.6: TopLog induced model for the Fibonacci series

Knowing Prolog it is relatively straightforward to interpret the induced rule. It states, in Prolog, how to get the Fibonacci value, B for an arbitrary index A greater than 1. The procedure is: 1) let C be the predecessor of A (i.e. $C = A - 1$), 2) let D be the predecessor of C (i.e. $D = C - 1 = A - 2$), 3) let E be the Fibonacci value for index D (i.e. $E = F_{A-2}$), 4) let F be the Fibonacci value for index C (i.e. $E = F_{A-1}$), 4) finally, let B, the head output variable be the sum of F with E (i.e. $B = F + E$).

In a more convenient notation the rule just says: $\forall_{n>1} F_n = F_{n-1} + F_{n-2}$.

Chapter 5

Empirical evaluation of TopLog

In this chapter we describe the ILP systems and datasets used to empirically evaluate TopLog.

5.1 Methods

We chose to compare TopLog against Aleph for several reasons. Aleph [Sri07], as TopLog, is implemented in YAP Prolog and is a mode directed inverse entailment ILP system as Progol[Mug95a] [1].

The experiments were performed on a Intel Core 2 Duo @ 2.13 GHz with 2Gb of RAM using Ubuntu Linux with kernel version 2.60.3. YAP version 5.1.3 was used for both ILP systems.

The task of each ILP system is identical in all datasets: given positive and negative examples of the concept to be learned, generate a compact theory compatible with the examples provided. The quality of the induced theories are assessed by measuring their ten-fold cross validated mean accuracy and standard deviation.

Aleph and TopLog were executed with similar settings to ensure a fair test. Clause length=4 (in DSSTox=10), noise=100%, evaluation function=compression and search nodes per example=1000. Aleph was called both with *induce* and *induce_max* settings. In *induce* (the default), after finding a compressive clause for an example, it retracts all positive examples covered by that clause while *induce_max*, as TopLog, does not. We should compare the times and accuracies between Aleph with *induce_max* and TopLog. Both ensure the induced theory does not depend on the example ordering.

[1] Aleph is considerably faster than Progol which is implemented in C. This may seem odd but is partly due to the merit of the underneath Prolog interpreter. The built-in Prolog interpreter in Progol is rather inefficient compared to YAP.

5.2 Datasets

We conducted experiments on five datasets, all binary classification tasks. See Table 5.1 for an overview of the datasets.

Mutagenesis [SKMS97] is a well known domain for structure-activity relation prediction. The problem is to classify compounds as mutagenic (or not) given their chemical structure describe in terms of atoms, bonds, atom charge and information about atom and bond types. No numeric features (e.g. logp, lumo) are used. The dataset is divided into two sets: a regression friendly (r.f.) set with 188 examples (125 positives, 63 negatives) and a regression unfriendly (r.u.) set with 42 examples (13 positives, 29 negatives).

Carcinogenesis [SMKS96] is also a domain for structure-activity relation prediction. The problem is to classify compounds as carcinogenic (or not) given compound structural elements (e.g. hetero_ar_6_ring, methyl). One of the features used to discriminate for carcinogenicity is mutagenicity. Induced rules found in the previous dataset were used as background knowledge in this dataset. The dataset has 162 positives and 136 negative examples.

For **Alzheimers** [KSS95] the aim is to compare 37 analogues of Tacrine, a drug against Alzheimer's disease, according to four desirable properties: inhibit **amine** re-uptake, low **toxicity**, high **acetyl** cholinesterase inhibition, and good **reversal** of scopolamine-induced memory deficiency. For any of these four properties, examples consist of pairs of *pos(X,Y)* (resp. neg(X,Y) of two analogues indicating that X is better (resp. worse) than Y with respect to the property. The relation is transitive and antisymmetric but not complete as for some pairs of compounds the comparison result could not be determined.

DSSTox [RW00] was made available from the National Health and Environmental Effects Research Laboratory, USA. The dataset represents one of the most diverse set of toxins presently available in the public domain. It contains organic and organometallic molecules with their toxicity values. The problem is to classify molecules as toxic (or not). Each molecule is solely described by its chemical structure (i.e. atoms and bonds). The dataset consists of 576 molecules (220 positives and 356 negatives).

Details on the datasets used are summarized in Table 5.1.

