Induction as a Search Procedure

Stasinos Konstantopoulos
Institute of Informatics & Telecommunications, NCSR 'Demokritos'
P.O. BOX 60228, Ag. Paraskevi 15310, Greece
Tel: +30 21 06503162, Fax: +30 21 06532175
Email: konstant@iit.demokritos.gr

Rui Camacho
Faculdade de Engenharia da Universidade do Porto
R. Dr Roberto Frias, s/m, 4200-465 Porto, Portugal
Tel: +351 22 508 1849, Fax: +351 22 508 1443
Email: rcamacho@fe.up.pt
and
Laboratório de Inteligência Artificial e Ciências de Computadores,
Rua de Ceuta, 118, 6^o, 4150-190 Porto, Portugal
Tel: +351 22 339 2090, Fax: +351 22 339 2099

Nuno A. Fonseca
Laboratório de Inteligência Artificial e Ciências de Computadores,
University of Porto
R. Campo Alegre 823, 4150 Porto, Portugal
Tel: +351 22 607 8830, Fax: +351 22 600 3654
Email: nf@ncc.up.pt

Vítor Santos Costa
COPPE/Sistemas, Universidade Federal do Rio de Janeiro, Brasil
Centro de Tecnologia, Bloco H-319, Cx. Postal 68511, Rio de Janeiro, Brasil
(CEP: 21945-970)
Tel: +55 21 2562 8648, Fax: +55 21 2562 8676
Email: vitor@cos.ufrj.br

Induction as a Search Procedure

Abstract

This chapter introduces Inductive Logic Programming (ILP) from the perspective of search algorithms in Computer Science. It first briefly considers the Version Spaces approach to induction, and then focuses on Inductive Logic Programming: from its formal definition and main techniques and strategies, to priors used to restrict the search space and optimized sequential, parallel, and stochastic algorithms. The authors hope that this presentation of the theory and applications of Inductive Logic Programming will help the reader understand the theoretical underpinnings of ILP, and also provide a helpful overview of the State-of-the-Art in the domain.

1 INTRODUCTION

Induction is a very important operation in the process of scientific discovery, which has been studied from different perspectives in different disciplines. Induction, or inductive logic, is the process of forming conclusions that reach beyond the data (facts and rules), i.e., beyond the current boundaries of knowledge. At the core of inductive thinking is the ‘inductive leap’, the stretch of imagination that draws a reasonable inference from the available information. Therefore, inductive conclusions are only probable, they may turn out to be false or improbable when given more data.

In this chapter we address the study of induction from an Artificial Intelligence (AI) perspective and, more specifically, from a Machine Learning perspective, which aims at automating the inductive inference process. As is often the case in AI, this translates to mapping the ‘inductive leap’ onto a search procedure. Search, therefore, becomes a central element of the automation of the inductive inference process.

We consider two different approaches to the problem of induction as a search procedure: Version Spaces is an informal approach, more in line with traditional Machine Learning approaches; by contrast, search-based algorithms for Inductive Logic Programming (ILP) rely on a formal definition of the search space. We compare the two approaches under several dimensions, namely, expressiveness of the hypothesis language underlying the space, completeness of the search, space traversal techniques, implemented systems and applications thereof.

Next, the chapter focuses on the issues related to a more principled approach to induction as a search. Given a formal definition and characterization of the search space, we describe the main techniques employed for its traversal: strategies and heuristics, priors used to restrict its size, optimized sequential search algorithms, as well as stochastic and parallel ones.

The theoretical issues presented and exposed are complemented by descriptions of and references to implemented and applied systems, as well as real-world application domains where ILP systems have been successful.

2 A PRAGMATIC APPROACH: VERSION SPACES

This section is structured into three parts. A first part presents a set of definitions and concepts that lay the foundations for the search procedure into which induction is mapped. In a second part we mention briefly alternative approaches that have been taken to induction as a search procedure and finally in a third part we present the *version spaces* as a general methodology to implement induction as a search procedure.

2.1 A Historical Road Map

Concept Leaning is a research area of Machine Learning that addresses the automation of the process of finding (inducing) a description of a concept (called the 'target concept' or hypothesis) given a set of instances of such concept. Concepts and instances have to be expressed in a *concept description language* or *hypothesis language* ($\mathcal{L}$). Given a set of instances of some concept it is usually a rather difficult problem to (automatically) induce the 'target concept'. The major difficulty is that there may be a lot, if not an infinite, number of plausible conjectures (hypotheses) that are 'consistent' with the given instances. Automating the induction process involves the generation of the candidate concept descriptions (hypotheses), their evaluation, and the choice of 'the best one' according to some criteria. The concept learning problem is often mapped into a search problem, that looks for a concept description that explains the given instances and lies within the concept description language.

An important step towards the automation of the learning process is the structuring of the elements of $\mathcal{L}$ in a manner that makes it possible to perform systematic searches, to justifiably discard some 'uninteresting' regions of candidate descriptions, and to have a compact description of the search space.

For this purpose, Machine Learning borrows from Mathematics the concept of a *lattice*, a partially ordered set with the property that all of its non-empty finite subsets have both a supremum (called join) and an infimum (called meet). A *semi-lattice* has either only a join or only a meet. The partial order can be any reflexive, anti-symmetric, and transitive binary relation. In Machine Learning a lattice is usually defined as follows:

Definition 1 *A lattice is a partially ordered set in which every pair of elements a, b has a greatest lower bound (glb, $a \sqcap b$) and least upper bound (lub, $a \sqcup b$).*

and the partial ordering is based of the concept of generality, which places clauses along general–specific axes. Such a partial ordering is called a *generalization ordering* or a *generalization model*, and has the advantage of imposing a structure that is convenient for systematically searching for general theories and for focusing on parts of the structure that are known to be interesting.

Different generalization models have been proposed in the Concept Learning literature; Mitchell (1997, p. 24), for example, defines generalization for a domain where instances (x) belong to a domain (X) of feature vectors and the target concept is encoded as a boolean-valued function (h):

Definition 2 *Let h_j and h_k be boolean-valued functions defined over X. Then h_j is* more general than or equal to *h_k (we write $h_j >_g h_k$) if and only if* $(\forall x \in X)[h_k(x) = 1 \rightarrow h_j(x) = 1]$.

Another popular definition of generality is based on the logical concept of semantic entailment:

Definition 3 *Given two clauses C_1 and C_2, we shall call C_1* more general (more specific) *than C_2, and write $C_1 >_g C_2$ ($C_1 <_g C_2$), if and only if C_1 entails C_2 (C_2 entails C_1).*

with various syntactic logical operators suggested for the purpose of inferring entailment. We shall revisit this point in the context of Inductive Logic Programming in the following section. This suggestion that induction may be automated by carrying a search through an order space was independently suggested by Popplestone (1970) and Plotkin (1970).

The search space of the alternative concept descriptions may be traversed using any of the traditional AI search strategies. In 1970 Winston (1970) describes a concept learning system using a depth-first search. In Winston's system one example at a time is analysed and a single concept description is maintained as the current best hypothesis to describe the target concept. The current best hypothesis is tested against a new example and modified to become consistent with the example while maintaining consistency with all previously seen examples. An alternative search strategy, breadth-first search, was used in concept learning by Plotkin (1970), Michalski (1973), Hayes-Roth (1974) and Vere (1975). These algorithms already take advantage of the order in the search space. Plotkin's work is revisited in Section 3.4.2 below, as it forms the basis for several developments in Inductive Logic Programming.

2.2 Introducing Version Spaces

We now focus our attention on a general approach to concept learning, proposed by Mitchell (1978), called *version spaces*. A *version space* is the set of all hypotheses consistent with a set of training examples. In most applications the number of candidate hypotheses consistent with the

Candidate-Elimination
Input: Examples (E) – positive (E^+) and negative (E^-).
Output: The sets G and S.

```
1.  G = set of maximally general hypotheses in H
2.  S = set of maximally specific hypotheses in H
3.  for all e $\in$ E
4.    if e $\in E^+$ then
5.      remove from G any hypothesis inconsistent with e
6.      for all s $\in$ S
7.        if s inconsistent with e then
8.          remove s from S
9.          add to S all minimal generalisations h of s such that
10.           h is consistent with e and $\exists_{g \in G}$ g $<_g$ h
11.         remove any s $\in$ S such that $\exists_{s' \in S}$ s $<_g$ s'
12.       endif
13.     end for all
14.   endif
15.   else (e $\in E^-$)
16.     remove from S any hypothesis inconsistent with e
17.     for all g $\in$ G
18.       if g inconsistent with e then
19.         remove g from G
20.         add to G all minimal specialisations h of g such that
21.           h is consistent with e and $\exists_{s \in S}$ h $<_g$ s
22.         remove any g $\in$ G such that $\exists_{g' \in G}$ g' $<_g$ g
23.       endif
24.     end for all
25. end for all
26. return S and G
```

Figure 1: The Candidate-Elimination algorithm

examples is very large or even infinite. It is therefore impractical—or even infeasible—to store an enumeration of such candidates. The version space uses a compact and elegant representation of the set of candidate hypotheses. The representation takes advantage of the order imposed over such hypotheses and stores only the most general and most specific set of descriptions that limit the version space.

In order to make a more formal presentation of the version spaces approach to concept learning let us introduce some useful definitions. Assume that the function $\mathrm{tg}(e)$ gives the 'correct' value of the target concept for each instance e. A hypothesis $\mathrm{h}(e)$ is said to be consistent with a set of instances (examples) E **iff** it produces the same value of $\mathrm{tg}(e)$ for each example $e \in E$. That is:

$$\mathrm{Consistent}(h, E) \equiv \forall_{e \in E}, \mathrm{h}(e) = \mathrm{tg}(e) \tag{1}$$

Examples are of two kinds: positive and negative. Positive examples are instances of the target concept whereas negative examples are not.

Let hypothesis space H be the set of all hypotheses that are part of hypothesis language $\mathcal{L}$. Version space $VS_{H,E}$ with respect to hypothesis space H and training set E, is the subset of H consistent with E.

$$VS_{H,E} \equiv \{h \in H | \mathrm{Consistent}(h, E)\} \tag{2}$$

Since the set represented by the version space may be very large, a compact and efficient way of describing such set is needed. To fully characterize the version space Mitchell proposes to represent the most general and more specific elements, i.e., a version space is represented by the upper and lower boundaries of the ordered search space. Mitchell proposes also the CANDIDATE-ELIMINATION algorithm to efficiently traverse the hypothesis space.

In the CANDIDATE-ELIMINATION algorithm the boundaries of the version space are the only elements stored. The upper boundary is called G, the set of the most general elements of the space, and the lower boundary S, the set of the most specific hypotheses of the version space.

The upper (more general) boundary G, with respect to hypothesis space H and examples E, is the set of the maximally general members of H consistent with E.

$$G \equiv \{g \in H | \mathrm{Consistent}(g, E) \wedge \neg\, (\exists_{g' \in H})(g' >_g g) \wedge \mathrm{Consistent}(g', E)\} \tag{3}$$

where $>_g$ denotes the 'more general than' relation. The lower (more specific) boundary S, with respect to hypothesis space H and training set E, is the set of maximally specific members of H consistent with E.

$$S \equiv \{s \in H | \mathrm{Consistent}(s, E) \wedge \neg\, (\exists_{s' \in H})(s >_g s') \wedge \mathrm{Consistent}(s', E)\} \tag{4}$$

The CANDIDATE-ELIMINATION algorithm is outlined in Figure 1. The algorithm proceeds incrementally, by analysing one example at a time. It starts with the most general set G containing all hypotheses from $\mathcal{L}$ consistent with the first example and with the most specific set S of hypotheses

training instances	version space	
	S set	G set
<meat, yes, yes, yes> (positive – an eagle)		
	{<meat, yes, yes, yes>}	{<_, _, _, _>}
<meat, no, yes, no> (negative – a wolf)		
	{<meat, yes, yes, yes>}	{<_, yes, _, _>, <_, _, _, yes>}
<seeds, yes, no, yes> (positive – a dove)		
	{<_, yes, _, yes>}	{<_, yes, _, _>, <_, _, _, yes>}
<insects, no, no, yes> (negative — a bet)		
	{<_, yes, _, yes>}	{<_, yes, _, _>}
<seeds, yes, no, no> (positive – an ostrich)		
	{<_, yes, _, _>}	{<_, yes, _, _>}

Figure 2: Simple example of the CANDIDATE-ELIMINATION algorithm sequence for learning the concept of bird.

from $\mathcal{L}$ consistent with the first example (the set with the first example only). As each new example is presented, G gets specialized and S generalized, thus reducing the version space they represent. Positive examples lead to the generalization of S, whenever S is not consistent with a positive example. Negative examples prevent over-generalization by specializing G, whenever G is inconsistent with the a negative example. When specializing elements of G, only specializations that are maximally general and generalizations of some element of S are admitted. Symmetrically, when generalizing elements of S, only generalizations that are maximally specific and specializations of some element of G are admitted. When all examples are processed, the hypotheses consistent with the presented data are the set of hypotheses from $\mathcal{L}$ within the G and S boundaries.

Some advantages of the version space approach with the CANDIDATE-ELIMINATION algorithm is that partial descriptions of the concepts may be used to classify new instances, and that each example is examined only once and there is no need for backtracking. The algorithm is complete, i.e., if there is a hypothesis in $\mathcal{L}$ consistent with the data it will be found.

Version spaces and the CANDIDATE-ELIMINATION algorithm have been applied to several problems such as learning regularities in chemical mass spectroscopy (Mitchell, 1978) and control rules for heuristic search (Mitchell et al., 1983). The algorithm was initially applied to induce rules for the DENDRAL knowledge-based system that suggested plausible chemical structure of molecules based on the analysis of information of the molecule chemical mass spectroscope data. Finally, Mitchell et al. (1983) uses the technique to acquire problem-solving heuristics for the LEX system in the domain of symbolic integration.

2.3 A Simple Example

The technique of version spaces is independent of the hypothesis representation language. According to the application, the hypothesis language used may be as simple as an attribute-value language or as powerful as First Order Logic. For illustrative purposes only (*proof-of-concept*) we present a very simple example using an attribute-value hypothesis language. We show an example of the CANDIDATE-ELIMINATION algorithm to induce a concept of a bird using five training examples of animals.

The hypothesis language will be the set of 4-tuples encoding the attributes *eats*, *has feathers*, *has claws* and *flies*. The domains of such attributes are: eats={meat, seeds, insects}, has feathers={yes, no}, has claws={yes, no} and flies={yes, no}. The representation of an animal that eats meat, has feathers, does not have claws and flies is the 4-tuple: <meat, yes, no, yes>. We use the symbol _ (underscore) to indicate that any value is acceptable. We say that a hypothesis h_1 matches an example if every feature value of the example is either equal for the corresponding value in h_1 or h_1 has the symbol "_" for that feature (meaning "any value" is acceptable). Let us consider that a hypothesis h_1 is more general than hypothesis h_2 if h_1 matches all the instances that h_2 matches but the reverse is not true. Figure 2 shows the sequence of values of the sets S and G after the analysis of five examples, one at a time. The first example is a positive example, an eagle (<meat, yes, yes, yes>), and is used to initialize the S set. The G set is initialized with the most general hypothesis of the language, the hypothesis that matches all the examples: <_, _, _, _>. The second example is negative and is a description of a wolf (<meat, no, yes, no>). The second example is not matched by the element of S and therefore S is unchanged. However the hypothesis in the G set covers the negative example and has to be minimally specialized. Minimal specializations of <_, _, _, _> that avoid covering the working example are: <seed, _, _, _>, <insects, _, _, _>, <_, yes, _, _>, <_, _, no, _>, <_, _, _, yes>. Among these five specializations only the third and the fifth are retained since they are the only ones that cover the element of S. Analysing the third example (a dove – a positive example) leads the algorithm to minimally generalize the element of S since it does not cover that positive example. The minimal generalization of S's hypothesis is <_, yes, _, yes> that covers all seen positive and is a specialization of at least one hypothesis in G. The forth example is negative and is a description of a bet (<insects, no, no, yes>). This examples forces a specialization of G. Finally the last example is a positive one (an ostrich – <seeds, yes, no, no>) that results in a minimal generalization of S. The final version space (delimited by the final S and G) defines bird as any animal that has a beak.

3 INDUCTIVE LOGIC PROGRAMMING THEORY

Inductive Logic Programming (ILP) is the Machine Learning discipline that deals with the induction of First-Order Predicate Logic programs. Related research appears as early as the late 1960's, but it is only in the early 1990's that ILP research starts making rapid advances and the term itself was coined.[1]

In this section we first present a formal setting for Inductive Logic Programming and highlight the mapping of this theoretical setting into a search procedure. The formal definitions of the elements of the search and the inter-dependencies between these elements are explained, while at same time the concrete systems that implement them are presented.

It should be noted at this point that we focus on the, so to speak, mainstream line of ILP research, that most directly relates to ILP's original inception and formulation. And that, even within this scope, it is not possible to describe in depth all ILP systems and variations, so some will only receive a passing reference. With respect to systems, in particular, we generally present in more detail systems were a concept or technique was originally introduced, and try to make reference to as many subsequent implementations as possible.

3.1 The Task of ILP

Inductive Logic Programming systems generalize from individual instances in the presence of background knowledge, conjecturing patterns about yet unseen data. Inductive Logic Programming, therefore, deals with learning concepts given some *background knowledge* and *examples* [2]. Both examples, background knowledge and the newly learnt concepts are represented as logic programs.

Given this common ground, the learning process in ILP can be approached in two different ways. In *descriptive* induction one aims at at describing regularities or patterns in the data. In *predictive* induction one aims at learning theories that solve classification/prediction tasks.

More precisely, let us assume there is some background knowledge B, some examples E, and a hypothesis language $\mathcal{L}$. We define *background knowledge* to be a set of axioms about the current task that are independent of specific examples. We define the set of *examples*, or *instances*, or *observations* as the data we want to generalize from. Often, but not always, examples are divided into *positive examples*, or E^+, and *negative examples*, or E^-, such that $E = E^+ \cup E^-$.

The task of *descriptive ILP* is finding a theory that explains the examples in the presence of the background. More formally, descriptive ILP is defined as follows:

Definition 4 *Given background knowledge B, examples E, and hypothesis language $\mathcal{L}$, if the following prior condition holds:*

(Prior Necessity) *B does not explain E*

then the task of descriptive ILP *is to find a maximally specific hypothesis $H \in \mathcal{L}$, such that:*

(Posterior Sufficiency) *H explains $B \wedge E$*

if such a hypothesis exists.

Definition 5 *Given background knowledge B, positive training data E^+, negative training data E^-, and hypothesis language $\mathcal{L}$,*
if the following prior conditions hold:

(Prior Satisfiability) *B does not explain E^-*

(Prior Necessity) *B does not explain E^+*

then the task of predictive ILP *is to find a hypothesis $H \in \mathcal{L}$ with the following properties:*

(Posterior Satisfiability) *$B \wedge H$ does not explain E^-*

(Posterior Sufficiency) *$B \wedge H$ explains E^+*

if such a hypothesis exists.

Informally, this definition states that first of all the negative data must be consistent with the background, since otherwise no consistent hypothesis can ever be constructed; and that a hypothesis must be necessary because the background is not a viable hypothesis on its own. Given that, predictive ILP augments existing (background) knowledge with a hypothesis, so that the combined result covers (predicts) the data.

