Polynomial Learnability and Inductive Logic Programming: Methods and Results

William W. Cohen
AT&T Bell Laboratories
600 Mountain Avenue
Murray Hill, NJ 07974
wcohen@research.att.com

C. David Page Jr.
Oxford University Computing Lab
Parks Road
Oxford, OX1 3QD
dpage@prg.ox.ac.uk

Abstract

Over the last few years, the efficient learnability of logic programs has been studied extensively. Positive and negative learnability results now exist for a number of restricted classes of logic programs that are closely related to the classes used in practice within inductive logic programming. This paper surveys these results, and also introduces some of the more useful techniques for deriving such results. The paper does not assume any prior background in computational learning theory.

1 Introduction

As noted by Stephen Muggleton in a recent invited talk on inductive logic programming (ILP) [Muggleton, 1994a, Muggleton, 1994b], "ILP is based on the lock-step development of Theory, Implementations, and Applications." The theory of ILP involves not only logical foundations, but also computational foundations. At the heart of the computational foundations of learning is analysis of the trade-offs between the accuracy of the hypotheses of the learner, and the computational resources required by the learner. This trade-off is captured by formal computational models of inductive learning, as studied in *computational learning theory*.

The most realistic of these models are the "polynomial learnability" models, the best-known of which is Valiant's [1984] model of *pac-learnability*. Polynomial learnability models require that learners produce accurate hypotheses using resources polynomial in the complexity of the concept to be learned: here "resources" includes both the number of examples required by the learner, and the time required by the learning system run.

Over the last few years, the polynomial learnability of various restricted classes of logic programs, as used in practical ILP systems, has been thoroughly studied. The goal of this paper is to survey and make accessible these results. We focus on giving the reader the basic intuitions behind the results, at the expense (in some cases) of rigorous technical detail. The presentation does not assume prior knowledge of computational learning theory, although we do assume that the reader has a solid knowledge of logic programming and some familiarity with complexity theory.

We begin by motivating and defining two common models of learnability in Section 2. We then present a series of positive and negative learnability results. In Section 3 we describe the principle proof methods used for negative results, illustrating these techniques primarily on the problem of learning nondeterministic clauses.

In Section 4 we present some of the proof methods used for obtaining positive results, and also survey some of the strongest-known positive learnability results. Section 4 is independent of Section 3 and may be read first if desired.

The paper concludes by summarizing known results in the area, and drawing some general conclusions about future research directions within ILP.

2 Models of polynomial learnability

2.1 Standard models of learning

2.1.1 Preliminaries

To undertake any formal analysis, it is necessary to first define the objects that are being studied. In machine learning generally, the goal is often to construct a *classifier:* a function that assigns to an *instance* a label from some fixed small set. Thus if X is some set of possible instances (called a *domain*) and $class_1, \ldots, class_k$ are the possible classes, a classifier C is a function

$$C : X \rightarrow \{class_1, \ldots, class_k\}$$

For simplicity, it is useful to assume that there are only two labels, "+" and "−". In this case C can be viewed as denoting a subset of the domain X—namely, the subset of instances to which the label "+" is assigned. It is also useful to assume that associated with X is some measure of the *size* or *complexity* of an instance; typically this measure will be closely related to the length in bits of some encoding of the instance. We also assume that there is some language $\mathcal{L}$ for denoting concepts C, and assume that there is a size measure for concepts that is closely related to the length of the smallest expression in $\mathcal{L}$ that denotes C. In this paper the size of a concept $C \in \mathcal{L}$ is denoted $\|C\|$, and the size of an instance e is denoted $\|e\|$. (Note that we usually not distinguish between the representation of a concept and the set denoted by this representation; typically it will be clear from context which we intend.)

As an illustration, consider the problem of learning atomic formulae (atoms) from ground atoms. Here the domain X would be the set of ground atoms. The language $\mathcal{L}$ would be the usual notation of first-order logic for atoms (including those that contain variables), and a concept C would contain an atom a (*i.e.*, label it with a "+" label) if a was a ground instance of C.

2.1.2 Learning from equivalence queries

For any single classifier C, there is some learner that quickly learns C—namely, the learner that always hypothesizes C regardless of the data. Such a highly-specific learning system, however, is unlikely to be useful in practice; thus most formalizations of a "learning problem" include some notion of a *concept class*, which is simply a set of possible classifiers $\mathcal{C} = \{C_1, \ldots, C_i, \ldots\}$. Learning systems are then required to perform well for all concepts in a class. Specifically, most formal models assume that learning proceeds as follows:

1. Some process picks a *target concept* $C_t \in \mathcal{C}$.
2. The learner receives some information about C_t. For example, it may see some set of instances together with the labels assigned to these instances by C_t.
3. The learner produces some hypothesis H. If learning is successful, H will be produced quickly, and will either closely approximate or be equivalent to C_t.