The meaning of the columns is: number of positive examples, number of negative examples, total number of examples, default accuracy, number of body mode declarations and size of the background knowledge measured in number of clauses. The default accuracy is the accuracy yielded by the simple model that classifies an example as belonging to the most common class, its value is thus $max(\#E^+, \#E^-)/\#E$

Dataset	#E$^+$	#E$^-$	# E	Def. Acc.	#Modeb	#Background
Mutagenesis	125	63	188	66.5%	3	14,379
Carcinogenesis	162	136	298	54.4%	40	24,672
Alzheimers	343	343	686	50.0%	32	628
DSSTox	220	356	576	61.8%	2	27,793

Table 5.1: Dataset statistics

and should be considered the baseline that any non-trivial classifier should beat.

5.3 Results

In the table below, time is the CPU seconds the ILP systems took to build a model in the training data and for ten folds (CV column). We distinguish between the two to highlight the benefits of TopLog's efficient cross validation. The accuracy column has the average (over the ten folds) percentage of correct predictions made by the ILP models with the respective standard deviation.

	Aleph with induce			Aleph with induce_max			TopLog		
		Times			Times			Times	
Dataset	CV Accuracy	Train	CV	CV Accuracy	Train	CV	CV Accuracy	Train	CV
Mutagenesis	77.2%±9.2%	0.4s	4s	68.6%±11.4%	2s	17s	70.2%±11.9%	0.4s	0.5s
Carcinogenesis	60.9%±8.2%	6s	54s	65.1%±8.6%	29s	245s	64.8%±6.9%	7.0s	7.4s
Alzheimers	67.2%±5.0%	5s	40s	72.6%±6.2%	18s	156s	70.4%±5.6%	17s	16s
DSSTox	70.5%±6.5%	30s	253s	71.3%±3.4%	82s	684s	71.7%±5.6%	3.4s	3.6s

Table 5.2: Accuracy and time comparison between Aleph and TopLog

In the *induce_max* setting TopLog is clearly faster than Aleph. In the *induce* setting the speed advantage for training is dataset dependent but considering only CV then TopLog is again clearly faster. Although this may seem a side point, built-in efficient CV is important both to tune parameters and to properly assess model accuracy. The accuracies are identical with none being statistically significantly different at $\rho = 0.01$ level.

5.4 Conclusions

These results are very preliminary and not conclusive. Apparently TopLog is not less accurate than Aleph, is faster in many settings. Moreover, it has the potential to be yet faster when coupled with hypotheses sampling (see Future work discussion below).

To draw stronger conclusions, more datasets have to be used and, more importantly, a proper theorethical characterization of the hypotheses search space.

Chapter 6

PrioLog: A stochastic TopLog

TopLog showed a novel mechanism to derive hypotheses. However it still has several caveats, the main one is the need to perform a full coverage on all examples when testing a new hypothesis.

In PrioLog we will instead guided the coverage tests in order to gain the maximum information while performing only a small fraction of TopLog's tests.

6.1 PrioLog main algorithm

1. Let E=Set of all examples
2. Let Es={} %Set of seen examples
3. Let T=Table where columns are hypotheses and rows are examples
4. Let e=current example to process
5. E-stage: if ShallPickSeenExample(Es, E) then
   - 5.1. e=get_old_example(Es)
   - 5.2. else
   - 5.3. e=get_new_example(E)
6. E=E U e
7. H-stage: if ShallPickOldHypothesis(H, e) then
   - 7.1. h=get_old_hypothesis(H)
   - 7.2. else
   - 7.3. h=get_new_hypothesis(H, Top, e)
8. UpdateCoverage(h,e)

Table 6.1: PrioLog main algorithm

At each st The key idea

Chapter 7

Conclusions and future work

In the previous chapters we showed the progress so far and are confident it is encouraging for this stage of the Ph.D. A short paper [SHMTN08] describing Top Directed Hypothesis Derivation and TopLog was accepted to the 24th International Conference in Logic Programming (ICLP 2008). This work was also accepted as a poster in the 18th International Conference in Inductive Logic Programming (ILP08).

Below we summarize and briefly analyze the work plan for the next two years.