3.1.1 Explanation Semantics

At the core of the ILP task definition is the concept of *explaining* observations by hypothesis, or, in other words, of hypotheses being *models* or *explanations*. There are two major *explanation semantics* for ILP, which substantiate this concept in different ways: *learning from interpretations* and *learning from entailment*.

Under learning from interpretations, a logic program covers a set of ground atoms which is a Herbrand interpretation of the program, if the set is a model of the program. That is, the interpretation is a valid grounding of the program's variables. Under learning from entailment, a program covers a set of ground atoms, if the program entails the ground atoms, that is, if the ground atoms are included in all models of the program. As De Raedt (1997) notes, learning from interpretations reduces to learning from entailment. This means that learning from entailment is a stronger explanation model, and solutions found by learning from interpretations can also be found by learning from entailment, but not vice versa.

From a practical point of view, in learning from interpretations each example is a separate Prolog program, consisting of multiple ground facts representing known properties of the example. A hypothesis covers the example when the latter is a model of the former, i.e., the example provides a valid grounding for the hypothesis. In learning from entailment, on the other hand, each example is a single fact. A hypothesis covers the example when the hypothesis entails the example.

A third explanation semantics that has been proposed is *learning from satisfiability* (De Raedt & Dehaspe, 1997), where examples and hypotheses are both logic programs, and a hypothesis covers an example if the conjunction of the hypothesis and the example is satisfiable. Learning from satisfiability is stronger than both learning from entailment and learning from interpretations.

3.1.2 ILP Settings

A *setting* of ILP is a complete specification of the task and the explanation semantics, optionally complemented with further restrictions on the hypothesis language and the examples representation.

Although in theory all combinations are possible, descriptive ILP is most suitable to learning from interpretations, since under this semantics, the data is organized in a way that is convenient for tackling the descriptive ILP task. Symmetrically, predictive ILP typically operates under learning from entailment semantics. In fact, this coupling is so strong, that almost all ILP systems operate in either the *non-monotonic* or the *normal setting*.

In the *non-monotonic setting* (De Raedt, 1997), descriptive ILP operates under learning from interpretations. The non-monotonic setting has been successfully applied to several problems, including learning association rules (Dehaspe & Toironen, 2000) and subgroup discovery (Wrobel, 1997). To perform subgroup discovery, the notion of explanation is relaxed to admit hypotheses that satisfy 'softer' acceptance criteria like, for example, similarity or associativity.

An interesting line of research is pursued by Blockeel & De Raedt (1998) who break the coupling we just described and propose TILDE, and ILP system that tackles the predictive ILP task under learning from interpretations. TILDE is used to induce *logical decision trees*, the first-order counter-part of decision trees.

A more surprising development is the application of ILP methodology to clustering, an unsupervised task where unlabeled examples are organized to clusters of examples with similar attributes. The underlying idea is based on Langley's (1996) view of decision trees as concept-hierarchies inducers where each node of the tree is associated with a concept. For this task, De Raedt & Blockeel (1997) build upon TILDE to propose C 0.5, a first-order clustering system that instead of class information to guide the search, borrows the concept of distance metrics from *instance based learning*.

In the *normal setting* (Muggleton, 1991), also called *explanatory ILP* or *strong ILP*, predictive ILP operates under learning from entailment semantics. Most ILP systems operate in this setting, or rather a specialization of the normal setting called the *example setting* of the *definite semantics* (Muggleton & De Raedt, 1994). This setting imposes the restrictions that the examples are ground instances of the target and the background and hypothesis are formulated as definite clause programs. As noted by Muggleton & De Raedt (ibid., Section 3.1, pp. 635–6), the restriction to definite semantics greatly simplifies the ILP task, since for every definite program T there is a model $\mathcal{M}^+(T)$ (its *minimal Herbrand model*) in which all formulae are decidable and the Closed World Assumption holds. In this example, definite setting, the task of normal ILP is defined as follows:

Definition 6 *Given background knowledge B, positive training data E^+, and negative training data E^-, a definite ILP algorithm operating within the example setting constructs a hypothesis H*

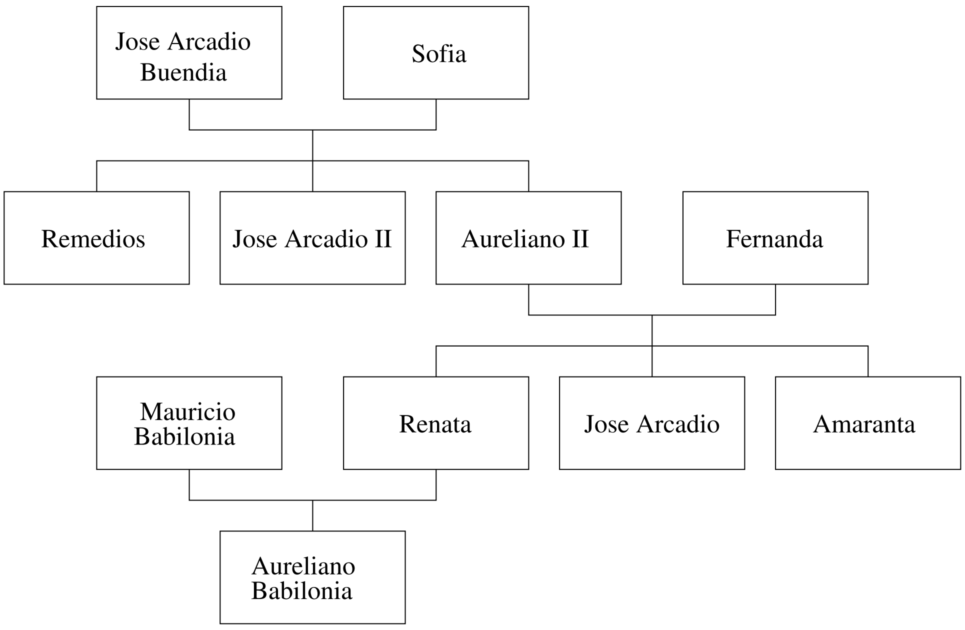

Figure 3: Family

with the following properties:

(Prior Satisfiability)	*All* $e \in E^-$ *are false in* $\mathcal{M}^+(B)$
(Prior Necessity)	*Some* $e \in E^+$ *are false in* $\mathcal{M}^+(B)$
(Posterior Satisfiability)	*All* $e \in E^-$ *are false in* $\mathcal{M}^+(B \wedge H)$
(Posterior Sufficiency)	*All* $e \in E^+$ *are true in* $\mathcal{M}^+(B \wedge H)$

3.1.3 A Simple Example

To make the difference between ILP settings more concrete, let us consider as an example the task of learning the grandfather relation based on the family shown in Figure 3.

In the definite setting, the background theory includes abstract definitions as well as ground facts. Abstract definitions provide potentially useful background predicates (like 'father' and 'sibling' in our example) that represent general prior knowledge about family relations. Ground facts (like parent(jArcadioBD,remedios) in our example) substantiate the basic predicates that are the 'primitives' from which more complex relations are built, and represent concrete knowledge about this particular family:

$$B = \left\{ \begin{array}{ll} \text{father}(X,Y) \leftarrow \text{parent}(X,Y) \wedge \text{male}(X) & \\ \text{sibling}(X,Y) \leftarrow \text{parent}(Z,X) \wedge \text{parent}(Z,Y) \wedge X \neq Y & \\ \quad \text{male}(\text{jArcadioBD}) & \text{female}(\text{remedios}) \\ \quad \text{male}(\text{aurelianoII}) & \text{female}(\text{renata}) \\ \text{parent}(\text{jArcadioBD}, \text{remedios}) & \\ \text{parent}(\text{sofia}, \text{remedios}) & \end{array} \right\} \tag{5}$$

We only show a fragment of the full background knowledge here, but the learning task needs that all relevant information in Figure 3 is included. Next, an ILP algorithm will receive the following examples:

$$\begin{aligned} E^+ &= \{\text{grandfather}(\text{jArcadioBD}, \text{jArcadio}), \text{grandfather}(\text{jArcadioBD}, \text{renata}), \\ &\quad\ \text{grandfather}(\text{aurelianoII}, \text{aureliano})\} \\ E^- &= \{\text{grandfather}(\text{sofia}, \text{jArcadio}), \text{grandfather}(\text{aurelianoII}, \text{renata})\} \end{aligned} \tag{6}$$

and construct a hypothesis regarding the intensional definition of the grandfather/2 predicate, so that it covers the positives without covering any of the negatives. Ideally, the learnt definition should *generalize* to unseen examples such as grandfather(jArcadioBD,amaranta).

In the non-monotonic setting, on the other hand, examples are organized as sets of ground terms, describing an object and the relations that it, or its sub-objects, take part in. Since we are interested in constructing the grandfather predicate, our objects are relationships between individuals, called *keys*:

$$E_1^+ = \left\{ \begin{array}{ll} \text{key}(\text{jArcadioBD}, \text{jArcadio}) & \\ \text{male}(\text{jArcadioBD}) & \text{male}(\text{jArcadio}) \\ \text{parent}(\text{jArcadioBD}, \text{remedios}) & \text{parent}(\text{jArcadioBD}, \text{jArcadioII}) \\ \text{parent}(\text{jArcadioBD}, \text{aurelianoII}) & \text{parent}(\text{aurelianoII}, \text{jArcadio}) \end{array} \right\} \tag{7}$$

$$E_2^+ = \left\{ \begin{array}{ll} \text{key}(\text{jArcadioBD}, \text{renata}) & \\ \text{male}(\text{jArcadioBD}) & \text{female}(\text{renata}) \\ \text{parent}(\text{jArcadioBD}, \text{remedios}) & \text{parent}(\text{jArcadioBD}, \text{jArcadioII}) \\ \text{parent}(\text{jArcadioBD}, \text{aurelianoII}) & \text{parent}(\text{renata}, \text{aureliano}) \\ \text{parent}(\text{aurelianoII}, \text{renata}) & \end{array} \right\} \tag{8}$$

$$E_1^- = \left\{ \begin{array}{ll} \text{key}(\text{sofia}, \text{renata}) & \\ \text{female}(\text{sofia}) & \text{female}(\text{renata}) \\ \text{parent}(\text{sofia}, \text{remedios}) & \text{parent}(\text{sofia}, \text{jArcadioII}) \\ \text{parent}(\text{sofia}, \text{aurelianoII}) & \end{array} \right\} \tag{9}$$

Each example contains facts related to two individuals. Note that facts may be repeated across examples: e.g., the first two examples share facts about jArcadioBD. From these examples an

ILP algorithm will construct a hypothesis such that interpretations E_1^+ and E_2^+ are models for it, whereas E_1^- is not.

Note, however, that in our example the generation that is the 'link' between grandfathers and grandchildren is present in the examples, since the members of this intermediate generation are connected to the terms in the key via the parent/2 predicate. Imagine, however, that we were trying to to formulate a hypothesis about a more general ancestor/2 predicate, by including $\mathrm{key(jArcadioBD, aureliano)}$ as a positive example. The properties of the key's arguments fail to include the *two* intermediate generations necessary to make the link between $\mathrm{jArcadioBD}$ and $\mathrm{aureliano}$.

This demonstrates the limitations of the non-monotonic setting when the data exhibits long-distance dependencies: the locality assumption under which learning from interpretations operates, keeps such long-distance dependencies from being present in the examples.

3.2 Setting up the Search

As mentioned earlier, the definition of the task of ILP is agnostic as to how to search for good hypotheses and even as to how to verify entailment.[3] Leaving aside the issue of mechanizing entailment for the moment, we discuss first how to choose candidate hypotheses among the members of the hypothesis language.

The simplest, naïve way to traverse this space of possible hypotheses would be to use a generate-and-test algorithm that first, lexicographically enumerates all possible hypotheses, and second, evaluates each hypothesis' quality. This approach is impractical for any non-trivial problem, due the large (possibly infinite) size of the hypothesis space. Instead, inductive machine learning systems (both propositional rule learning systems as well as first-order ILP systems) structure the search space in order to allow for the application of well-tested and established AI search techniques and optimizations. This structure must be such that, first, it accommodates a heuristics-driven search that reaches promising candidates as directly as possible; and, second, it allows for uninteresting sub-spaces to be efficiently pruned without having to wade through them.

3.2.1 Sequential Cover

Predictive ILP systems typically take advantage of the fact that the hypotheses are logic programs and therefore sets of clauses to use a strategy called *sequential cover* or *cover removal*, a greedy separate-and-conquer strategy that breaks the problem of learning a set of clauses into iterations of the problem of learning a single clause.

The iterative algorithm relies on a pool of *active* examples, that is, of examples which have not been covered yet. Initially, the pool is set to all the positive examples. Each iteration first calls a lower-level algorithm to search for a single clause of the target predicate. After finding a clause,

```
covering(E)
Input: Positive and negative examples $E^+, E^-$.
Output: A set of consistent rules.

1. $Rules\_Learned = \varnothing$
2. while $E^+ \neq \varnothing$ do
3.   $R = \textsc{LearnOneRule}(E, START\_RULE)$
4.   $Rules\_Learned = Rules\_Learned \cup R$
5.   $E^+ = E^+ \setminus \textsc{Coverage}(R, E^+)$
6. end while
7. return Rules_Learned
```

Figure 4: The Sequential Cover strategy. The `LearnOneRule()` procedure returns the best rule found that explains a subset of the positive examples E^+. `Coverage()` returns the subset of E^+ covered by R.

all positive examples that are covered by the clause are removed from the pool, so that the next iteration is guaranteed to find a different clause. Note that in this algorithm, as iterations proceed, clauses are evaluated on how well they cover the active examples, not the total set of positive examples. The process continues until all the examples are covered or some other termination criterion is met, e.g., a constraint on execution time or the maximum number of clauses. This procedure is also shown in Figure 4.

Most predictive ILP systems use some variant of this general strategy, and implement a variety of different algorithms to perform the clausal search that identifies the best clause to append to the theory at each iteration (step 4 in Figure 4).

3.2.2 Clause Generalization Search

In ILP, the search for individual rules is simplified by structuring the rule search space as a lattice or a semi-lattice, where a generality partial order is semantically defined using the concept of logical entailment, in the same way that we have discussed in the context of Version Spaces in the previous section.

Again, various generalization models have been proposed in the ILP literature, each based on different inference mechanisms for inferring entailment. Each of these defines a different *traversal operator* which maps conjunctions of clauses into generalizations (or, depending on the direction of the search, specializations) thereof.

Based on the above, the the `LearnOneRule()` procedure of Figure 4 works as shown in Figure 5, performing a lattice search by recursively transforming an initial clause into a series of hypotheses. The initial clause is one of the extremities of the lattice, and the search is either *open-ended* (for

learnOneRule(E, R_0 ***)***
Input: Examples E, used for evaluation. Initial rule R_0.
Output: A rule R.

```
1.  $BestRule = R = R_0$
2.  while stopping criteria not satisfied do
3.    $NewRules = \textsc{RefineRule}(R)$
4.    $GoodRules = \textsc{PickRules}(E, NewRules)$
5.    for $r \in GoodRules$
6.      $NewR = \textsc{LearnOneRule}(E, r)$
7.      if $NewR$ better than $BestRule$
8.        $BestRule = NewR$
9.      endif
10.   end for
11. end while
12. return $BestRule$
```

Figure 5: A procedure which returns the best rule found that explains a subset of the positive examples in E. The starting point of the search R_0 is one of the extremities of the lattice, `RefineRule()` is the lattice traversal operator, and `PickRules()` combines the search strategy, the heuristics, and prior constraints of valid rules to generate an ordered subset of the $NewRules$ set.

semi-lattices) or bound by the other extremity.

Regarding the extremities of the lattice, the maximally general clause (the *top clause*) of the generalization lattice is $\square$, the empty clause. The construction of the maximally specific clause varies with different ILP algorithms and generalization models, and can be *saturation* or *least general generalization*, or simply a ground example. Which one of the maximally general and the maximally specific assumes the rôle of the join and which of the meet depends on the direction of the search along the general–specific dimension.

3.2.3 The Elements of the Search

We have identified the elements of the ILP search lattice, the mathematical construct that defines a search problem equivalent to the ILP task, as logically formulated before. With these elements in place, we can proceed to apply a lattice search strategy, a methodology for traversing the search space looking for a solution. Search strategies are typically driven not only by the shape of the search space, but also by a heuristics, which evaluates each node of the lattice and estimates its ‘distance’ from a solution.

To recapitulate, in order to perform an ILP run, the following elements must be substantiated:

- The hypothesis language, which is the set of eligible hypothesis clauses. The elements of this set are also the elements that form the lattice.

- A traversal operator that in-order visits the search space, following either the general-to-specific or the specific-to-general direction.
- the top and bottom clause, the extremities of the lattice. The top clause is the empty-bodied clause that accepts all examples, and the bottom clause is a maximally specific clause that accepts a single examples. The bottom clause is constructed by a process called *saturation*.
- The search strategy applied to the lattice defined by the three elements above.
- The heuristics that drive the search, an estimator of a node's 'distance' from a solution. Due to the monotonic nature of the definite programs, this distance can be reasonably estimated by an evaluation function that assigns a value to each clause related to its 'quality' as a solution.

We now proceed to describe how various ILP algorithms and systems realize these five elements, and show the impact of each design decision on the system's performance. The focus will be on the clause-level search performed by sequential-cover predictive ILP systems, but it should be noted that the same general methodology is followed by theory-level predictive ILP and (to a lesser extend) descriptive ILP algorithms as well.

3.3 Hypothesis Language

One of the strongest, and most widely advertised, points in favour of ILP is the usage of prior knowledge to specify the hypothesis language within which the search for a solution is to be contained. ILP systems offer a variety of tools for specifying the exact boundaries of the search space, which modify the space defined by the *background theory*, the clausal theory that represents the concrete facts and the abstract, first-order generalizations that are known to hold in the domain of application.

In more practical terms, the *background predicates*, the predicates defined in the background theory, are the building blocks from which hypothesis clauses are constructed. The background predicates should, therefore, represent all the relevant facts known about the domain of the concept being learnt; the ILP algorithm's task is to sort though them and identify the ones that, connected in an appropriate manner, encode the target concept. Even ILP algorithms that go one step further and revise the background theory (cf. Section 3.8.2 below), rely on the original background to use as a starting point.

Prior knowledge in ILP is not, however, restricted to the set of predefined concepts available to the ILP algorithm, but extends to include a set of syntactic and semantic restrictions imposed on the set of admissible clauses, effectively restricting the search space. Such restrictions cannot be encoded in the background theory program of Definitions 5 and 5, but ILP systems provide external mechanisms in order to enforce them.[4] These mechanisms should be thought of as filters that reject or accept clauses—and, in some cases, whole areas of the search space—based on syntactic or semantic pre-conditions.

3.3.1 Hypothesis Checking

As explained before, managing the ILP search space often depends on having filters that reject uninteresting clauses. The simplest, but crudest approach, is to include or omit background predicates. By contract, some ILP systems, e.g., ALEPH (Srinivasan, 2004), allow users to build themselves filters that verify whether the clause is eligible for consideration or whether it should be immediately dropped. Such a *hypothesis checking* mechanism allows the finest level of control over the hypothesis language, but is highly demanding on the user and should be complemented with tools or mechanisms that facilitate the user's task.