In the first 11 months of the Ph.D. we have:

1. Months 1-3: Got acquainted to previous work and implement a simple ILP system (Spectre)
2. Months 4-7: Matured the TDHD framework and developed a prototype system implementing it (TopLog)
3. Months 7-8: Submitted a paper about this system to a top conference in the area: ILP 2008
4. Months 9-10: Submitted a paper about this system to a top conference in the area: ILP 2008
5. Month 11: Revise submitted papers and update transfer report

Our plans for the next 25 months are:

1. Month 12: Prepare and present work in ILP08 conference.
2. Months 13-15: Revise TopLog in order to simplify the $\top$ theory. Characterize search space.
3. Months 16-17: Update the $\top$ theory to a Stochastic Logic Program
4. Months 18-20: Deal with new challenges from the SLP representation
5. Months 21-22: Submit paper with the recent TopLog improvements

6. Month 23: Parallelize TopLog to take advantage of multiple cores in same machine
7. Months 24-27: Do a relevant application showing the full advantage of the novel TopLog features
8. Months 29-30: Submit paper demonstrating these advantages
9. Months 31-36: Write and defend the thesis

We plan to write, at least, two more substantial papers until the end of the Ph.D. One about further developments in TopLog, specially the upgrading of the $\top$ theory to an Stochastic Logic Program [Mug95b]. The second about an application, possibly in the biological domain, showing how the latest TopLog developments are crucial for the chosen problem.

Below we further detail some of these topics.

7.1 Upgrade $\top$ theory to an SLP

If the $\top$ theory represents a Stochastic Logic Program rather than a regular logic program as it is now, it is possible to elegantly bias the hypotheses search space. The most immediate advantage is sampling the hypotheses space. If during the learning we update the SLP probabilities to further bias towards the most compressive hypotheses we may have additional speed gains. However, this subject should be properly investigated as there is also the danger of finding only locally optimal hypothesis.

7.2 Parallelization

In TopLog building the hypotheses set is example independent. This allows for a straightforward parallelization of TopLog main algorithm by dividing the examples through all available cpus. The main complication that may arise from here are implementation-wise.

7.3 Relevant applications

In order to demonstrate the benefits of TopLog's new ideas, specially the sampling of the hypotheses space using an SLP as the $\top$ theory, it is important to find a

suitable application. It is not clear yet what will this application be, but is very likely to come from the biological domain.

The choice of the biological domain is natural because often extensive relational knowledge is required to express the problem. It is possible that some of the larger datasets used in section 5.2 are good candidates to empirically validate our claim. However, we will try to find new problems.

An initial discussion with Prof. Mike Sternberg and Dr. Lawrence Kelly showed possible interest from their side in using TopLog in a Support Vector Inductive Logic Programming framework [MLAS05]. It is relatively straightforward to do so.

Bibliography

[BIA94] H. Boström and P. Idestam-Almquist. Specialisation of logic programs by pruning SLD-trees. In S. Wrobel, editor, *Proceedings of the Fourth Inductive Logic Programming Workshop (ILP94)*, pages 31–48, Bonn, 1994. GDM-studien Nr. 237.

[BM94] Michael Bain and Stephen Muggleton. Learning optimal chess strategies. In *Machine Intelligence 13*, pages 291–309, 1994.

[Bon70] Mikhail Moiseevich Bongard. *Pattern Recognition*. Spartan Books, 1970.

[CL01] Chih-Chung Chang and Chih-Jen Lin. *LIBSVM: a library for support vector machines*, 2001.

[Coh94] William W. Cohen. Grammatically biased learning: Learning logic programs using an explicit antecedent description language. *Artif. Intell.*, 68(2):303–366, 1994.

[Cos08] Vítor Santos Costa. *YAP*. OPorto University, 2008.

[CSL07] Vítor Santos Costa, Konstantinos F. Sagonas, and Ricardo Lopes. Demand-driven indexing of prolog clauses. In Dahl and Niemelä [DN07], pages 395–409.

[DN07] Verónica Dahl and Ilkka Niemelä, editors. *Logic Programming, 23rd International Conference, ICLP 2007, Porto, Portugal, September 8-13, 2007, Proceedings*, volume 4670 of *Lecture Notes in Computer Science*. Springer, 2007.

[dSC07] Anderson Faustino da Silva and Vítor Santos Costa. Design, implementation, and evaluation of a dynamic compilation framework for the yap system. In Dahl and Niemelä [DN07], pages 410–424.

[FMPS98] P. Finn, S. Muggleton, D. Page, and A. Srinivasan. Pharmacophore discovery using the inductive logic programming system progol. *Machine Learning*, 30:241–271, 1998.