Hypothesis checking mechanisms can be seen as a way to bias the search language, and therefore usually known as the *bias language*. One very powerful such example are declarative languages such as *antecedent description grammars*, definite-clause grammars that describe acceptable clauses (Cohen, 1994; Jorge & Brazdil, 1995), or the DLAB language of clause templates (Dehaspe & De Raedt, 1996) used in CLAUDIEN (De Raedt & Dehaspe, 1996), one of the earliest descriptive ILP systems. As Cohen notes, however, such grammars may become very large and cumbersome to formulate and maintain; newer implementations of the CLAUDIEN algorithm (CLASSICCL, Stolle et al. 2005) also abandon DLAB and replace it with type and mode declarations (see below).

Finally, note that hypothesis checking mechanisms can verify a variety of parameters, both syntactic and semantic. As an example of semantic parameter, it is common for ILP systems to discard clauses with very low coverage, as such clauses often do not generalize well.

3.3.2 Determinacy

A further means of controlling the hypothesis language is through the concept of *determinacy*, introduced in this context by Muggleton & Feng (1990). Determinacy is a property of the variables of a clause and, in a way, specifies how 'far' they are from being bound to ground terms: a variable in a term is j-determinate if there are up to j variables in the same term. Furthermore, a variable is ij-determinate if it is j-determinate and its *depth* is up to i. A clause is ij-determinate is all the variables appearing in the clause are ij-determinate.

The depth of a variable appearing in a clause is recursively defined as follows :

Definition 7 *For clause C and variable v appearing in C, the* depth *of v, $d(v)$, is*

$$d(v) = \begin{cases} 0 & \textit{if } v \textit{ is in the head of } C \\ 1 + \min_{w \in \mathrm{var}(C,v)} d(w) & \textit{otherwise} \end{cases}$$

where $\mathrm{var}(C, v)$ are all the variables appearing in those atoms of the body of C where v also appears.

The intuition behind this definition is that depth increases by 1 each time a body literal is needed to 'link' two variables, so that a variable's depth is the number of such links in the 'chain' of body literals that connects the variable with a variable of the head.[5] Simply requiring that the hypothesis language only include determinate clause, i.e., clauses with arbitrarily large but finite determinacy parameters, has a dramatic effect in difficulty of the definite ILP task, as it renders definite hypotheses PAC-learnable (De Raedt & Džeroski, 1994). Naturally, specifying smaller values for the determinacy parameters tightens the boundaries of the search space. Consider, for example, the following clause as a possible solution for our grandfather-learning example:

$$C = \mathrm{grandfather}(X, Y) \leftarrow \mathrm{father}(X, U) \wedge \mathrm{father}(U, V) \wedge \mathrm{mother}(W, V) \wedge \mathrm{mother}(W, Y) \quad (10)$$

This clause perfectly classifies the instances of our example (Figure 3) and is $2, 2$-determinate.[6] Although this clause satisfies the posterior requirements for a hypothesis, a domain expert might have very good reasons to believe that the grandfather relation is a more direct one, and that solutions should be restricted to $1, 2$-determinate clauses. Imposing this prior requirement would force a learning algorithm to reject clause C (Eq. 10) as a hypothesis and formulate a more direct one, for example:

$$D = \mathrm{grandfather}(X, Y) \leftarrow \mathrm{father}(X, U) \wedge \mathrm{parent}(U, Y) \quad (11)$$

3.3.3 Type and Mode Declarations

Type and *mode declarations* are further bias mechanisms that provide ILP algorithms with prior information regarding the semantic properties of the background predicates as well as the target concept. Type and mode declarations were introduced in PROGOL (Muggleton, 1995) and have since appeared in many modern ILP systems, most notably including the CLASSICCL (Stolle et al., 2005) re-implementation of CLAUDIEN, where they replace the declarative bias language used in the original implementation.

Mode declarations state that some literal's variables must be bound at call time (input, $+$) or not (output, $-$) and thus restrict the ways that literals can combine to form clauses. Mode declaration also specify the places where constants may appear (constant, #). For example, with the following mode declarations in the background:

$$B = \left\{ \begin{array}{l} \mathrm{mode}(\mathrm{grandfather}(+, -)) \\ \mathrm{mode}(\mathrm{father}(+, +)) \\ \mathrm{mode}(\mathrm{parent}(+, +)) \end{array} \right\} \quad (12)$$

clause D (Eq. 11) can not be formulated, as variable Y cannot be bound in the parent/2 literal if it is unbound in the head. The desired 'chain' effect of input/output variables may be achieved with the following background:

$$B = \left\{ \begin{array}{l} \text{mode}(\text{grandfather}(+, -)) \\ \text{mode}(\text{father}(+, -)) \\ \text{mode}(\text{parent}(+, -)) \end{array} \right\} \tag{13}$$

achieves the desired 'chain' effect of input/output variables: X is bound in the head, so that father/2 can be applied to bind the unbound variable U, which in its turn can serve as input to parent/2. Modes are usually combined with *recall bounds* that restrict the number of possible instantiations of a literal's variables for each mode of application:

$$B = \left\{ \begin{array}{l} \text{mode}(1, \text{grandfather}(+, +)) \vee \text{mode}(2, \text{grandfather}(+, -)) \\ \text{mode}(1, \text{father}(+, +)) \vee \text{mode}(3, \text{father}(+, -)) \\ \text{mode}(1, \text{parent}(+, +)) \vee \text{mode}(3, \text{parent}(+, -)) \\ \text{mode}(1, \text{female}(+)) \vee \text{mode}(5, \text{female}(-)) \end{array} \right\} \tag{14}$$

This background restricts the second mode of $\text{grandfather}/2$ to two possible instantiations, so that hypothesis D (Eq. 11) becomes unattainable and a different solution has to be identified. Consider, for example:

$$E = \text{grandfather}(X, Y) \leftarrow \text{father}(X, U) \wedge \text{parent}(U, Y) \wedge \text{female}(Y) \tag{15}$$

Clause E focuses on grandfathers of grand-daughters, and so satisfies the requirements of Eq. 14.

Finally, mode declarations are extended with a rudimentary type system, such that different modes apply to different instances. To demonstrate, the following background knowledge:

$$B = \left\{ \begin{array}{l} \text{mode}(2, \text{grandfather}(+M, -M)) \vee \text{mode}(2, \text{grandfather}(+F, -F)) \\ \text{mode}(2, \text{father}(+M, -M)) \vee \text{mode}(2, \text{father}(+M, -F)) \\ \text{mode}(2, \text{parent}(+M, -M)) \vee \text{mode}(2, \text{parent}(+M, -F)) \\ \vee\text{mode}(2, \text{parent}(+F, -M)) \vee \text{mode}(2, \text{parent}(+F, -F)) \\ \text{mode}(1, \text{female}(+F)) \vee \text{mode}(5, \text{female}(-F)) \\ \text{mode}(1, \text{male}(+M)) \vee \text{mode}(5, \text{male}(-M)) \end{array} \right\} \tag{16}$$

admits hypotheses like D (Eq. 11) which cover up to two grandchildren of each gender. Note, however, that types in ILP systems are most often flat tags; they do not support the supertype–subtype hierarchy that is so fundamental to modern type systems. As a result, it is not, for example, possible to express the fact that that although the $\text{parent}/2$ predicate can be instantiated for up to 2 children of each gender, it can be instantiated for up to 3 children of either gender, as this would require the definition of a *person* supertype for the M and F types.

3.4 Hypothesis Space Traversal

As already seen above, the elements of the hypothesis language are organized in a lattice by a partial-ordering operator. In theory, it would suffice to have an operator that simply evaluates the relation between two given clause as ‘more general than’, ‘less general than’, or neither. This would, however, be impractical since it would require that the whole search space is generated and the relationship between all pairs of search nodes is evaluated before the search starts.

Instead, ILP systems use generative operators which map a given clause into a set of clauses that succeed the original clause according to the partial ordering.[7] Using such a *traversal operator* only the fragment of the search space that is actually visited is generated as the search proceeds. There are three main desiderata for the traversal operator:

- it should *only* generate clauses that are in the search space,
- it should generate *all* the clauses that are in the search space,
- it should be *efficient* and generate the most interesting candidate hypotheses first.

Satisfying these points does not only involve the traversal operator, but also its interaction with the other elements of the search. Prior knowledge, in particular, defines the hypothesis language and thus the elements of the search space and operates on the assumption of a monotonic entailment structure of the search space: whenever $C <_g D$, it must be that $C \models D$ (Definition 3). This places the requirement that the traversal operator is a syntactic inference operator that mechanizes the process of validating semantic entailment. Deductive inference operators that deduce D from C *only* when $C \models D$ are called *sound* and those that deduce D from C for *all* C, D where $C \models D$ are called *complete*.

A variety of sound first-order deductive inference operators have been proposed in the logic programming literature, each with their own advantages and limitations with respect to completeness and efficiency. ILP borrows the deductive operators from the logic programming community for searching in the general-to-specific direction and derives their inductive inversions for the specific-to-general direction.

3.4.1 Subsumption

The earliest approaches to first-order induction were based on θ-subsumption (Plotkin, 1970), a sound deductive operator which deduces clause B from clause A if the antecedents of A are a subset of the antecedents of B, up to variable substitution:

Definition 8 *If there is a substitution θ such that $A\theta \subseteq B$, then A θ-subsumes B ($A \preceq B$).*

So, for example, given the clauses C and D:

$$\begin{array}{rcl} C & = & \mathrm{mother}(X,Y) \leftarrow \mathrm{parent}(X,Y) \wedge \mathrm{female}(X) \\ D & = & \mathrm{mother}(\mathrm{sofia},Y) \leftarrow \mathrm{parent}(\mathrm{sofia},Y) \wedge \mathrm{female}(\mathrm{sofia}) \wedge \mathrm{female}(Y) \end{array} \tag{17}$$

it is the case that $C \preceq D$, since $C\theta \subseteq D$ for $\theta = \{X/\mathrm{sofia}\}$. And, indeed, it is also the case that C entails D since the mothers of daughters are a subset of mothers of children of either gender.

In practice, θ-subsumption is a purely syntactic operator: using θ-subsumption to make a clause more specific amounts to either adding a literal or binding a variable in a literal to a term. Inversely, generalization is done by dropping literals, replacing ground terms with variables, or introducing a new variable in the place of an existing one that occurs more than once in the same literal. When searching in either direction (specific-to-general or general-to-specific) between a (typically very specific) clause $\{H, \neg B_1, \neg B_2, \ldots, \neg B_n\}$ and the most general clause $\{H\}$ (where H contains no ground terms and all variables are different), the search space is confined within the power-set of $\{\neg B_i\}$

The FOIL algorithm (Quinlan, 1990) uses θ-subsumption to perform open-ended search. FOIL is the natural extension of the propositional rule-learning system CN2 (Clark & Niblett, 1989) to first order: it employs the same open-ended search strategy starting with an empty-bodied top clause and specializing it by adding literals. FOIL searches using a best-first strategy, where each step consists of adding all possible literals to the current clause, evaluating their quality, and then advancing to the next 'layer' of literals once the best clause at the current clause length has been identified. The list of candidate-literals at each layer consists of all background predicates with all possible arguments according to the current language, that is, new variables, variables already appearing in the body so far, and all constants and function symbols. FOIL allows some bias to constraint the search space. First, each new literal must have have at least one variable that already appears in the current clause. Second, FOIL supports a simple type system that restricts the number of constants, functions and variables that can be placed as arguments.

3.4.2 Relative Least General Generalization

The direct approach of performing an open-ended θ-subsumption can generate search spaces with a very high branching factor. The problem is that whenever we expand a clause with a new literal we need to consider *every* literal allowed by the language, that is, every literal in the language for which we have input variables bound. This is especially problematic if the new literal can have constants as arguments: in this case, we always need to consider every constant in the (typed) language when we expand a clause with a literal.

In order to achieve more informed search-coming, Muggleton & Feng (1990) introduced the concept of the *bottom clause*, the minimal generalization of a set of clauses. The process of achieving this idea is somewhat involved, but the original idea was based on Plotkin's (1970) work on inductive operators and, more specifically, work on defining the *least general generalization* (LGG) of two given terms as the most specific term that θ-subsumes them. Essentially, the LGG inverts unification by introducing variables that replace ground atoms in the original terms. See, as an example, the following clauses and their LGG:

$$\begin{aligned} E_1 &= \mathrm{grandfather}(\mathrm{jArcadioBD}, \mathrm{renata}) \\ E_2 &= \mathrm{grandfather}(\mathrm{jArcadioBD}, \mathrm{amaranta}) \\ \mathrm{lgg}(E_1, E_2) &= \mathrm{grandfather}(\mathrm{jArcadioBD}, X) \end{aligned} \tag{18}$$

LGG generalizes two (or more, by repeated application) terms in the absence of any background predicates that can be used to restrict the variables introduced. Plotkin (1971a) proceeds to define the *relative least general generalization* (RLGG) of a set of terms relative a set of background terms. RLGG generalizes the set of terms into a single ungrounded term and uses the background knowledge to restrict the possible bindings of the variables introduced. Plotkin's original idea was to induce theories by applying the RLGG to a set of examples relative to the background:

$$\begin{aligned} \mathrm{rlgg}(\{E_i\}, \{B_i\}) = {} & \mathrm{grandfather}(X, Y) \leftarrow \mathrm{father}(X, Z) \wedge \mathrm{parent}(X, Z) \wedge \\ & \mathrm{parent}(Z, Y) \wedge \mathrm{sibling}(U, Z) \wedge \mathrm{female}(U) ... \end{aligned} \tag{19}$$

where the E_i are the positive examples in Eq. 6 and the B_i are the ground facts and all extensions of the intensionally defined predicates in Eq. 5. We only show a fragment of the full RLGG here; the full clause contains all possible ways to relate the argument X with the argument Y, but even the fragment shown here is already overly specific and—correct as this theory might be according to the data—a shorter theory can be identified that accurately classifies the examples. In general, Plotkin (1971a) notes that the RLGG can be very long even for ground clauses, and potentially infinite when ungrounded clauses are involved, prohibiting its practical application.

Muggleton & Feng (1990) observe that the RLGG of the examples is an accurate but unnecessarily specific theory: thus, it should be considered a starting point to be further refined by searching in the space of clauses that can be found in the θ-subsumption lattice bounded between the *RLGG bottom* and an empty-bodied *top clause*. As a second step, and in order to avoid infinite RLGG bottoms, the hypothesis language is further restricted to *ij-determinate* definite clauses. Under the ij-determinacy restriction, the RLGG of a set of ground clauses (examples) relative to a set of background clauses is unique and finite.

These ideas were originally implemented in GOLEM, an ILP system which alleviates Plotkin's problem of unnecessarily specific and long hypotheses while at the same time bounding the search into clauses that take into account the totality of the background and the examples. This was a moment of paramount importance in ILP research, as it marked the transition from open-ended searches in semi-lattices to full-lattice searches; a development which, although by no means sufficient to guarantee termination, at least voided one of the possible causes of non-termination.

3.4.3 Inverse Resolution

Although θ-subsumption search have been successfully used in the early days of ILP, it has been known that, as Plotkin (1971b) originally noted, θ-subsumption is not *complete*, or, in other words, it is not able to infer B from A in all cases where A entails B; it naturally follows that inverse θ-subsumption is also not complete. Consider, for example, the following clauses:

$$\begin{aligned} A &= \text{ancestor}(X,Y) \leftarrow \text{parent}(X,Z) \wedge \text{ancestor}(Z,Y) \\ B &= \text{ancestor}(X,Y) \leftarrow \text{parent}(X,Z) \wedge \text{parent}(Z,W) \wedge \text{ancestor}(W,Y) \end{aligned} \tag{20}$$

where B represents a stricter form of 'ancestry' than A, so that all models of A are also models of B. But, although $A \models B$, there is no variable substitution θ such that $A\theta \subseteq B$.[8] This incompleteness naturally propagates to inverse θ-subsumption as well: clause B cannot be generalized into A by inverse θ-subsumption (effectively, literal dropping and variable substitution) although A is more general than B.

In order to close the gap between true implication and θ-subsumption, ILP research has turned to complete deductive inference operators, like Robinson's *resolution rule*. The Robinson resolution rule (1965) raises the propositional resolution rule to the first order, and is defined as follows:

Definition 9 *Clause R is resolved from clauses C_1 and C_2 if and only if there are literals $l_1 \in C_1$ and $l_2 \in C_2$ and substitution θ such that:*

$$\begin{aligned} l_1\theta &= \neg l_2\theta \\ R &= (C_1 - \{l_1\})\,\theta \cup (C_2 - \{l_2\})\,\theta \end{aligned}$$

By algebraically solving the equation in Definition 9 for C_2, an *inverse resolution* operator can be defined which generalizes clause R, given background clause C_1:

$$C_2 = (R - (C_1 - \{l_1\})\,\theta_1)\,\theta_2^{-1} \cup \left\{\neg l_1\theta_1\theta_2^{-1}\right\} \tag{21}$$

where $l_1 \in C_1$ and $\theta_1\theta_2$ is a factorization of the unifying substitution θ from Definition 9, such that θ_i contains all and only substitutions involving variables from C_i. Such a factorization is always possible and unique, because C_1 and C_2 are distinct clauses, hence the variables appearing in C_1 are separate from those appearing in C_2.

The above generalization operator was the basis of CIGOL (Muggleton & Buntine, 1988), one of the earliest successful ILP systems. CIGOL follows a sequential-covering strategy, with individual clause search proceeding in the specific-to-general direction. The starting point of the search is a ground positive example, randomly selected from the pool of uncovered positives, that is generalized by repeated application of inverse resolution. The advantage is that *all* and *only* consistent clauses are generated, allowing for a complete, yet focused search.

3.4.4 Mode-Directed Inverse Entailment

While inverted resolution is complete, it disregards the background (modulo a single background clause at each application), so it does not provide for a focused search. RLGG-based search, on the other hand, does take into account the whole of the background theory, but it relies on inverting θ-subsumption, so it performs an incomplete search that can potentially miss good solutions.

In order to combine completeness with the informedness of the minimal-generalization bottom, Muggleton (1995) explores implication between clauses and ways of reducing inverse implication to a θ-subsumption search without loss of completeness, and defines the *inverse entailment* operator, based on the following proposition:

Lemma 1 *Let C, D be definite, non-tautological clauses and $S(D)$ be the sub-saturants of D. Then $C \models D$ if and only if there exists $C' \in S(D)$ such that $C \preceq C'$.*

where the sub-saturants of a clause D are, informally,[9] all ungrounded clauses that can be constructed from the symbols appearing in D *and* cannot prove the complement of D (i.e., do not resolve to any of the atoms in the Herbrand model of the complement of D).

Inverse entailment starts by *saturating* a ground positive example into a *bottom clause*, an ungrounded clause which can prove only one ground positive example and no other. Because of Lemma 1, any clause which entails the original example will also θ-subsume the bottom clause; it is now sufficient to perform a θ-subsumption search in the full lattice between the empty-bodied top clause and the bottom clause without loss of completeness.

Mode-directed inverse entailment (MDIE) is a further refinement of inverse entailment where the semantics of the background predicates are taken into consideration during saturation (see below). The search in MDIE is organized in the general-to-specific direction in the original PROGOL system (Muggleton, 1995), as well as in most other successful MDIE ILP systems, like, e.g., ALEPH (Srinivasan, 2004), INDLOG (Camacho, 2000), CILS (Anthony & Frisch, 1999), and APRIL (Fonseca, 2006) although the alternative direction has been explored as well (Ong et al., 2005).