[KCM87] S.T. Kedar-Cabelli and L.T. McCarty. Explanation-based generalization as resolution theorem proving. In P. Langley, editor, *Proceedings of the Fourth International Workshop on Machine Learning*, pages 383–389, Los Altos, 1987. Morgan Kaufmann.

[KMSS96] R. King, S. Muggleton, A. Srinivasan, and M. Sternberg. Structure-activity relationships derived by machine learning: the use of atoms and their bond connectives to predict mutagenicity by inductive logic programming. In *Proceedings of the National Academy of Sciences*, volume 93, pages 438–442, 1996.

[Kow74] R. Kowalski. Predicate logic as programming language. In *Proceedings of IFIP Congress 74*, pages 569–574. North Holland Publishing Co., Amsterdam, 1974.

[KSS95] R.D. King, A. Srinivasan, and M.J.E. Sternberg. Relating chemical activity to structure: an examination of ILP successes. *New Generation Computing*, 13:411–433, 1995.

[Llo87] J.W. Lloyd. *Foundations of Logic Programming*. Springer-Verlag, Berlin, 1987. Second edition.

[LY94] Carsten Lund and Mihalis Yannakakis. On the hardness of approximating minimization problems. *J. ACM*, 41(5):960–981, 1994.

[Mit97] Tom M. Mitchell. *Machine Learning*. McGraw-Hill, 1997.

[MLAS05] Stephen Muggleton, Huma Lodhi, Ata Amini, and Michael J. E. Sternberg. Support vector inductive logic programming. In Achim G. Hoffmann, Hiroshi Motoda, and Tobias Scheffer, editors, *Discovery Science*, volume 3735 of *Lecture Notes in Computer Science*, pages 163–175. Springer, 2005.

[Mug95a] S. Muggleton. Inverse entailment and Progol. *New Generation Computing, Special issue on Inductive Logic Programming*, 13(3-4):245–286, 1995.

[Mug95b] S. Muggleton. Stochastic logic programs. In L. De Raedt, editor, *Proceedings of the 5th International Workshop on Inductive Logic Programming*. Department of Computer Science, Katholieke Universiteit Leuven, 1995.

[NCdW97] S-H. Nienhuys-Cheng and R. de Wolf. *Foundations of Inductive Logic Programming*. Springer-Verlag, Berlin, 1997. LNAI 1228.

[Qui93] J. Ross Quinlan. *C4.5: Programs for Machine Learning*. Morgan Kaufmann, 1993.

[Rob65] J. A. Robinson. A machine oriented logic based on the resolution principle. *Journal of the ACM*, 12(1):23–41, 1965.

[RW00] A.M. Richard and C.R. Williams. Distributed structure-searchable toxicity (DSSTox) public database network: A proposal. *Mutation Research*, 499:27–52, 2000.

[SHMTN08] Jose C. A. Santos Stephen H. Muggleton and Alireza Tamaddoni-Nezhad. Toplog: Ilp using a logic program declarative bias. In *Proceedings of the 24th International Conference on Logic Programming (ICLP08)*, volume TBA of *Lecture Notes in Computer Science*, page TBA. Springer, 2008.

[SKMS97] A. Srinivasan, R. D. King, S.H. Muggleton, and M. Sternberg. Carcinogenesis predictions using ilp. In N. Lavrac and S. Dzeroski, editors, *Proceedings of the Seventh International Workshop on Inductive Logic Programming*, volume 1297 of *LNAI*, pages 273–287. Springer-Verlag, Berlin, 1997.

[SMKS96] A. Srinivasan, S. Muggleton, R. King, and M. Sternberg. Theories for mutagenicity: a study of first-order and feature based induction. *Artificial Intelligence*, 85(1,2):277–299, 1996.

[Sri07] Ashwin Srinivasan. *The Aleph Manual.* University of Oxford, 2007.

[THCS01] Ronald L. Rivest Thomas H. Cormen, Charles E. Leiserson and Clifford Stein. *Introduction to Algorithms, Second Edition.* MIT Press and McGraw-Hill, 2001.

[TMS98] M. Turcotte, S.H. Muggleton, and M.J.E. Sternberg. Protein fold recognition. In C. D. Page, editor, *Proceedings of the Eighth International Workshop on Inductive Logic Programming*, volume 1446 of *LNAI*, pages 53–64. Springer-Verlag, Berlin, 1998.

[Wik] Wikipedia. Machine learning. http://en.wikipedia.org/wiki/Machine_learning.