3.5 Saturation

Saturation is the process of constructing a *bottom clause*[10] which constitutes the most-specific end of the search lattice in mode-directed inverse entailment. The most prominent characteristic of saturation is that it fills the 'gap' between θ-subsumption and entailment: the literals of the bottom clause are the sub-saturants of a clause C, hence identifying clauses that entail C amounts to identifying clauses that θ-subsume the bottom clause (Lemma 1).

Saturation is performed by repeated application of inverse resolution on a ground example, called the *seed*. This introduces variables in place of the ground terms in a 'safe' manner: the bottom clause is guaranteed to include all of and only the sub-saturants of the seed.

Let us revisit the clauses in Eq. 20, used to demonstrate the incompleteness of θ-subsumption. Given the example $\mathrm{ancestor}(\mathrm{jArcadioBD}, \mathrm{aureliano})$ as a seed, saturation would yield the following bottom clause:

$$\begin{aligned} \bot_1 \quad = \quad & \mathrm{ancestor}(A, B) \leftarrow \\ & \mathrm{parent}(A, C) \wedge \mathrm{parent}(A, D) \wedge \mathrm{parent}(A, E) \wedge \mathrm{ancestor}(A, C) \wedge \\ & B = aureliano \wedge A = jArcadioBD \wedge \mathrm{parent}(E, F) \wedge \mathrm{parent}(E, G) \wedge \\ & \mathrm{parent}(E, H) \wedge \mathrm{sibling}(E, D) \wedge \mathrm{sibling}(E, C) \wedge \mathrm{sibling}(D, E) \wedge \\ & \mathrm{sibling}(D, C) \wedge \mathrm{sibling}(C, E) \wedge \mathrm{sibling}(C, D) \wedge \mathrm{ancestor}(E, B) \wedge \\ & \mathrm{ancestor}(A, E) \wedge \mathrm{ancestor}(A, D) \wedge \mathrm{ancestor}(E, F) \wedge E = aurelianoII \wedge \\ & D = jArcadioII \wedge C = remedios \wedge \mathrm{parent}(G, B) \wedge \mathrm{sibling}(H, G) \wedge \\ & \mathrm{sibling}(H, F) \wedge \mathrm{sibling}(G, H) \wedge \mathrm{sibling}(G, F) \wedge \mathrm{sibling}(F, H) \wedge \\ & \mathrm{sibling}(F, G) \wedge \mathrm{ancestor}(G, B) \wedge \mathrm{ancestor}(E, H) \wedge \mathrm{ancestor}(A, H) \wedge \\ & \mathrm{ancestor}(E, G) \wedge \mathrm{ancestor}(A, G) \wedge \mathrm{ancestor}(A, F) \wedge H = \mathrm{amaranta} \wedge \\ & G = \mathrm{renata} \wedge F = \mathrm{jArcadio} \end{aligned} \tag{22}$$

Notice that $\bot_1$ is θ-subsumed by *both* clauses of Eq. 20.[11] In general, for all clauses A, B that subsume the seed, if A entails B, a complete generalization operator would first inverse-resolve B and, later in the search, A. In cases where $A \not\preceq B$, the bottom clause will complete a θ-subsumption search by including enough body literals to be θ-subsumed by both A and B.

Besides completing θ-subsumption search, saturation is also designed to take type-mode and determinacy declarations into consideration, so that they do not need to be explicitly checked during the search: saturation guarantees a bottom clause that is θ-subsumed only by conforming clauses. Consider, for example, a background in which all predicates are required to be determinate. In this case the bottom clause of $\mathrm{ancestor}(\mathrm{jArcadioBD}, \mathrm{aureliano})$ would be:

$$\begin{aligned} \bot_1 \quad = \quad & \mathrm{ancestor}(A, B) \leftarrow \mathrm{parent}(A, C) \wedge \mathrm{ancestor}(A, C) \wedge \\ & B = aureliano \wedge A = jArcadioBD \wedge C = \mathrm{remedios} \end{aligned} \tag{23}$$

since the constraint that all predicate be determinate can only be satisfied in our example model (Figure reffig:family) by the $\mathrm{ancestor}(\mathrm{renata}, \mathrm{aureliano})$ term.

Similarly, restricting variable depth to 1 would yield the following bottom clause:

$$\begin{aligned} \bot_1 \quad = \quad & \mathrm{ancestor}(A, B) \leftarrow \\ \bot_1 \quad = \quad & \mathrm{ancestor}(A, B) \leftarrow \\ & \mathrm{parent}(A, C) \wedge \mathrm{parent}(A, D) \wedge \mathrm{parent}(A, E) \wedge \mathrm{ancestor}(A, C) \wedge \\ & B = \mathrm{aureliano} \wedge A = \mathrm{jArcadioBD} \end{aligned} \tag{24}$$

which is the subset of the bottom clause in Eq. 22 where all variables are of depth 0 or 1. Notice that there are no ancestor/2 literals, as they amount to ancestor/2 literals when depth is restricted to 1.

3.6 Search Strategy

Given a search space with some structure, we can consider a strategy to traverse the search space. The most important components of the strategy are the *direction* of the search and the *method* used to achieve it.

The *direction* of the search can be either top-down or bottom-up, depending on the traversal operator used. In a top-down, general-to-specific search the initial rule is the top clause, which is incrementally specialized through the repeated application of downward traversal operators, until inconsistencies with negative examples have been removed. In a bottom-up, specific-to-general search, on the other hand, a most specific clause is generalized by applying upward traversal operators. The original, most specific clause can be either one or more ground examples or the bottom clause.

More recently, there has been interest on stochastic approaches to search that do not properly fall under either of these strategies. Examples of these approaches are Stochastic Clause Selection and Randomized Rapid Restarts algorithms. (Železný et al., 2003; Tamaddoni-Nezhad & Muggleton, 2003; Srinivasan, 2000).

A second component of the search is the search method used to find a hypothesis. As is generally the case with problem solving in Artificial Intelligence, search methods can be either uninformed or informed, and can rely on heuristics to estimate how 'promising' or close to a solution each node of the search space is.

3.6.1 Uninformed Search

Some of the most well-known uninformed search methods are breadth-first, depth-first, and iterative-deepening search. *Breadth-first* search is a simple method in which the root node is expanded first, then all direct successors of the root node are expanded, then all their successors, and so on. In general, all the nodes are expanded at a given depth of the search before any nodes in the following level are expanded. The main advantage of this method is that it is guaranteed to find a solution if one exists. The main drawbacks are the high memory requirements leading to poor efficiency. Several ILP systems use this search method, e.g., MIS (Shapiro, 1983) and WARMR (Dehaspe & Toironen, 2000). An advantage is that it visits clauses near the top of the lattice first, favouring clauses that are shorter and more general.

By contrast, *depth-first* search expands the deepest node first. This search strategy has the advantage of having lower memory requirements, since a node can be removed from memory after being expanded as soon all its descendants have been fully explored. On the other hand, this strategy can get lost in infinite branches. A variant of depth-first search, called *depth-limit search* imposes a limit on the depth of the graph explored. Obviously, this search is incomplete if a solution is in a region deeper than this pre-set limit.

Iterative depth-first search (IDS) is a general strategy often used in combination with depth-first search, that finds the best depth limit. It does this by gradually increasing the depth limit (first 0, then 1, then 2, and so on) until a solution is found. The drawback of IDS is the wasteful repetition of the same computations. Iterative deepening search is used in ILP systems like ALEPH (Srinivasan, 2004) and INDLOG (Camacho, 2000).

3.6.2 Informed Search

Informed (heuristic) search methods can be used to find solutions more efficiently than uninformed but can only be applied if there is a measure to evaluate and discriminate the nodes. Several heuristics have been proposed for ILP.

A general approach to informed searching is best-first search. *Best-first* search is similar to breadth-first but with the difference that the node selected for expansion is the one with the least value of an evaluation function that estimates the 'distance' to the solution.

Note that the evaluation functions will return an *estimate* of the quality of a node. This estimate is given by a *heuristics*, a function that tries to predict the cost/distance to the solution from a given node.

Best-first search is used in many ILP systems, like FOIL,CIGOL, INDLOG, and ALEPH. A form of best-first search is *greedy best-first* search as it expands only the node estimated to be closer to the solution on the grounds that it would lead to a solution quickly. The most widely known form of best-first search is called A^* *search*, which evaluates nodes by combining the cost to reach the node and the distance/cost to the solution. This algorithm is used by the PROGOL ILP system.

The search algorithms mentioned so far explore the search space systematically. Alternatively, *local search algorithms* is a class of algorithms commonly used for solving computationally hard problems involving very large candidate solution spaces. The basic principle underlying local search is to start from an initial candidate solution and then, iteratively, make moves from one candidate solution to another candidate solution *in its direct neighbourhood.* The moves are based on information local to the node, and continue until a termination condition is met. Local search algorithms, although not systematic, have the advantage of using little memory and can find reasonable solutions in large or infinite search spaces, where systematic algorithms are unsuitable.

Hill-climbing search Russell & Norvig (2003) is one example of a local search algorithm. The algorithm does not look ahead beyond the immediate successor nodes of the current node, and for this reason it is sometimes called *greedy* local search. Although it has the advantage of reaching a solution more rapidly, it has some potential problems. For instance, it is possible to reach a *foothill* (*local maxima* or *local minima*), a state that is better than all its neighbours but it is not better than some other states farther away. Foothills are potential traps for the algorithm. Hill-climbing is good for a limited class of problems where we have an evaluation function that fairly accurately predicts the actual distance to a solution. ILP systems like FOIL and FORTE (Richards & Mooney, 1995) use hill-climbing search algorithms.

3.7 Preference Bias and Search Heuristics

Prior knowledge representations described so far express strict constraints imposed on the hypothesized clauses, realized in the form of yes–no decisions on the acceptability of a clause. The previous section demonstrated the need for heuristics that identify the solution closest to the unknown target predicate, but, in the general case, there may be multiple solutions that satisfy the requirements of the problem; or we may have no solution but still want the best approximation. ILP algorithms therefore need to choose among partial solutions and provide justification for this choice besides consistency with the background and the examples. *Preference bias* does just that: it assigns a 'usefulness' rating to clauses which typically says how close we are to the best solution. In practice, the is translated into an *evaluation function* that tries to estimate how well a clause will perform when applied to data unseen during training.

The user may provide general preferences, for instance towards shorter clauses, or towards clauses that achieve wide coverage, or towards clauses with higher predictive accuracy on the training data, or any other problem-specific preference deemed useful. Preference bias also must navigate a balance between specificity and generality: a theory with low coverage may describe the data too tightly and may not generalize well (overfit). Indeed, at its extreme may it only accepts the positive examples it was constructed from and nothing else. On the other hand, a theory that makes unjustified generalizations can underfit, and at the extreme, accept everything as a positive.

3.7.1 Entropy and m-Probability

Evaluation functions will typically judge a clause semantically and so the most commonly used evaluation functions refer to a clause's coverage over positive and negative examples. Such functions include simple coverage, information gain (Clark & Niblett 1989, based on the concept of entropy as defined by Shannon 1948), m-probability estimate (Cestnik, 1990), and the Laplace expected accuracy:

$$\begin{aligned}
\mathrm{Coverage}(P,N) &= P - N && (25)\\
\mathrm{InfGain}(P,N) &= -P_{rel}\cdot\log_2 P_{rel} - N_{rel}\cdot\log_2 N_{rel} && (26)\\
\mathrm{MEst}_{m,P_0}(P,N) &= (P + m\cdot P_0)\,/\,(P+N+m) && (27)\\
\mathrm{Laplace}(P,N) &= (P+1)\,/\,(P+N+2) && (28)
\end{aligned}$$

where P, N is the positive and negative coverage, $P_{rel} = P/(P+N)$, $N_{rel} = 1 - P_{rel}$ the relative positive and negative coverage, P_0 is the prior probability of positive examples (i.e., the probability that a random example is positive) and m a parameter that weighs the prior probability against the observed accuracy. It should be noted that the Laplace function is a specialization of the m-probability estimate for uniform prior distribution of positives and negatives ($P_0 = 0.5$) with the m parameter set to a value of 2, a value that balances between prior and observation.

	Inf. Gain	Laplace Acc.
C_1 (980,20)	0.141	0.979
C_2 (98,2)	0.141	0.971
C_3 (1,0)	1.000	0.667

Table 1: Evaluation of three clauses (positive and negative coverage in parenthesis) by the Information Gain function and the Laplace Accuracy function.

The advantage that accuracy estimation offers over information gain is that it favours more general clauses, even at the expense of misclassifying a small number of training examples. This makes it more appropriate for real-world applications where the training data contains noise and inconsistencies, since it will generalize at the expense of outliers.

As an example, let us consider a situation where three possible clauses C_1, C_2 and C_3 meet the minimum accuracy constraint we have set, and a choice needs to be made according to preference bias. Let us assume that the examples covered by each clause are $(980, 20)$, $(98, 2)$ and $(1, 0)$ respectively, where each pair represents the number of positive and negative examples covered by the clause. Knowing that we are dealing with noisy data, the intuitive choice would be C_1, since it performs as well as C_2, but seems to be making a much broader generalization. And we would argue that both are preferable to C_3, which is simply re-iterating a piece of the training data.

The evaluation of these three clauses by Information Gain and Laplace accuracy is shown in Table 1. Evaluation functions that focus on minimizing entropy assign absolute preference to perfectly 'pure' partitions (like the one imposed by C_3), even in the trivial cases of maximally-specific rules that cover exactly one example. But even assuming that such extreme situations are treated by, for example, minimum coverage filters, information gain is unable to distinguish between C_1 and C_2, since they have identical relative coverage (980/1000 and 98/100).

Accuracy estimation, on the other hand, balances between rewarding wide coverage and penalizing low accuracy on the training data, the point of balance determined by the m parameter.

3.7.2 Bayesian Evaluation

Another approach to clause evaluation is derived from Bayes' Theorem (Bayes, 1763, Proposition 3), which provides a formula for calculating $\Pr(A|B)$, the probability of event A given event B:

Theorem 2 *Given random variables A, B with probabilities $\Pr(A)$ and $\Pr(B)$, then the probability of A given B is*

$$\Pr(A|B) = \frac{\Pr(B|A) \cdot \Pr(A)}{\Pr(B)}$$

where $\Pr(B|A)$ is the probability of B given A.

Suppose we have a clause of the form $A \leftarrow B$, and $\Pr(A)$, $\Pr(B)$ are the probabilities that predicates A and B hold for some random example. We can now apply Bayes' Theorem to calculate the probability that, for some random example, if the clause's antecedents hold, then the subsequent holds. In a semantics sense, this is the probability that the clause accurately classifies the example. Probability $\Pr(A|B)$ is called the *posterior distribution* of A, by contrast to *prior distribution* $\Pr(A)$, the distribution of A before seeing the example.

Bayes (idem., Proposition 7) continues by using the empirical data to estimate an unknown prior distribution. This last result is transferred to ILP by Cussens (1993) as the pseudo-bayes[12] evaluation function:

$$\begin{aligned} Acc &= P/(P+N) \\ K &= Acc \cdot (1 - Acc)\,/\,(P_0 - Acc)^2 \\ \text{PBayes} &= (P + K \cdot P_0)\,/\,(P+N+K) \end{aligned} \tag{29}$$

where P_0 is the positive distribution observed in the data, and P and N are the positive and negative coverage of the clause. A comparison of the pseudo-Bayes evaluation function with m-estimation (Eq. 27) shows that one way of looking at pseudo-Bayes evaluation is as an empirically justified way of calculating the critical m parameter of m-estimation.

3.7.3 Syntactic Considerations

Some evaluation functions will also refer to syntactic aspects of the hypothesized clause, implementing (for example) preference bias towards shorter or simpler clauses, in accordance with the general principle of parsimony known as Ockham's Razor. Unfortunately, quantifying 'simplicity' is hard, and there is no universal answer to what a 'simple' hypothesis is. Most approaches are equating being the simplest to being the smallest, in accordance with Rissanen's (1978) *minimum description length* (MDL) principle that the most probable hypothesis is the one that can be most economically encoded.

Such evaluation functions range from simply counting the number of literals in a clause, to taking into account each literal's structural complexity:

$$\text{size}(T) = \begin{cases} 1 & \text{if } T \text{ is a variable} \\ 2 & \text{if } T \text{ is a constant} \\ 2 + \sum_{i=1}^{n} \text{size}\left(\text{arg}_i\,(T)\right) & \text{if } T \text{ is a term of arity } n \end{cases} \tag{30}$$

This quantification of clause complexity was used in the MDL evaluation function of the COCKTAIL system (Tang & Mooney, 2001).

On the other hand, Muggleton, Srinivasan, & Bain (1992) take a different approach and re-introduce a semantic element in the notion of logic program complexity: the *proof complexity* C of a logic program given a dataset is the number of choice-points that SLDNF resolution goes though in order to derive the data from the program. The *proof encoding length* L_{proof} of the hypothesis for each example can be approximated by combining the proof complexity with coverage results which estimate the compression achieved by the hypothesis and with the total (positive and negative) coverage, which is taken as a measure of the generality of the clause (idem., Section 3):

$$L_{proof} = (P+N) \cdot \left(C_{av} + \log \frac{1}{Acc^{Acc} \cdot (1-Acc)^{1-Acc}} \right) \tag{31}$$

where Acc is the observed accuracy $P/\left(P+N\right)$ and C_{av} the average proof complexity of the hypothesis over all the examples. It is easy to see that the logarithmic term (which represents the performance of the hypothesis) is dominated by the proof complexity term and the generality factor, so that it comes into consideration when comparing hypotheses with similar (semantic) generality and (syntactic) complexity characteristics.

3.7.4 Positive-Only Evaluation

One other category of evaluation functions that should be particularly noted is those that facilitate learning in the absence of negative examples (positive examples only). Conventional semantic evaluation functions cannot be used because in this case, in the absence of negative examples, the trivial clause that accepts all examples will always score best.

For this reason, positive-only evaluation functions balance positive coverage against a clause's generality, to avoid favouring over-generalization. Generality can be estimated syntactically as well as semantically, as in the *posonly* function (Muggleton, 1996):

$$\mathrm{PosOnly}_{C,R_{all}}(P,R,L) = \log P - C \log \frac{R+1.0}{R_{all}+2.0} - \frac{L}{P} \tag{32}$$

where $(R+1.0)/(R_{all}+2.0)$ is a Laplace-corrected estimation of the clause's generality. The estimation is made by randomly generating R_{all} examples and measuring the clause's coverage R over them. The formula is balancing between rewarding coverage (the first term) and penalizing generality (the second term), so as to avoid over-general clauses in the absence of negative data. The C parameter implements prior preference towards more general or more specific clauses and the third term implements syntactic bias towards shorter clauses, unless extended coverage compensates for the length penalty.

3.8 Other Approaches to ILP

We have so far focused ILP systems that perform a clause-level search on a pre-defined, static search space. Although these systems form the mainstream of ILP research, other options have been explored as well, namely searching at the theory level and searching in a dynamic space.

3.8.1 Theory-level Search

The sequential cover strategy has a profound impact on the size of the search space, as it restricts the search to the space of single clauses. It does so, however, at the risk of missing out on good solutions, since the the concatenation of good clauses is not necessarily a good theory: the best clause, given a set of positives, might leave a positives pool that is not easily separable, whereas a (locally) worse clause might allow for a better overall theory.

This is especially true for systems that saturate examples to build the bottom clause, as the (random) choice of the seed is decisive for the search sub-space that will be explored. An unfortunate seed choice might 'trap' subsequent clausal searches with an unnecessarily difficult positives pool. In sequential-cover systems the problem can be somewhat alleviated by saturating multiple seeds; in the ALEPH system, to name one example, the user can specify the number of examples that must be saturated for each positives pool. After performing as many searches as there were bottom clauses constructed, the best clause is appended to the theory and the process re-iterates.

Such techniques can improve the quality of the constructed theories, but not completely solve the problem, which can only be tackled by evaluating the theory as a whole at each step for the search. In *theory-level searching* the search space is the lattice formed by the set all admissible *theories* (as opposed to clauses), structured into a lattice by a theory-level traversal operator. It is immediately obvious that not only does search space become dramatically larger, but the traversal operator also gains more degrees of indeterminism, making the search even harder: the traversal operator can (a) specify or generalize a clause, (b) delete a clause altogether, or (c) start a new (top or bottom) clause.

In order to handle the increase in the size and complexity of the search space ILP systems that support theory-level search, like ALEPH, exploit *randomized search methods* to improve efficiency. Randomized search methods and other efficiency optimizations used in ILP systems are further described below.

3.8.2 Dynamically Changing Search Spaces

We end the section with an overview of algorithms that tackle dynamically changing search spaces, due to bias shift and background theory revision (background predicate invention and refinement).

Descriptive ILP systems readily offer themselves to predicate invention, since they are oriented towards discovering 'property clusters' in the data and multi-predicate learning. Systems like CLAUDIEN (De Raedt & Dehaspe, 1996) propose predicates that separate such clusters, and also re-use these predicates in the definitions of subsequently constructed predicate clauses.

Single-predicate learning, predictive ILP systems, on the other hand, typically assumed that the background predicates are both correct and sufficient to solve the problem, and no attempt is made to revise or supplement them. Some systems, however, do attempt background theory revision. SPECTRE (Boström, 1996), for example, attempts background theory refinement by examining the

SLD proof-trees generated during hypothesis matching. When pruning a background predicate's proof-tree (effectively, specializing the predicate) eliminates negative coverage, the background predicate is amended accordingly.

The same idea was further explored in the MERLIN2.0 system (Boström, 1998), this time attempting *predicate invention*, which expands the background theory with new predicates deemed useful for constructing a theory. MERLIN2.0 unfolds the SLD proof-trees of background predicates, and identifies new sub-predicates (in the semantic sense of predicates achieving a subset of the original predicate's coverage) which improve accuracy when used to construct a theory.

Inverse resolution also offers an opportunity for predicate invention: Muggleton (1988) proposes a predicate invention algorithm which asserts a predicate when its presence allows an—otherwise unattainable—inverse resolution step to proceed, if the generalization performed by this step evaluates well on the data. This idea was implemented in CIGOL (Muggleton & Buntine, 1988), an inverse-resolution ILP system, but was later abandoned as predictive ILP research moved towards inverse entailment systems that use θ-subsumption as their traversal operator.

3.8.3 Statistical Inductive Logic Programming

A new line of ILP research that is being actively pursued combines statistical Machine Learning with ILP. One such approach (Muggleton, 2003) combines a symbolic ILP step with a numerical step to learn a Stochastic Logic Program (SLP). The structure of the SLP is learnt without taking the numerical parameters into account, which are subsequently estimated to fit the program.

Another approach abandons the learning semantics discussed earlier in this section, and introduces *learning from proofs*. Under learning from proofs, the input is the proof trees of positive and negative derivations, which are used (in conjunction with a background theory) to build a statistical model of 'good' or 'applicable' derivations using. De Raedt et al. (2005) use a variant of Expectation Maximisation and Passerini et al. (2006) kernel-based methods to build the statistical model.

4 EFFICIENT INDUCTIVE LOGIC PROGRAMMING

A crucial point for the applicability of ILP is its efficiency in terms of computational resources needed to construct a theory. As has been experimentally verified Železný et al. (2003), ILP systems' run-times exhibit considerable variability, depending on the heuristics, the problem instance, and the choice of seed examples.

The total execution time of an ILP system can be roughly divided into three major components:

- time spent *generating* clauses, which can be a major concern for large real-world problems, and particularly in novel application domains where prior domain knowledge is fragmentary and the search space cannot be restricted without risking to exclude good solutions;

- time spent *evaluating* clauses, typically due to the size of the data, although Botta et al. (2003) have shown that there are circumstances under which evaluation can become extremely hard, even if model-spaces and datasets are quite small; and

- time spent *accessing* the data, as some datasets can be extremely large, e.g., biological databases arising from sequencing animal and plant genomes. Constructing models of such data requires an ILP system to be able to efficiently access millions of data items.

The relative weight of each component depends on the ILP system used, on the search algorithm, on the parameter settings, and on the data, but in most cases evaluating rule quality is responsible for most of the execution time.

In this section we describe methods for improving the efficiency of ILP systems directly related to the three concerns raised above (large search spaces, large datasets, evaluation). Some of these methods are improvements and optimizations of the sequential execution of ILP systems, whereas some are stochastic approaches to search or parallelisms of the search strategy.

Before proceeding, we provide a classification of the techniques regarding their correctness. In this context, a *correct technique* should be understood as yielding the same results as some reference algorithm that does not employ the technique. An *approximately correct technique* does not preserve correctness, but still gives results that are, with high probability, similar (in quality) to the results produced by the reference technique.

4.1 Reducing the Search Space

In the Hypothesis Language section above, we have discussed a variety of tools that ILP systems offer for controlling the search space. More specifically, we have discussed how the search space is initially defined by the vocabulary provided by the background theory and then further refined by language bias such as hypothesis checking and pruning, determinacy constrains, and type-mode declarations.

In the theoretic context of our previous discussion of these tools, they have been presented as a means of excluding potentially good solutions on grounds that the ILP algorithm cannot access through example-driven evaluation, like non-conformance with a theoretical framework. We shall here revisit the same tools in order to review them from a different perspective: instead of a means of excluding empirically good solutions that should be avoided on theoretical grounds, they are viewed as an optimization that excludes bad solutions before having to evaluate them in order to reject them.

4.1.1 Hypothesis Checking and Pruning

We have seen how hypothesis checking can be used to impose syntactic as well as as semantic constraints on the hypothesis language. In many situations it is possible to capitalize on the monotonicity of definite clause logic in order to direct the search away not from individual clauses, but from whole areas of the search space that are known to not contain any solutions.

Consider, for example, a clause containing in its body literals that are known to be inconsistent with each other, say $\mathrm{male}(X)$ and $\mathrm{female}(X)$. Not only is such a clause bound to not cover any examples, but all its specializations can also safely ignored. In systems that support *pruning*, prior knowledge can be provided which, once such a clause in encountered during the search, prunes away the whole sub-space subsumed by the clause.

Another method that takes advantage of prior expert knowledge to reduce the hypothesis language is *redundancy declarations*. Fonseca et al. (2004) propose a classification of redundancy in the hypothesis language and show how expert knowledge can be provided to an ILP system to in order reduce it. The technique is correct (if complete search is performed) and yields good results.

Two things must be noted at this point: first that useful as they might be, semantic hypothesis checking and pruning are bound to make a smaller difference with respect to efficiency that their syntactic counterparts since they require the—potentially expensive—coverage computations to be carried out whereas purely syntactic checking can discard a hypothesis beforehand. And, second, that the hypothesis checking, pruning, and redundancy declarations must be provided by the domain expert. This task that can be tedious and error-prone and has not been successfully automated.

4.1.2 Determinacy and Type-mode Declarations

Determinacy and type-mode declarations can also be used to state not a theoretical restriction, but an actual fact about the background theory. When used in this manner they are not excluding empirically possible—but otherwise unacceptable—solutions, but they are rather steering the search away from areas of the search that are known to be infertile.

Consider, for example, the declarations in Eq. 16 above. While the type-mode declarations for the $\mathrm{grandfather}/2$ predicate are, as we have already discussed, enforcing a genuine restriction, the rest are stating actual facts about the semantics of the background predicates. So, for instance, the $\mathrm{parent}/2$ declaration is prohibiting the consideration of hypotheses like this one:

$$\begin{aligned} C = \quad & \mathrm{grandfather}(X,U) \leftarrow \mathrm{father}(X,Y) \wedge \mathrm{parent}(Y,U) \wedge \mathrm{male}(U) \wedge \\ & \mathrm{parent}(Y,V) \wedge \mathrm{male}(V) \wedge \mathrm{parent}(Y,W) \wedge \mathrm{male}(W) \end{aligned} \tag{33}$$

which only admits grand-fatherhood in the presence of three or more grand-sons. This clause is rejected because the background theory does not allow instantiations where three children are male. The clause can only succeed by unifying two of U, V, W and can be safely ignored: no ILP algorithm would have chosen this clause anyway, as the model of our example identical semantics can be achieved by a simpler clause.

In general, providing the smallest determinacy values that will not leave any solutions out of the hypothesis space is a correct optimization of the ILP search. Similarly for type declarations, where assigning incompatible types can help avoid evaluating obviously inconsistent clauses like, for example:

$$D = \text{grandfather}(X, Y) \leftarrow femaleX \wedge \text{father}(X, Z) \wedge \text{parent}(Z, Y) \tag{34}$$

Prior knowledge of small, but safe, determinacy and type-mode parameters can significantly tighten the search space around the solutions. These semantic characteristics of the background theory are typically expected as user input, but research on their automatically extraction from the background has also been pursued (McCreath & Sharma, 1995).

4.1.3 Incremental Search

Another approach is to not restrict the hypothesis language, but to incrementally consider larger and larger subsets thereof. The whole hypothesis language will only be considered if we cannot find a model within one of these subsets. Srinivasan et al. (2003) propose a technique that explores human expertise to provide a relevance ordering on the set of background predicates. The technique follows a strategy of incrementally including sets of background knowledge predicates in decreasing order of relevance and has shown to yield good results.

Another technique that incrementally increases the hypothesis space is Incremental Language Level Search (ILLS) (Camacho, 2002). It uses an iterative search strategy to, starting from one, progressively increase the upper-bound on the number of occurrences of a predicate symbol in the generated hypotheses. This technique is correct if a complete search is performed and Camacho report substantial efficiency improvements on several ILP applications.

4.2 Efficient Evaluation

As described above, hypotheses are evaluated by metrics that make reference to their coverage over the training data. Coverage is calculated by matching (or testing) all examples against the hypothesis; this involves finding a substitution such that the body of one of the hypothesis' clauses is true given the example and background knowledge.

An often used approach in ILP to match hypotheses is to use logical querying. The logical querying approach to clause matching involves the use of a Prolog engine to evaluate the clause with the examples and background knowledge. The execution time to evaluate a query depends on the number of examples, number of resolution steps, and on the execution time of individual literals. Thus, scalability problems may arise when dealing with a great number of examples or/and when the computational cost to evaluate a rule is high. Query execution using SLDNF resolution grows exponentially with the number of literals in the query (Struyf, 2004). Hence, evaluating a single example can take a long time.

Several techniques have been proposed to improve the efficiency of hypotheses evaluation. Firstly, the evaluation of a hypothesis can be optimized by transforming the hypothesis into an equivalent one that can be more efficiently executed (Santos Costa et al., 2000; Santos Costa, Srinivasan, et al., 2003). An important characteristic of these techniques is that the transformations between equivalent hypothesis are correct. The transformation can be done at the level of individual hypotheses (Santos Costa et al., 2000; Struyf & Blockeel, 2003) or at the level of sets of hypotheses (Tsur et al., 1998; Blockeel et al., 2002).

A different method to speedup evaluation explores the existing redundancy on the sets of hypotheses evaluated: similar hypotheses are generated and evaluated in batches, called *query packs* (Blockeel et al., 2002), thus avoiding the redundant repetition of the same proof steps in different, but similar, hypotheses.

Approximate evaluation is another well-studied way of reducing the execution time of hypothesis evaluation. Stochastic matching (Sebag & Rouveirol, 1997), or stochastic theorem proving, was tested with the PROGOL ILP system, and has yielded considerable efficiency improvements, without sacrificing predictive accuracy or comprehensibility. This approach was further pursued by Giordana et al. (2000), making the benefits of replacing deterministic matching with stochastic matching clearly visible. A variety of other approximate matching schemes (Srinivasan, 1999; Kijsirikul et al., 2001; DiMaio & Shavlik, 2004; Bockhorst & Ong, 2004) have also been successfully tried.

Another technique, called *lazy evaluation*,[13] of examples (Camacho, 2003) aims at speeding up the evaluation of hypotheses by avoiding the unnecessary use of examples in the coverage computations, yielding considerable reduction of the execution time. The rationale underlying lazy evaluation is the following: a hypothesis is allowed to cover a small number of negative examples (the *noise* level) or none. If a clause covers more than the allowed number of negative examples it must be specialized. *Lazy evaluation of negatives* can be used when we are interested in knowing if a hypothesis covers more than the allowed number of negative examples or not. Testing stops as soon as the number of negative examples covered exceeds the allowed noise level or when there are no more negative examples to be tested. Therefore, the number of negative examples effectively tested may be very small, since the noise level is quite often very close to zero. If the evaluation function used does not involve negative coverage in its calculations, then this produces exactly the same results (clauses and accuracy) as the non-lazy approach but with a reduction on the number of negative examples tested.

One may also allow positive coverage to be computed lazily (*lazy evaluation of positives*). A clause is either specialized (if it covers more positives than the best consistent clause found so far) or justifiably pruned away otherwise. When using lazy evaluation of positives it is only relevant to determine if a hypothesis covers more positives than the current best consistent hypothesis or not. We might then just evaluate the positive examples until we exceed the best cover so far. If the best cover is exceeded we retain the hypothesis (either accept it as final if it is consistent or refine it otherwise) or we may justifiably discard it. We need to evaluate its exact positive cover only when accepting a consistent hypothesis. In this latter case we don't need to restart the positive coverage computation from scratch, we may simply continue the test from the point where we left it before.

Storing intermediate results during the evaluation for later use (i.e., by performing a kind of a cache) can be a solution to reduce the time spent in hypothesis evaluation. The techniques that follow this method are categorized as: improving the evaluation of the literals of a hypothesis (Rocha et al., 2005), or by reducing the number of examples tested (Cussens, 1996; Berardi et al., 2004). All these techniques are correct and attempt to improve time efficiency at the cost of increasing memory consumption.

4.3 Handling Large Datasets

The explanation semantics used to formulate the task at hand has a profound influence on how data is represented, stored, and manipulated and, subsequently, a great impact on the performance of an ILP system. Under learning from entailment, examples may relate to each other, so they cannot be handled independently. Therefore, there is no separation of either the examples (apart from being positive or negative) or the background knowledge.

Learning from interpretations, on the other hand, assumes that that the data exhibits a certain amount of locality and each example is represented as sub-database, i.e., a separate Prolog program, encoding its specific properties. This allows ILP techniques that operate under this semantics to scale up well (Blockeel et al., 1999). Naturally, this assumption makes learning from interpretations weaker than learning from entailment, but applications and purposes for which this semantics is sufficient, benefit from its inherent scalability.

Learning from subsets of data is another way of dealing with large datasets. For instance, *windowing* is a well known technique that learns from subsets of data. It tries to identify a subset of the original data from which a theory of sufficient quality can be learnt. It as been shown (Quinlan, 1993; Fürnkranz, 1998) that this technique increases the predictive accuracy and reduces the learning time. In ILP, studies shown that windowing reduces the execution time while preserving the quality of the models found (Srinivasan, 1999; Fürnkranz, 1997).

4.4 Search Algorithms

The hypothesis space determines the set of possible hypotheses that can be considered while searching for a good hypothesis. Several approaches to reduce the hypothesis space were described above. On the other hand, the search algorithm defines the order by which the hypotheses are considered and determines the search space (i.e., the hypotheses effectively considered during the search). The search algorithm used can have a great impact on efficiency, however, it can also have an impact on the quality of the hypothesis found.

A wide number of search techniques have been used in ILP systems, namely breadth-first search (in PROGOL), depth-first search (in ALEPH), beam-search (Džeroski, 1993; Srinivasan, 2004), heuristic-guided hill-climbing variants (Quinlan, 1990; Srinivasan, 2004), and simulated annealing (Srinivasan, 2004; Serrurier et al., 2004), just to mention a few. The choice of one in detriment of another has several effects (Russell & Norvig, 2003), namely on memory consumption, execution time, and completeness.

More advanced search techniques have been exploited in ILP systems. A genetic search algorithm was proposed in (Tamaddoni-Nezhad & Muggleton, 2000) but the impact on efficiency was not reported.

Probabilistically searching large hypothesis space (Srinivasan, 2000) restricts the search space by sacrificing optimality. It consists in randomly selecting a fixed-size sample of clauses from the search space which, with high probability, contains a good clause. The evaluation of the technique on three real world applications showed reductions in the execution time without significantly affecting the quality of the hypothesis found. However, this approach has difficulties with ‘needle in a haystack’ problems, where very few good hypotheses exist.

Randomized rapid restarts (Železný et al., 2003) combines (local) complete search with the probabilistic search. It performs an exhaustive search up to a certain point (time constrained) and then, if a solution is not found, restarts into randomly selected location of the search space, The application of the RRR technique in two applications yielded a drastic reduction of the search time at the cost of a small loss in predictive accuracy (Železný et al., 2003).

4.5 Parallelism

Parallelism provides an attractive solution for improving efficiency. ILP systems may profit from exploiting parallelism by decreasing learning time, handling larger datasets, and improving the quality of the induced models. The exploitation of parallelism introduces several challenges. Designing and validating a parallel algorithm is often harder than designing and validating sequential algorithms. Performance issues are complex: splitting work into too many tasks may introduce significant overheads, whereas using fewer tasks may result in load imbalance and bad speedups.

There are three main strategies to exploit parallelism in ILP systems are (Fonseca et al., 2005): parallel exploration of the search space (Dehaspe & De Raedt, 1995; Ohwada et al., 2000; Ohwada & Mizoguchi, 1999; Wielemaker, 2003); parallel hypothesis evaluation (Matsui et al., 1998; Konstantopoulos, 2003); and parallel execution of an ILP system over a partition of the data (Ohwada & Mizoguchi, 1999; Graham et al., 2003). A survey on exploiting parallelism in ILP is presented in (Fonseca et al., 2005). An evaluation of several parallel ILP algorithms showed that a good approach to parallelize ILP systems is one of the simplest to implement: divide the set of examples by the computers/processors available; run the ILP system in parallel on each subset; in the end, combine the theories found into a single one (Fonseca et al., 2005; Fonseca, 2006). This approach not only reduced the execution time but also improved predictive accuracy.

5 APPLICATIONS

In this section we briefly outline and provide references to various real-world applications of ILP, from medicine and biology, to language technology where ILP systems have made significant contributions to the discovery of new scientific knowledge.

Although we discuss in more detail applications in the three main areas of Life Sciences, Language Processing, and Engineering, ILP has been used in a variety of other domains. Examples of important applications of ILP must, at the very least, include domains such as music (Pompe et al., 1996; Tobudic & Widmer, 2003), the environment (Džeroski et al., 1995; Blockeel et al., 2004), intelligence analysis (Davis, Dutra, et al., 2005), and mathematical discovery (Colton & Muggleton, 2003; Todorovski et al., 2004).

5.1 Life Sciences

Inductive Logic Programming has been applied in a wide variety of domains of the Life Sciences, ranging over domains as diverse as Medical Support Systems (Carrault et al., 2003) and Computational Biochemistry (Srinivasan et al., 1994; Page & Craven, 2003).

Clinical data is one of the major sources of challenging datasets for ILP. To cite but a few example applications we will mention successful work on the characterization of cardiac arrhythmias (Quiniou et al., 2001), intensive care monitoring (Morik et al., 1999), diagnosis support systems for rheumatic diseases (Zupan & Džeroski, 1998) or breast cancer (Davis, Burnside, et al., 2005).

One of the major applications of Inductive Logic Programming so far has been in the area of Structure-Activity Relationships (SAR), the task of predicting the activity of drug molecules based on their structure. Early examples of this work include detecting mutagenic (Srinivasan et al., 1994) and carcinogenic (Srinivasan et al., 1997) properties in compounds. In the continuation, researchers have used ILP for 3D-SAR, where one uses a 3D description of the main elements in the compound to find *pharmacophores* that explain drug activity (Finn et al., 1998). Among the successful examples of ILP applications one can mention pharmacophores for dopamine agonists, ACE inhibitors, Thermolysin inhibitors, and antibacterial peptides (Enot & King, 2003). A related application with a different approach is the work in Diterpene structure elucidation by Džeroski et al. (1996).

ILP has also made significant contributions to the expanding area of Bio-informatics and Computational Biology. One major interest in this area has been in explaining protein structure, including secondary structure (Muggleton, King, & Sternberg, 1992; Mozetic, 1998) and fold prediction (Turcotte et al., 2001). Another very exciting area of ILP research is helping understand cell machinery; the work of Bryant et al. (2001) should be mentioned as the seminal work on understanding metabolic pathways of yeast. Finally, in genetics, recent work has also achieved promising results on using ILP to predict the functional class of genes (King, 2004) and to understand the combination of gene expression data with structure and/or function gene data (Struyf et al., 2005).

Recent studies on automating the scientific discovery process King et al. (2004) embed ILP systems in the cycle of scientific experimentation. The cycle includes the automatic generation of hypotheses to explain observations, the devising of experiments to evaluate the hypotheses, physically run the experiments using a laboratory robot, interprets the results to falsify hypotheses inconsistent with the data and then repeats the cycle.

5.2 Language Technology

The ability to take advantage of explicit background knowledge and bias constitutes one of the strongest points in favour of applying ILP to language technology tasks, as it provides a means of directly exploiting a very long and rich tradition of theoretical linguistics research.

More specifically, syntactic bias allows for the restriction of the hypothesis space within the limits of a meta-theory or theoretical framework. Except for the theoretical merit it carries on its own, this ability also offers the opportunity to capitalize on linguistic knowledge in order to reduce the computational cost of searching the hypothesis space, whereas many alternative learning schemes cannot make such explicit use of existing knowledge.

Language technology experimentation with ILP runs through the whole range of fields of the study of language, from phonology and morphology all the way to syntax, semantics and discourse analysis: we will mention work on prediction of past tenses (Muggleton & Bain, 1999), nominal paradigms (Džeroski & Erjavec, 1997), part-of-speech tagging (Dehaspe & De Raedt, 1997; Cussens, 1997), learning transfer rules (Boström, 2000), and focus on parsing and phonotactics (below).

Another exciting application of ILP is in the area of Information Extraction (IE). This includes work on document analysis and understanding (Esposito et al., 1993) and work on web page characterization that has had a profound practical impact (Craven & Slattery, 2001). Interesting results have been achieved when using ILP for performing IE over scientific literature (Califf & Mooney, 1999; Goadrich et al., 2004; Malerba et al., 2003).

5.2.1 Parser Construction

The syntax of an utterance is the way in which words combine to form grammatical phrases and sentences and the way in which the semantics of the individual words combine to give rise to the semantics of phrases and sentences. It is, in other words, the structure hidden behind the (flat) utterance heard and seen on the surface.

Since the seminal work of Chomsky (1957), it is the fundamental assumption of linguistics that this structure has the form of a tree, where the terminal symbols are the actual word-forms and the non-terminal symbols abstractions of words or multi-word phrases. It is the task of a *computational grammar* to describe the mapping between syntactic trees structures and flat utterances. Computational grammars are typically Definite Clause Grammars,[14] which describe syntactic structures in terms of derivations. Such grammars are used to control a *parser*, the piece of computational machinery that builds the *parse tree* of the phrase from the derivations necessary to successfully recognise the phrase.

Definite Clause Grammars can be naturally represented as Definite Clause Programs and their derivations map directly to Definite Logic Programming proofs. This does not, however, mean that grammar construction can be trivially formulated as an ILP task: this is for various reasons, but most importantly because ILP algorithms will typically perform single-predicate learning without attempting background knowledge revision.

Because of these difficulties, the alternative of refining an incomplete original grammar has been explored. In this approach learning is broken down in an abductive example generation and an inductive example generalization step. In the first step, examples are generated from a linguistic corpus as follows: the incomplete parser is applied to the corpus and, each time a parse fails, the most specific missing piece of the proof tree is identified. In other words, for all sentences that fail to parse, a ground, phrase-specific rule is identified that—if incorporated in the grammar—would have allowed that phrase's parse to succeed.

For the inductive step the parsing mechanism and the original grammar are included in the background knowledge and an ILP algorithm generalizes the phrase-specific rule examples into general rules that are appended to the original grammar to yield a complete grammar of the language of the corpus.

This abductive-inductive methodology has been successfully applied to Definite Clause grammars of English in two different parsing frameworks: (Cussens & Pulman, 2000) uses it to to extend the coverage of a chart parser (Pereira & Warren, 1983) by learning chart-licensing rules and Zelle & Mooney (1996) to restrict the coverage of an overly general shift-reduce parser (Tomita, 1986) by learning control rules for the parser's shift and reduce operators.

5.2.2 Phonotactic Modelling

The *Phonotactics* of a given language is the set of rules that identifies what sequences of phonemes constitute a possible word in that language. The problem can be broken down to the *syllable structure* (i.e. what sequences of phonemes constitute a possible syllable) and the processes that take place at the syllable boundaries (e.g. assimilation). Phonotactic models assist in many Information Extraction tasks, like optical character recognition and speech recognition, as they can be applied to catch recognition errors or limit the space of possibilities that the main recognition algorithm has to consider.

Phonotactic modelling tasks have been tackled by various Machine Learning methodologies and in various languages. Just to mention learning the syllable structure of the Dutch language from the CELEX (Burnage, 1990) lexical corpus, we find symbolic methods, like abduction (Tjong Kim Sang & Nerbonne, 2000) and ILP (Nerbonne & Konstantopoulos, 2004), as well as stochastic and distributed models, like Hidden Markov Models (Tjong Kim Sang, 1998) and Simple Recurrent Networks (Stoianov et al., 1998).

Nerbonne & Konstantopoulos (2004) compare the results of all these approaches, and identify the relative advantages and disadvantages of ILP. In short, ILP and abduction achieve similar accuracy, but ILP constructs a first-order syllabic theory which is formulated in about one fifth of the number of rules used by abduction to formulate a propositional syllabic theory.

Furthermore, it should be noted that the background knowledge in these experiments is both language-neutral and unrelated to the problem at hand. In a second series of experiments, both ILP and abduction are fortified with a background theory that transfers results from phonology theory. ILP was given access to background encoding a hierarchical feature-class description of Dutch phonetic material (Booij, 1995), a theory which is language-specific but not related to the problem at hand. Abduction was biased toward a general theory of syllable structure (Cairns & Feinstein, 1982), which is language-neutral but does constitute a step towards solving the problem at hand. In these second series of experiments, the first-order ILP solution is expressed in one tenth of the clauses of the propositional one.[15] These results leave ample space for further experimentation, since the backgrounds provided to the two algorithms are not directly comparable, but do show a very clear tendency of ILP to take very good advantage of sophisticated background knowledge when such is provided.

5.3 Engineering

ILP has been applied to a wide variety of engineering applications. One classical application of ILP is in finite element mesh design for structure analysis (Dolšak et al., 1994,9). This is a difficult task that depends on body geometry, type of edges, boundary conditions and loading.

Camacho (1998, 2000) has used ILP to automatically construct an autopilot for a flight simulator. The approach used was based on a reverse engineering technique called *behavioural cloning*. The basic idea is to have a human expert pilot flying the flight simulator and collect state and control variables of the plane during the flight manoeuvres conducted by the human pilot. In a second step the collected traces of the flights are pre-processed and taken as examples for a Machine Learning (ILP in our case) algorithm. The learning algorithm produces a model of the human pilot for each of the flight manoeuvres. These models are put together and an auto-pilot is assembled.

Further examples include traffic control (Džeroski et al., 1998), spatial data mining (Popelinsky, 1998), and intrusion detection in computer networks (Gunetti & Ruffo, 1999; Ko, 2000)

6 CONCLUSIONS

In this chapter we have described Inductive Logic Programming and demonstrated how it builds upon Artificial Intelligence strategies for searching in lattices. ILP relies on the vast body of research on search strategies that has been developed within AI. On the other hand, ILP fortifies lattice searching with powerful tools for expressing prior knowledge to control the search space, which constitutes a significant theoretical and computational contribution. We argue, therefore, that ILP is of significant interest to AI's research on search.

ILP systems are very versatile tools, that have been adapted to an extensive domain of applications. As these applications expand to range over more sophisticated and complex domains, ILP has needed to tackle larger searcher spaces. Indeed, David & Srinivasan (2003) argue that improvements in search are one of the critical areas where ILP research should focus on. This chapter shows a number of recent works on improving search in ILP, through techniques such as stochastic search and parallelism. We expect that this research will continue and we believe that major contributions will be of interest to the whole AI community. In the end, ILP is not the silver bullet that will tackle all AI problems, but a sound understanding of its strengths and weaknesses can yield fruitful results in many domains.

6.1 The Limitations of ILP

ILP inherits from its logical foundations the difficulties of symbolic (as opposed to numerical) computation: it is computationally demanding and difficult to scale up, and does not handle numerical data well.

With respect to the computational demands of ILP, it should be noted that most often cost is dominated by hypothesis matching, which relies on performing unification and symbolic manipulation on a large scale. Although many important optimizations have been proposed, as discussed earlier, the elementary search step in ILP is very inefficient when compared to other machine learning and pattern recognition methodologies. On a more optimistic note, the inefficiency of the elementary search step is counter-balanced by the fact that fewer such steps need to be taken to reach a solution, as ILP is particularly good at capitalizing on prior domain knowledge to restrict the search space and perform a more focused search.

Scalability issues pertaining to the volume of the data that needs to be processed are also difficult to tackle, but considerable ground has been covered in that direction as well. First of all, learning from interpretations takes advantage of localities in the data and scales up well. But under the stronger model of learning from entailment, where no such data locality assumptions are made, stochastic evaluation and parallelization techniques allow ILP systems to handle large volumes of data, as discussed.

A second limitation is that handling numerical data is difficult and cumbersome in ILP, as is usually the case with symbolic manipulation systems. In fact, it is only very recently that hybrid modelling languages like Stochastic Logic Programming (Cussens, 2000) and Bayesian Logic Programming (Kersting & Raedt, 2001) combine first-order inference with probability calculus. Searching in this combined hypothesis language is a very promising and exciting new subdiscipline of ILP (Muggleton, 2003; Kersting & Raedt, 2002; Santos Costa, Page, et al., 2003).

6.2 The Argument for Prior Knowledge

The explicit control over the exact boundaries of the hypothesis language that ILP systems offer, constitutes one of the strongest advantages of ILP. This is of relevance to theoretical as well as computational aspects of Artificial Intelligence.

From a *theoretical* point of view, it provides a means of restricting the search space within the limits of a meta-theory or theoretical framework. Such a framework might be justified by theoretical grounds and not necessarily relate to (or be derivable from) the empirical evaluation of hypotheses, hence purely empirical machine learning algorithms will not necessarily propose conforming hypotheses.

From a *computational* point of view, it is generally accepted in the Computer Science literature and practice that one of the most important aspects of setting up a search task properly is tightly delineating the search space so that it contains the solution, but as little more than that as possible. Explicit control over the hypothesis language offers an opportunity to capitalize on prior domain knowledge in order to reduce the computational cost of searching the hypothesis space, by encoding known qualities that good hypotheses possess, so that the search for a solution focuses on clauses where these qualities are present.

This is, of course, not to argue that control over the feature set, constraint satisfaction and bias cannot be implemented in statistical or distributed-computation approaches to machine learning. But the qualitative difference that ILP makes, is the ability to express those in Definite Horn clauses, an explicit and symbolic formalism that is considered—for reasons independent from its being at the foundations of ILP—to be particularly suitable for knowledge representation.

6.3 The Versatility of ILP

We have seen how several different evaluation measures have been proposed that estimate the quality of a hypothesis, reviewed here and also by Lavrač et al. (1999) and Fürnkranz & Flach (2003). This diversity a consequence of the variety of learning tasks and applications that ILP has been applied to, and different evaluation methods have been designed for different kinds of problems.

One point that should made here is the flexibility that this wide range of evaluation functions gives to ILP. In particular the ability to take advantage of explicit negative examples should be noted, by allowing the distinction between explicit negative examples (propositions that should not be covered) and non-positives examples (propositions that are not in the set of proposition that should be covered, propositions that are not interesting.)

At the same time, the ability to exploit negatives does not imply a reliance on their existence or their reliability. Restrictions on maximum negative coverage can be relaxed to accommodate noisy domains and some evaluation functions, like the the m-probability estimate, also provide parameters for applying a preference bias towards stricter or more liberal clauses. Finally, in the total absence of explicit negatives, positive-only evaluation is employed. This last approach most closely resembles some regression-based machine learning disciplines in that they are both trying to guess a concept's boundaries by interpolating between positive data-points, rather than by looking for a separating line between the positive and the negative points.

6.4 Future Directions of ILP

As more and more data is stored in more powerful computers, ILP systems face a number of challenges. David & Srinivasan propose five areas as critical to the future of ILP: the ability to process and reason with uncertain information, leading to the use of probabilities; improvements in search; the ability to take best advantages of progress in computing power, and namely of parallel computing; and, last not least, improvements in user interaction that will facilitate using ILP across a wider community. This is but a simplification. Current research shows a number of other exciting directions such as progress in database interfacing, better usage of propositional learners, learning second order extensions such as aggregated functions, and even a recent revival of areas such as predicate invention. Most important, ILP is now being applied on larger, and more complex datasets than some would believe was possible a few years ago, with exciting results. Ultimately, we believe it is the fact that ILP is indeed useful for practical applications that will drive its growth, and that will guarantee its contribution to the wider Computer Science community.

APPENDIX A: Logic Programming

This chapter assumes that the reader is familiar with Logic Programming terminology. We provide here a very short and incomplete introduction to Logic Programming for ease of reference, but for a more complete treatment the reader is referred to Logic Programming textbooks like the one by Lloyd (1997).

A *term* of *arity* N is a *functor symbol* followed by an N-tuple of terms. A *variable* is also a term. Thus, f is a term of arity 0 and $\mathrm{g}(\mathrm{h}(\mathrm{a},\mathrm{b},\mathrm{c}),X)$ is a term of arity 2 with sub-terms $\mathrm{g}(\mathrm{a},\mathrm{b},\mathrm{c})$ (arity 3) and variable X. An atom is a *predicate symbol* of arity N followed by a N-tuple of terms. If A is an atom, then A is a *positive literal* and $\neg A$ is a *negative literal*.

A *clause* is a disjunction of literals. All the variables in a clause are implicitly universally quantified. The empty clause and the logical constant false are represented by $\Box$. A *Horn clause* is a disjunction of any number of negated literals and up to one non-negated literal, for example:

$$\begin{array}{l} h \vee \neg l_1 \vee \neg l_2 ... \vee \neg l_n \\ \neg l_1 \vee \neg l_2 ... \vee \neg l_n \\ h \end{array}$$

The positive literal is called the *head* of the clause and the negative literals (if any) are collectively known as the *body*. Horn clauses can be equivalently represented as sets of literals that are meant to be logically or'ed together or as Prolog clauses:

$$\begin{array}{lcl} \{h, \neg l_1, \neg l_2, \ldots \neg l_n\} & & \mathtt{h} \leftarrow \mathtt{l_1}, \mathtt{l_2}, \ldots \mathtt{l_n}. \\ \{\neg l_1, \neg l_2, \ldots \neg l_n\} & \Longleftrightarrow & \perp \leftarrow \mathtt{l_1}, \mathtt{l_2}, \ldots \mathtt{l_n}. \\ \{h\} & & \mathtt{h}. \end{array}$$

A *definite Horn clause* is a Horn clause with *exactly* one positive literal, for example:

$$\begin{array}{l} h \vee \neg l_1 \vee \neg l_2 ... \vee \neg l_n \\ h \end{array}$$

A *substitution* is a function that maps a set of variables to a set of terms or variables. We apply a substitution to a term by replacing all variables in the term with the terms or variables indicated by the mapping. We usually denote a substitution as a set of from/to pairs and we write $A\theta$ to denote the result of applying substitution θ to term A. For example, if $\theta = \{X/Z, Y/\mathrm{aureliano}\}$ and $A = \mathrm{parent}(X, Y)$ then $A\theta = \mathrm{parent}(Z, \mathrm{aureliano})$.

Related to substitution is the concept of *unification*. Unification is the operation of making two terms identical by substitution. Such substitutions are called *unifiers* of the terms. The *most general unifier* is the unifier which minimally instantiates the two terms.

A *predicate* is a disjunction of definite Horn clauses where (a) no pair of clauses shares a common variable and (b) the heads of the clauses are the same up to substitution, i.e. they are terms with the same functor and arity. A *program* is a conjunction of predicates where no pair of predicates shares a common variable. The empty program and the logical constant true are represented by $\blacksquare$.

The *Herbrand base* of a clause, predicate, or program is the set of all ground (variable-free) atoms composed from symbols found in the clause, predicate, or program. An *interpretation* is a total function from ground atoms to $\{\mathrm{true}, \mathrm{false}\}$. A *Herbrand interpretation* for program P is an interpretation for *all* the atomic symbols in the Herbrand base of P. The valuation of a non-atomic clauses, predicates, and programs can be derived from the definitions of the logical connectives (conjunction, disjunction, negation, implication, existential and universal quantification) used to construct them from atoms. The logical connectives are set-theoretically defined in the usual manner (intersection, union, complement, etc) which we shall not re-iterate here.

We shall use the term interpretation to mean Herbrand interpretation. An interpretation M for P is a *model* of P if and only if P is true under M. Every program P has a unique *minimal Herbrand model* $\mathcal{M}^+(P)$ such that every atom a in P is true under $\mathcal{M}^+(P)$ if and only if a is true under all models of P.

Let P and Q be programs. We say that P entails Q ($P \models Q$) if and only if every model of P is also a model of Q. A clause, predicate, or program is *satisfiable* if and only if there exists a model for it. Equivalently, P is satisfiable if and only if $P \not\models \Box$ and *unsatisfiable* if and only if $P \models \Box$.

A query is a conjunction of positive literals. Posing a query Q to a program P amounts to asking about the satisfiability of $P \models Q$; if the program $P \not\models Q$ is satisfiable then the query gets an affirmative answer. Computational algorithms for answering first-order logical queries rely on deductive methods which can infer semantic entailment by syntactic manipulation of the programs involved. Deductive inference that deduces D from C *only* when $C \models D$ (according to the set-theoretic, semantic definitions of the logical symbols and connectives) is called *sound* and deductive inference that deduces D from C for *all* C, D where $C \models D$ is called *complete. Tableaux methods* and *resolution* and sound and complete deductive methodologies.

Selection linear definite resolution (SLD resolution) is an algorithm for resolving define clause programs. SLD *negation-as-failure* resolution (SLDNF resolution) extends SLD resolution to normal clause programs, which admit negative literals as premises, but only under the *closed world assumption* (CWA). CWA is the assumption that statements that cannot be proved true, are not true. Prolog is a resolution engine that operates under the CWA to perform SLDNF resolution. Almost all ILP systems (and especially predictive ILP systems) rely on a Prolog implementation for inferring entailment relationships.

Notes

[1]To the best of our knowledge, Muggleton was the first to use the term as the title of his invited talk at the first conference on Algorithmic Learning Theory (Tokyo, 1990). It is noteworthy that Muggleton & Feng presented a research paper on the ILP system GOLEM at the same conference where the term does not appear at all. The term ILP was formally introduced by Muggleton one year later at his landmark publication with the New Generation Computing journal (Muggleton, 1991).

[2]Arguably, this definition may nowadays be somewhat narrow. The representation and learning strategies used in ILP are germane to work such as relational instance based learning (Emde & Wettschereck, 1996; Horváth et al., 2001) and Analogical Prediction (Muggleton & Bain, 1999). ILP is also a key influence in the development of statistical relational learning, that often combines logical and statistical modelling.

[3]Inferring entailment is necessary for validating a hypothesis under learning from entailment semantics. Similarly, learning under interpretations requires a procedure for inferring whether an example is a model of a hypothesis.

[4]Only a second-order background theory would be able to represent such restrictions, and handling such a background theory would be far beyond the scope of current ILP. The external mechanisms currently used, effectively amount to 'mildly' second-order expressivity, that allows for certain kinds of restrictions to be represented and extra-logically tested. It might, in theory, be possible to identify a logic which captures only and all of these restrictions and

thus incorporate them into the background theory. Such a line of research has not, however, been pursued by the ILP community which builds upon Logic Programming and first-order SLDNF resolution engines.

[5]Variable depth is defined as the shortest link path to a head variable in various of Muggleton's publications, including the original definition (Muggleton & Feng, 1990, p. 8) and the Journal of Logic Programming article about the theory of ILP (Muggleton & De Raedt, 1994, p. 657). There is one notable exception, however, as Muggleton (1995, p 11) defines depth to be the *longest* path to the variable in his PROGOL article. The authors tested current ILP implementations, and found them conforming to the original definition.

[6]Variables X and Y have depth 0; variables U and W have depth 1; variable V has depth 2.

[7]Strictly speaking, traversal operators map conjunctions of clauses to sets of conjunctions of clauses. However, we typically focus on a single clause and perceive this mapping as being applied to one of the clauses in the conjunction *in the presence* of a background.

[8]Z/W is forced in order to unify the `ancestor/2` literals, but then `parent(W,Y)` cannot unify with either `parent(Z,Y)` or `parent(W,Z)` unless we further substitute X/Z, which would render the heads ununifiable.

[9]Muggleton (1995, Section 6) offers a formal and complete exposition of this reduction of implication to a θ-subsumption search and its logical foundations.

[10]Not to be confused with what is usually called bottom in logic programming, namely the empty predicate. There is an analogy, however, in that in the sense used here, the bottom is also the 'most specific' clause.

[11]For $\theta_1 = \{X/A, Y/B, Z/E, W/G\}$ we get $A\theta \subseteq \bot_1$ and $B\theta \subseteq \bot_1$.

[12]'Pseudo' in the sense that a true posterior probability can only be derived from a true prior probability, whereas here we only have an empirical approximation of the true prior.

[13]The term is used in the sense of making the minimal computation to obtain useful information.

[14]Many grammatical formalisms of greater computational complexity, for example, Head Phrase-driven Structure Grammar (HPSG), have also been proposed in the literature and gained wide acceptance. It is, however, the general consensus that grammatical formalisms beyond DCGs are justified on grounds of linguistic-theoretical conformance and compression on the grammar and that definite-clause grammars are powerful enough to capture most of the actual linguistic phenomena.

[15]The propositional theory describes syllable structure with 674 prevocalic and 456 postvocalic clauses. The first-order theory comprises 13 and 93 clauses, resp. When no language-specific prior phonological knowledge is used, the exact numbers are: for the propositional theory 577 clauses for the prevocalic and 577 clauses for the postvocalic material, and for the first-order theory 145 and 36 clauses, resp.

References

Anthony, S., & Frisch, A. M. (1999, January). Cautious induction: An alternative to clause-at-a-time induction in inductive logic programming. *New Generation Computing*, *17*(1), 25–52.

Bayes, T. (1763). An essay towards solving a problem in the doctrine of chances. *Philosophical Transactions of the Royal Society of London*, *53*, 370–418.

Berardi, M., Varlaro, A., & Malerba, D. (2004). On the effect of caching in recursive theory learning. In *Proceedings of the 14th international conference on inductive logic programming* (pp. 44–62). Berlin: Springer-Verlag.

Blockeel, H., Dehaspe, L., Demoen, B., Janssens, G., Ramon, J., & Vandecasteele, H. (2002). Improving the efficiency of Inductive Logic Programming through the use of query packs. *Journal of Machine Learning Research*, *16*, 135–166.

Blockeel, H., & De Raedt, L. (1998). Top-down induction of first-order logical decision trees. *Artificial Intelligence*, *101*(1–2), 285–297.

Blockeel, H., Džeroski, S., Kompare, B., Kramer, S., Pfahringer, B., & Laer, W. V. (2004). Experiments in predicting biodegradability. *Applied Artificial Intelligence*, *18*(2), 157–181.

Blockeel, H., Raedt, L. D., Jacobs, N., & Demoen, B. (1999). Scaling up inductive logic programming by learning from interpretations. *Data Mining and Knowledge Discovery*, *3*(1), 59–93.

Bockhorst, J., & Ong, I. M. (2004). FOIL-D: Efficiently scaling FOIL for multi-relational data mining of large datasets. In *Proceedings of the 14th international conference on inductive logic programming* (pp. 63–79).

Booij, G. (1995). *The phonology of Dutch.* Oxford: Clarendon Press.

Boström, H. (1996). Theory-guided induction of logic programs by inference of regular languages. In *Proc. of the 13th international conference on machine learning* (pp. 46–53). San Francisco: Morgan Kaufmann.

Boström, H. (1998). Predicate invention and learning from positive examples only. In *Proceedings of the tenth european conference on machine learning* (pp. 226–237). Berlin: Springer Verlag.

Boström, H. (2000, June). Induction of recursive transfer rules. In J. Cussens & S. Džeroski (Eds.), *Lecture notes in computer science: Vol. 1925. Learning Language in Logic* (pp. 237–246). Berlin: Springer-Verlag.

Botta, M., Giordana, A., Saitta, L., & Sebag, M. (2003). Relational learning as search in a critical region. *Journal of Machine Learning Research*, *4*, 431–463.

Bryant, C., Muggleton, S., Oliver, S., Kell, D., Reiser, P., & King, R. (2001). Combining Inductive Logic programming, Active Learning and robotics to discover the function of genes. *Electronic Transactions on Artificial Intelligence*, *5*(B1), 1–36.

Burnage, G. (1990). *CELEX: A guide for users.*

Cairns, C., & Feinstein, M. (1982). Markedness and the theory of syllable structure. *Linguistic Inquiry*, *13*.

Califf, M. E., & Mooney, R. J. (1999, July). Relational learning of pattern-match rules for information extraction. In *Proceedings of the 17th national conference on artificial intelligence.* Orlando, FL.

Camacho, R. (1998). Inducing models of human control skills. In C. Nedellec & C. Rouveirol (Eds.), *Lecture notes in computer science: Vol. 1398. ECML* (pp. 107–118). Berlin: Springer-Verlag.

Camacho, R. (2000). *Inducing models of human control skills using machine learning algorithms*. Unpublished doctoral dissertation, Department of Electrical Engineering and Computation, Universidade do Porto.

Camacho, R. (2002). Improving the efficiency of ILP systems using an incremental language level search. In *Annual machine learning conference of Belgium and the Netherlands.*

Camacho, R. (2003). As lazy as it can be. In P. Doherty, B. Tassen, P. Ala-Siuru, & B. Mayoh (Eds.), *The eighth Scandinavian conference on Artificial Intelligence (SCAI '03), Bergen, Norway, November 2003* (pp. 47–58).

Carrault, G., Cordier, M., Quiniou, R., & Wang, F. (2003, July). Temporal abstraction and Inductive Logic Programming for arrhythmia recognition from electrocardiograms. *Artificial Intelligence in Medicine*, *28*(3), 231–63.

Cestnik, B. (1990). Estimating probabilities: A crucial task in machine learning. In *European conference on artificial intelligence* (pp. 147–149).

Chomsky, N. (1957). *Syntactic structures*. The Hague: Mouton. (first mention of trees and such)

Clark, P., & Niblett, T. (1989). The CN2 induction algorithm. *Machine Learning*, *3*(4), 261–83.

Cohen, W. W. (1994). Grammatically biased learning: Learning logic programs using an explicit antecedent description language. *Artificial Intelligence*, *68*, 303–366.

Colton, S., & Muggleton, S. (2003). ILP for mathematical discovery. In T. Horváth (Ed.), *Lecture notes in computer science: Vol. 2835. ILP* (pp. 93–111). Berlin: Springer-Verlag.

Craven, M., & Slattery, S. (2001, April). Relational learning with statistical predicate invention: Better models for hypertext. *Machine Learning*, *43*(1/2), 97–119.

Cussens, J. (1993). Bayes and pseudo-Bayes estimates of conditional probabilities and their reliability. In *Proceedings of the european conference on machine learning (ecml93)* (pp. 136–152). Berlin: Springer Verlag.

Cussens, J. (1996). *Part-of-speech disambiguation using ILP* (Tech. Rep. No. PRG-TR-25-96). Oxford University Computing Laboratory.

Cussens, J. (1997). Part-of-speech tagging using progol. In S. Džeroski & N. Lavrač (Eds.), *Lecture notes in artificial intelligence: Vol. 1297. Proceedings of the 7th International Workshop on Inductive Logic Programming* (pp. 93–108). Berlin: Springer-Verlag.

Cussens, J. (2000). Stochastic logic programs: Sampling, inference and applications. In C. Boutilier & M. Goldszmidt (Eds.), *Proceedings of the 16th annual conference on uncertainty in artificial intelligence.* Morgan Kaufmann.

Cussens, J., & Džeroski, S. (Eds.). (2000). *Learning language in logic.* Berlin: Springer-Verlag.

Cussens, J., & Pulman, S. (2000). Experiments in inductive chart parsing. In J. Cussens & S. Džeroski (Eds.), *Lecture notes in artificial intelligence: Vol. 1925. Learning Language in Logic.* Berlin: Springer-Verlag.

David, P. C., & Srinivasan, A. (2003, August). ILP: A short look back and a longer look forward. *Journal of Machine Learning Research*(4), 415–430.

Davis, J., Burnside, E. S., Dutra, I. d. C., David, P. C., & Santos Costa, V. (2005). Knowledge discovery from structured mammography reports using inductive logic programming. In *American medical informatics association 2005 annual symposium.*

Davis, J., Dutra, I. d. C., Page, C. D., & Santos Costa, V. (2005). Establishing identity equivalence in multi-relational domains. In *Proceedings of the 2005 International Conference on Intelligence Analysis.*

Dehaspe, L., & De Raedt, L. (1995). Parallel inductive logic programming. In *Proceedings of the MLnet familiarization workshop on statistics, machine learning and knowledge discovery in databases.*

Dehaspe, L., & De Raedt, L. (1996, July). DLAB: *a declarative language bias for concept learning and knowledge discovery engines* (Tech. Rep. No. CW 214). Leuven: Department of Computing Science, K.U.Leuven.

Dehaspe, L., & De Raedt, L. (1997). Mining association rules in multiple relations. In S. Džeroski & N. Lavrač (Eds.), *Lecture notes in artificial intelligence: Vol. 1297. Proceedings of the 7th International Workshop on Inductive Logic Programming* (pp. 125–132). Berlin: Springer-Verlag.

Dehaspe, L., & Toironen, H. (2000). Relational data mining. In (pp. 189–208). Berlin: Springer-Verlag.

De Raedt, L. (1997). Logical settings for concept-learning. *Artificial Intelligence*, *95*(1), 187–201.

De Raedt, L., & Blockeel, H. (1997). Using logical decision trees for clustering. In *Proceedings of the 7th international workshop on inductive logic programming* (pp. 133–140). Springer-Verlag.

De Raedt, L., & Dehaspe, L. (1996). *Clausal discovery* (Tech. Rep. No. CW 238). Leuven: Department of Computing Science, K.U.Leuven.

De Raedt, L., & Dehaspe, L. (1997). Learning from satisfiability. In *Proceedings of the ninth dutch conference on artificial intelligence (NAIC'97)* (pp. 303–312).

De Raedt, L., & Džeroski, S. (1994). First-order jk-clausal theories are pac-learnable. *Artificial Intelligence*, *70*(1-2), 375–392.

De Raedt, L., Kersting, K., & Torge, S. (2005). Towards learning stochastic logic programs from proof-banks. In *Proceedings of the 23th national conference on artificial intelligence, (AAAI 2005)* (pp. 752–757).

DiMaio, F., & Shavlik, J. W. (2004). Learning an approximation to inductive logic programming clause evaluation. In *Proceedings of the 14th international conference on inductive logic programming* (pp. 80–97).

Dolšak, B., Bratko, I., & Jezernik, A. (1994). Finite element mesh design: an engineering domain for ILP applications. In *Proc. of the 4th international workshop on inductive logic programming (ILP-94), Bad Honnef/Bonn, Germany, September 12–14.*

Dolšak, B., Bratko, I., & Jezernik, A. (1997). Application of Machine Learning in finite element computation. In R. Michalski, I. Bratko, & M. Kubat (Eds.), *Machine learning, data mining and knowledge discovery: Methods and applications.* John Wiley and Sons.

Džeroski, S. (1993). Handling imperfect data in inductive logic programming. In *Proceedings of the 4th scandinavian conference on artificial intelligence* (pp. 111–125). IOS Press.

Džeroski, S., Dehaspe, L., Ruck, B., & Walley, W. (1995). Classification of river water quality data using machine learning. In *Proceedings of the 5th international conference on the development and application of computer techniques to environmental studies.*

Džeroski, S., & Erjavec, T. (1997). Induction of Slovene nominal paradigms. In N. Lavrac & S. Džeroski (Eds.), *Lecture notes in computer science: Vol. 1297. ILP* (pp. 141–148). Berlin: Springer-Verlag.

Džeroski, S., Jacobs, N., Molina, M., & Moure, C. (1998). ILP experiments in detecting traffic problems. In C. Nedellec & C. Rouveirol (Eds.), *Lecture notes in computer science: Vol. 1398. ECML* (pp. 61–66). Berlin: Springer-Verlag.

Džeroski, S., Schulze-Kremer, S., Heidtke, K., Siems, K., & Wettschereck, D. (1996). Applying ILP to diterpene structure elucidation from ^{13}C NMR spectra. In *Proceedings of the MLnet Familiarization Workshop on Data Mining with Inductive Logic Programing* (pp. 12–24).

Emde, W., & Wettschereck, D. (1996). Relational instance-based learning. In L. Saitta (Ed.), *Proceedings of the 1996 international machine learning conference* (p. 122-130).

Enot, D. P., & King, R. D. (2003). Application of inductive logic programming to structure-based drug design. In N. Lavrac, D. Gamberger, H. Blockeel, & L. Todorovski (Eds.), *Lecture notes in computer science: Vol. 2838. Proceedings of the 7th European Conference on Principles of Data Mining and Knowledge Discovery* (pp. 156–167). Berlin: Springer-Verlag.

Esposito, F., Malerba, D., & Semeraro, G. (1993). Automated acquisition of rules for document understanding. In *Proceedings of the 2nd international conference on document analysis and recognition* (pp. 650–654).

Finn, P. W., Muggleton, S. H., Page, C. D., & Srinivasan, A. (1998). Pharmacophore discovery using the inductive logic programming system PROGOL. *Machine Learning*, *30*(2-3), 241-270.

Fonseca, N. A. (2006). *Parallelism in inductive logic programming systems*. Unpublished doctoral dissertation, University of Porto.

Fonseca, N. A., Santos Costa, V., Camacho, R., & Silva, F. (2004). On avoiding redundancy in Inductive Logic Programming. In R. Camacho, R. D. King, & A. Srinivasan (Eds.), *Lecture notes in artificial intelligence: Vol. 3194. Proceedings of the 14th International Conference on Inductive Logic Programming, Porto, Portugal, September 2004* (pp. 132–146). Berlin: Springer-Verlag.

Fonseca, N. A., Silva, F., Santos Costa, V., & Camacho, R. (2005). Strategies to paralilize ILP systems. In *Lecture notes in ai. 15th International Conference on Inductive Logic Programming (ILP 2005), Bonn, August 2005* (pp. 136–153). Berlin: Springer-Verlag. (Best paper ILP 2005)

Fürnkranz, J. (1997, August). Dimensionality reduction in ILP: a call to arms. In L. De Raedt & S. H. Muggleton (Eds.), *Proceedings of the IJCAI-97 workshop on frontiers of inductive logic programming* (pp. 81–86). Nagoya, Japan.

Fürnkranz, J. (1998). Integrative windowing. *Journal of Machine Learning Research*, *8*, 129–164.

Fürnkranz, J., & Flach, P. (2003). An analysis of rule evaluation metrics. In *Proceedings of the 20th international conference on machine learning (icml-03).* San Francisco: Morgan Kaufmann.

Giordana, A., Saitta, L., Sebag, M., & Botta, M. (2000). Analyzing relational learning in the phase transition framework. In *Proceedings of the 17th international conference on machine learning* (pp. 311–318). San Francisco: Morgan Kaufmann.

Goadrich, M., Oliphant, L., & Shavlik, J. W. (2004). Learning ensembles of first-order clauses for recall-precision curves: A case study in biomedical information extraction. In R. Camacho, R. D. King, & A. Srinivasan (Eds.), *Lecture notes in computer science: Vol. 3194. ILP* (pp. 98–115). Berlin: Springer-Verlag.

Graham, J., Page, C. D., & Kamal, A. (2003). Accelerating the drug design process through parallel inductive logic programming data mining. In *Proceeding of the computational systems bioinformatics (csb'03).* IEEE.

Gunetti, D., & Ruffo, G. (1999). Intrusion Detection through Behavioral Data. In S. Wrobel (Ed.), *Lecture notes in computer science. Proc. of The Third Symposium on Intelligent Data Analysis.* Berlin: Springer-Verlag.

Hayes-Roth, F. (1974). Schematic classification problems and their solution. *Pattern Recognition*, *6*(2), 105-113.

Horváth, T. (Ed.). (2003). *Inductive logic programming: Proceeding of the 13th international conference, ILP 2003, Szeged, Hungary, September 29–October 1, 2003.* Berlin: Springer-Verlag.

Horváth, T., Wrobel, S., & Bohnebeck, U. (2001, April). Relational instance-based learning with lists and terms. *Machine Learning*, *43*(1/2), 53–80.

Jorge, A., & Brazdil, P. (1995). Architecture for iterative learning of recursive definitions. In L. De Raedt (Ed.), *Proceedings of the 5th international workshop on inductive logic programming* (pp. 95–108). Department of Computer Science, Katholieke Universiteit Leuven.

Kersting, K., & Raedt, L. D. (2001, November). *Bayesian logic programs* (Tech. Rep. No. 151). Georges-Koehler-Allee, D-77110, Freiburg: Institute for Computer Science, University of Freiburg.

Kersting, K., & Raedt, L. D. (2002). *Basic principles of learning bayesian logic progras* (Tech. Rep. No. 174). Georges-Koehler-Allee, D-77110, Freiburg: Institute for Computer Science, University of Freiburg.

Kijsirikul, B., Sinthupinyo, S., & Chongkasemwongse, K. (2001). Approximate match of rules using backpropagation neural networks. *Machine Learning*, *44*(3), 273–299.

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

King, R. D., Whelan, K. E., Jones, F. M., Reiser, P. G., Bryant, C. H., Muggleton, S. H., et al. (2004, January). Functional genomic hypothesis generation and experimentation by a robot scientist. *Nature*, *427*(6971), 247–252.

Ko, C. (2000). Logic induction of valid behavior specifications for intrusion detection. In *SP 2000: Proceedings of the 2000 IEEE symposium on security and privacy.* Washington, DC, USA: IEEE Computer Society.

Konstantopoulos, S. (2003, September). A data-parallel version of Aleph. In *Proceedings of the workshop on parallel and distributed computing for Machine Learning, co-located with ECML/PKDD'2003.* Dubrovnik, Croatia.

Langley, P. (1996). *Elements of machine learning*. Morgan Kaufmann.

Lavrac, N., & Džeroski, S. (Eds.). (1997). *Proceedings of the 7th international workshop on inductive logic programming (ILP-97), Prague, Czech Republic, September 17-20, 1997.* Berlin: Springer-Verlag.

Lavrač, N., Flach, P., & Zupan, B. (1999, June). Rule evaluation measures: A unifying view. In S. Džeroski & P. Flach (Eds.), *Lecture notes in artificial intelligence: Vol. 1634. Proceedings of the 9th International Workshop on Inductive Logic Programming* (pp. 174–185). Berlin: Springer-Verlag.

Lloyd, J. W. (1997). *Foundations of logic programming* (2nd ed.). Berlin: Springer-Verlag.

Malerba, D., Esposito, F., Altamura, O., Ceci, M., & Berardi, M. (2003). Correcting the document layout: A machine learning approach. In *Icdar* (p. 97-). IEEE Computer Society.

Matsui, T., Inuzuka, N., Seki, H., & Itoh, H. (1998). Comparison of three parallel implementations of an induction algorithm. In *8th int. parallel computing workshop* (pp. 181–188). Singapore.

Matwin, S., & Sammut, C. (Eds.). (2003). *Proceedings of the 12th international workshop on inductive logic programming (ilp 2002).* Berlin: Springer Verlag.

McCreath, E., & Sharma, A. (1995, November). Extraction of meta-knowledge to restrict the hypothesis space for ILP systems. In X. Yao (Ed.), *Proceedings of the eighth australian joint conference on artificial intelligence* (pp. 75–82). World Scientific.

Michalski, R. S. (1973). Aqval/1-computer implementation of a variable-valued logic system vl1 and examples of its application to pattern recognition. In *Proceeding of the first international joint conference on pattern recognition* (p. 3-17).

Mitchell, T. M. (1978). *Version spaces: An approach to concept learning*. Unpublished doctoral dissertation, Stanford University.

Mitchell, T. M. (1997). *Machine learning* (Second ed.). McGraw Hill.

Mitchell, T. M., Utgoff, P., & Banerji, R. (1983). Learning by experimentation: Acquiring and refining problem-solving heuristics. In R. Michalski, J. Carbonnel, & T. Mitchell (Eds.), *Machine learning: An artificial intelligence approach.* Palo Alto, CA: Tioga.

Morik, K., Brockhausen, P., & Joachims, T. (1999). Combining statistical learning with a knowledge-based approach — a case study in intensive care monitoring. In I. Bratko & S. Džeroski (Eds.), *Icml* (pp. 268–277). San Francisco: Morgan Kaufmann.

Mozetic, I. (1998, December). Secondary structure prediction by inductive logic programming. In *Third meeting on the critical assessment of techniques for protein structure prediction, CASP3* (pp. 38–38). Asilomar, CA: CASP3 organizers.

Muggleton, S. H. (1988). A strategy for constructing new predicates in first order logic. In D. Sleeman (Ed.), *Proceedings of the 3rd european working session on learning* (pp. 123–130). Pitman.

Muggleton, S. H. (1991). Inductive logic programming. *New Generation Computing*, *8*(4), 295–317.

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

Muggleton, S. H. (1996). Learning from positive data. In *Proceedings of the sixth international workshop on inductive logic programming, lnai 1314* (pp. 358–376). Berlin: Springer-Verlag.

Muggleton, S. H. (2003). Learning structure and parameters of stochastic logic programs. In S. Matwin & C. Sammut (Eds.), *Lecture notes in artificial intelligence: Vol. 2583. Proceedings of the 12th International Workshop on Inductive Logic Programming (ILP 2002)* (pp. 198–206). Berlin: Springer Verlag.

Muggleton, S. H., & Bain, M. (1999). Analogical prediction. In S. Džeroski & P. A. Flach (Eds.), *Lecture notes in computer science: Vol. 1634. ILP* (pp. 234–244). Berlin: Springer-Verlag.

Muggleton, S. H., & Buntine, W. (1988). Machine invention of first-order predicates by inverting resolution. In *Proceedings of the 5th international conference on machine learning* (pp. 339–352). Kaufmann.

Muggleton, S. H., & De Raedt, L. (1994). Inductive Logic Programming: Theory and methods. *Journal of Logic Programming*, *19*(20), 629–679. (Updated version of technical report CW 178, May 1993, Department of Computing Science, K.U. Leuven)

Muggleton, S. H., & Feng, C. (1990). Efficient induction of logic programs. In *Proceedings of the 1st conference on algorithmic learning theory* (pp. 368–381). Tokyo: Ohmsha.

Muggleton, S. H., King, R. D., & Sternberg, M. J. E. (1992). Protein secondary structure prediction using logic. In S. H. Muggleton (Ed.), *Report ICOT TM-1182. Proceedings of the 2nd International Workshop on Inductive Logic Programming* (pp. 228–259).

Muggleton, S. H., Srinivasan, A., & Bain, M. (1992). Compression, significance and accuracy. In D. Sleeman & P. Edwards (Eds.), *Proceedings of the 9th international workshop on machine learning* (pp. 338–347). San Francisco: Morgan Kaufmann.

Nedellec, C., & Rouveirol, C. (Eds.). (1998). *Proceedings of the 10th european conference on machine learning (ECML 1998), Chemnitz, Germany, April 21-23.* Berlin: Springer-Verlag.

Nerbonne, J., & Konstantopoulos, S. (2004, May). Phonotactics in Inductive Logic Programming. In M. Klopotek, S. Wierzchon, & K. Trojanowski (Eds.), *Advances in soft computing. Intelligent Information Processing and Web Mining. Proceedings of the International IIS: IIPWM'04 Conference, Zakopane, Poland, May 17–20, 2004* (pp. 493–502). Berlin: Springer-Verlag.

Ohwada, H., & Mizoguchi, F. (1999). Parallel execution for speeding up inductive logic programming systems. In *Lecture notes in artificial intelligence. Proceedings of the 9th International Workshop on Inductive Logic Programming* (pp. 277–286). Springer-Verlag.

Ohwada, H., Nishiyama, H., & Mizoguchi, F. (2000). Concurrent execution of optimal hypothesis search for inverse entailment. In J. Cussens & A. Frisch (Eds.), *Lecture notes in artificial intelligence: Vol. 1866. Proceedings of the 10th International Conference on Inductive Logic Programming* (pp. 165–173). Berlin: Springer-Verlag.

Ong, I. M., Dutra, I. d. C., Page, C. D., & Santos Costa, V. (2005). Mode directed path finding. In *Lecture notes in computer science: Vol. 3720. Proceedings of the 16th European Conference on Machine Learning* (pp. 673–681).

Page, C. D., & Craven, M. (2003). Biological applications of multi-relational data mining. *SIGKDD Explor. Newsl.*, *5*(1), 69–79.

Passerini, A., Frasconi, P., & De Raedt, L. (2006). Kernels on prolog proof trees: Statistical learning in the ILP setting. In *Probabilistic, logical and relational learning. Towards a synthesis. 30 Jan - 4 Febr 2005* (pp. 307–342). Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany.

Pereira, F., & Warren, D. S. (1983). Parsing as deduction. In dunno (Ed.), *Proceedings of the 21st conference of the acl* (pp. 137–44).

Plotkin, G. D. (1970). A note on inductive generalization. In B. Meltzer & D. Michie (Eds.), *Machine intelligence* (Vol. 5, pp. 153–163). Edinburgh University Press.

Plotkin, G. D. (1971a). *Automatic methods of inductive inferrence*. Unpublished doctoral dissertation, University of Edinburgh.

Plotkin, G. D. (1971b). A further note on inductive generalization. In D. Michie, N. L. Collins, & E. Dale (Eds.), *Machine intelligence* (Vol. 6, pp. 101–124). Edinburgh University Press.

Pompe, U., Kononenko, I., & Makše, T. (1996). An application of ILP in a musical database: Learning to compose the two-voice counterpoint. In *Proceedings of the MLnet Familiarization Workshop on Data Mining with Inductive Logic Programing* (pp. 1–11).

Popelinsky, L. (1998, September). Knowledge discovery in spatial data by means of ILP. In J. M. Zytkow & M. Quafafou (Eds.), *Lecture notes in computer science: Vol. 1510. Principles of Data Mining and Knowledge Discovery. PKDD'98 Nantes France.* (pp. 271–279). Berlin: Springer Verlag.

Popplestone, R. J. (1970). An experiment in automatic induction. In *Machine Intelligence* (Vol. 5, p. 204-215). Edinburgh University Press.

Quiniou, R., Cordier, M.-O., Carrault, G., & Wang, F. (2001). Application of ILP to cardiac arrhythmia characterization for chronicle recognition. In C. Rouveirol & M. Sebag (Eds.), *Lecture notes in computer science: Vol. 2157. ILP* (pp. 220–227). Berlin: Springer-Verlag.

Quinlan, J. R. (1990). Learning logical definitions from relations. *Machine Learning*, *5*, 239–266.

Quinlan, J. R. (1993). *C4.5: programs for machine learning*. San Francisco: Morgan Kaufmann Publishers Inc.

Richards, B. L., & Mooney, R. J. (1995). Automated refinement of first-order horn-clause domain theories. *Machine Learning*, *19*(2), 95–131.

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

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

Rocha, R., Fonseca, N. A., & Costa, V. S. (2005). On Applying Tabling to Inductive Logic Programming. In *Lecture notes in artificial intelligence: Vol. 3720. Proceedings of the 16th*

European Conference on Machine Learning, ECML-05, Porto, Portugal, October 2005 (pp. 707–714). Berlin: Springer-Verlag.

Russell, S., & Norvig, P. (2003). *Artificial intelligence: A modern approach* (2nd edition ed.). Prentice-Hall, Englewood Cliffs, NJ.

Santos Costa, V., Page, C. D., Qazi, M., & Cussens, J. (2003, August). CLP($\mathcal{BN}$): Constraint Logic Programming for Probabilistic Knowledge. In *Proceedings of the 19th conference on uncertainty in artificial intelligence (UAI03), Acapulco, Mexico* (pp. 517–524).

Santos Costa, V., Srinivasan, A., & Camacho, R. (2000). A note on two simple transformations for improving the efficiency of an ILP system. In *Proceedings of the 10th international conference on inductive logic programming* (pp. 225–242).

Santos Costa, V., Srinivasan, A., Camacho, R., Blockeel, H., Demoen, B., Janssens, G., et al. (2003). Query transformations for improving the efficiency of ILP systems. *Journal of Machine Learning Research*, *4*, 465–491.

Sebag, M., & Rouveirol, C. (1997). Tractable induction and classification in first order logic via stochastic matching. In *Proceedings of the 15th international joint conference on artificial intelligence* (pp. 888–893). San Francisco: Morgan Kaufmann.

Serrurier, M., Prade, H., & Richard, G. (2004). A simulated annealing framework for ILP. In *Proceedings of the 14th international conference on inductive logic programming* (pp. 288–304).

Shannon, C. E. (1948, July, October). The mathematical theory of communication. *Bell Systems Technical Journal*, *27*, 379–423, 623–656.

Shapiro, E. (1983). *Algorithmic program debugging*. The MIT Press.

Srinivasan, A. (1999). A study of two sampling methods for analysing large datasets with ILP. *Data Mining and Knowledge Discovery*, *3*(1), 95–123.

Srinivasan, A. (2000). *A study of two probabilistic methods for searching large spaces with ILP* (Tech. Rep. No. PRG-TR-16-00). Oxford University Computing Laboratory.

Srinivasan, A. (2004). *The Aleph manual.* (Available at `http://www.comlab.ox.ac.uk/oucl/research/areas/machlearn/Aleph/`)

Srinivasan, A., King, R. D., & Bain, M. E. (2003). An empirical study of the use of relevance information in Inductive Logic Programming. *Journal of Machine Learning Research.*

Srinivasan, A., King, R. D., Muggleton, S., & Sternberg, M. J. E. (1997). Carcinogenesis predictions using ILP. In N. Lavrac & S. Džeroski (Eds.), *Lecture notes in computer science: Vol. 1297. Inductive Logic Programming* (pp. 273–287). Berlin: Springer-Verlag.

Srinivasan, A., Muggleton, S., King, R. D., & Sternberg, M. J. E. (1994). ILP experiments in a non-determinate biological domain. In S. Wrobel (Ed.), *Gmd-studien. Proceedings of the Fourth International ILP Workshop* (pp. 217–232). Gesellschaft für Mathematik und Datenverarbeitung MBH.

Stoianov, I., Nerbonne, J., & Bouma, H. (1998). Modelling the phonotactic structure of natural language words with Simple Recurrent Networks. In *Proceedings of the 8th conference on computational linguistics in the netherlands (CLIN 98).*

Stolle, C., Karwath, A., & De Raedt, L. (2005). CLASSIC'CL: an integrated ILP system. In *Lecture notes in artificial intelligence: Vol. 3735. Proceedings of the 8th International Conference of Discovery Science, DS 2005* (pp. 354–362). Berlin: Springer-Verlag.

Struyf, J. (2004). *Techniques for improving the efficiency of inductive logic programming in the context of data mining*. Unpublished doctoral dissertation, Katholieke Universiteit Leuven Department of Computer Science.

Struyf, J., & Blockeel, H. (2003). Query optimization in inductive logic programming by reordering literals. In *Proceedings of the 13th international conference on inductive logic programming* (pp. 329–346).

Struyf, J., Džeroski, S., Blockeel, H., & Clare, A. (2005). Hierarchical multi-classification with predictive clustering trees in functional genomics. In C. Bento, A. Cardoso, & G. Dias (Eds.), *Lecture notes in computer science: Vol. 3808. Progress in Artificial Intelligence, 12th Portuguese Conference on Artificial Intelligence, EPIA 2005, Covilhã, Portugal, December 5-8, 2005, Proceedings* (pp. 272–283). Berlin: Springer-Verlag.

Tamaddoni-Nezhad, A., & Muggleton, S. (2000). Searching the subsumption lattice by a genetic algorithm. In J. Cussens & A. Frisch (Eds.), *Lecture notes in artificial intelligence: Vol. 1866. Proceedings of the 10th International Conference on Inductive Logic Programming* (pp. 243–252). Springer-Verlag.

Tamaddoni-Nezhad, A., & Muggleton, S. (2003). A genetic algorithm approach to ILP. In S. Matwin & C. Sammut (Eds.), *Lecture notes in artificial intelligence: Vol. 2583. Proceedings of the 12th International Conference on Inductive Logic Programming* (pp. 285–300). Berlin: Springer-Verlag.

Tang, L. R., & Mooney, R. J. (2001). Using multiple clause constructors in inductive logic programming for semantic parsing. In *Lecture notes in computer science: Vol. 2167. Proceedings of the 12th European Conference on Machine Learning (ECML-2001).* Berlin: Springer Verlag.

Tjong Kim Sang, E. F. (1998). Machine learning of phonotactics (Doctoral dissertation, Rijksuniversiteit Groningen). *Groningen Dissertations in Linguistics Series.*

Tjong Kim Sang, E. F., & Nerbonne, J. (2000). Learning the logic of simple phonotactics. In J. Cussens & S. Džeroski (Eds.), *Lecture notes in artificial intelligence: Vol. 1925. Learning Language in Logic.* Berlin: Springer-Verlag.

Tobudic, A., & Widmer, G. (2003). Relational ibl in music with a new structural similarity measure. In T. Horváth (Ed.), *Lecture notes in computer science: Vol. 2835. ILP* (pp. 365–382). Berlin: Springer-Verlag.

Todorovski, L., Ljubic, P., & Džeroski, S. (2004). Inducing polynomial equations for regression. In J.-F. Boulicaut, F. Esposito, F. Giannotti, & D. Pedreschi (Eds.), *Lecture notes in computer science: Vol. 3201. Proceedings of the 15th European Conference on Machine Learning* (pp. 441–452). Berlin: Springer-Verlag.

Tomita, M. (1986). *Efficient parsing for natural language.* Boston: Kluwer Academic Publishers.

Tsur, D., Ullman, J. D., Abiteboul, S., Clifton, C., Motwani, R., Nestorov, S., et al. (1998). Query flocks: a generalization of association-rule mining. In *SIGMOD '98: Proceedings of the 1998 ACM SIGMOD international conference on management of data* (pp. 1–12). New York, NY, USA: ACM Press.

Turcotte, M., Muggleton, S. H., & Sternberg, M. J. E. (2001, April). The effect of relational background knowledge on learning of protein three-dimensional fold signatures. *Machine Learning*, *43*(1/2), 81–95.

Vere, S. A. (1975). Induction of concepts in the predicate calculus. In *Proceedings of the 4th international joint conference on artificial intelligence* (p. 281-287).

Železný, F., Srinivasan, A., & Page, C. D. (2003). Lattice-search runtime distributions may be heavy-tailed. In S. Matwin & C. Sammut (Eds.), *Lecture notes in artificial intelligence: Vol. 2583. Proceedings of the 12th International Workshop on Inductive Logic Programming (ILP 2002)* (pp. 333–345). Berlin: Springer Verlag.

Wielemaker, J. (2003). Native preemptive threads in SWI-Prolog. In C. Palamidessi (Ed.), *Lecture notes in artificial intelligence: Vol. 2916. Proceedings of the 19th International Conference on Logic Programming* (pp. 331–345). Springer-Verlag.

Winston, P. H. (1970). *Learning structural descriptions from examples*. Unpublished doctoral dissertation, MIT.

Wrobel, S. (1997). An algorithm for multi-relational discovery of subgroups. In *Proceedings of the first european symposium on principles of data mining and knowledge discovery* (pp. 78–87). Springer-Verlag.

Zelle, J. M., & Mooney, R. J. (1996). Learning to parse database queries using Inductive Logic Programming. In dunno (Ed.), *Proceedings of the 13th national conference on artificial intelligence, Portland, USA.*

Zupan, B., & Džeroski, S. (1998). Acquiring background knowledge for machine learning using function decomposition: a case study in rheumatology. *Artificial Intelligence in Medicine*, *14*(1–2), 101-117.

Index