To completely define a formal model of learning, one must precisely specify the three steps above. One common formal model of learning is *learning by equivalence queries* [Angluin, 1988]. In this model

1. The target concept $C_t \in \mathcal{C}$ is chosen by an adversary.
2. The learner receives information about C_t by asking *equivalence queries*. An *equivalence query* is a concept $C_q \in \mathcal{C}$, which is sent to an oracle that replies as follows:
   - If C_q is equivalent to the target, the oracle answers "equivalent".
   - Otherwise, the oracle returns as a *counterexample* some instance e that C_q and C_t classify differently. (Note that the label given to e by C_t can be easily derived, as C_t's label for e is simply the opposite of C_q's.)

3. Learning is judged "successful" if the final hypothesis is equivalent to C_t, and if the time and number of queries used by the learner is polynomial in the size of C_t and the total size of the counterexamples provided by the oracle.

More formally, a language $\mathcal{L}$ is defined to be *learnable from equivalence queries* iff there exists a deterministic algorithm LEARN and a polynomial $p(\cdot,\cdot)$ such that for all $C_t \in \mathcal{L}$ whenever LEARN is run (with an oracle answering equivalence queries for C_t) it eventually halts and outputs some $H \in \mathcal{L}$ such that H is equivalent to C_t, and if at every point in the execution of LEARN the total running time is bounded by $p(n_t, n_e)$, where n_t is the size of C_t, and n_e is the size of the largest counterexample seen so far (and $n_e = 0$ if no equivalence queries have been made). The polynomial $p(n_t, n_e)$ is called the *query complexity* of LEARN.

2.1.3 Pac-learning

While the model of learnability from equivalence queries is simple and easy to understand, the requirement that the learner converge to a hypothesis exactly equivalent to the target concept is quite strong; in practise, it is often sufficient to find a classifier that approximates the target concept. A common formal model of learnability that allows such an approximation is Valiant's model of *pac-learnability* [Valiant, 1984]. In this model:

1. The target concept $C_t \in \mathcal{C}$ is chosen by an adversary.

2. The information received by the learner about C_t consists of two sets S^+, S^- drawn stochastically from the domain X according to some probability distribution D, where S^+ contains only positive examples of C_t, and S^- contains only negative examples. The distribution D is also chosen by an adversary.

3. Informally, learning is judged "successful" if for any sufficiently large sample S^+, S^-, the hypothesis H of the learner will with high probability closely approximate C_t on instances chosen according to D—that is, if the hypothesis is probably approximately correct.

More formally, let X_n (respectively $\mathcal{L}_n$) to stand for the set of all elements of the domain X (respectively the language $\mathcal{L}$) of size no greater than n. If D is a probability distribution function, a *sample of C_t from X_n drawn according to D* is a pair of multisets S^+, S^- drawn from X_n according to D, S^+ containing only positive examples of C_t, and S^- containing only negative ones. A language $\mathcal{L}$ is *pac-learnable* iff there is an algorithm LEARN and a polynomial function $m(\frac{1}{\epsilon}, \frac{1}{\delta}, n_e, n_t)$ so that

for every $n_t > 0$,
  for every $n_e > 0$,
    for every $C_t \in \mathcal{L}_{n_t}$,
      for every $\epsilon : 0 < \epsilon < 1$,
        for every $\delta : 0 < \delta < 1$,
          for every probability distribution function D,
            if S^+, S^- is a sample of C_t from X_{n_e} drawn according to D
              that contains at least $m(\frac{1}{\epsilon}, \frac{1}{\delta}, n_e, n_t)$ examples, then

1. LEARN(S^+,S^-,ϵ,δ) outputs a hypothesis H such that
   $$Prob(D(H-C)+D(C-H)>\epsilon)<\delta$$
2. LEARN runs in time polynomial in $\frac{1}{\epsilon}$, $\frac{1}{\delta}$, n_e, n_t, and the number of examples in S^+ and S^-, and
3. The hypothesis H of the learner is in $\mathcal{L}$.

The probability in (1) is taken over the possible samples S^+ and S^-, and (if the learner is a randomized algorithm) the coin flips made by the learner.

The function $m(\frac{1}{\epsilon}, \frac{1}{\delta}, n_e, n_t)$ is called the *sample complexity* of the learning algorithm LEARN. Notice that the sample complexity depends on several parameters: $\frac{1}{\epsilon}$, where ϵ is the intended error rate of the hypothesis; $\frac{1}{\delta}$, where δ is the confidence that the hypothesis will have error less than ϵ; n_e, the size of the examples, and n_t, the size of the target concept C_t.

2.1.4 Discussion of the standard models

Equivalence-query learners and pac-learners use different protocols for obtaining information about C_t, and hence appear to be quite different. However, it turns out that they are closely related: whenever a language $\mathcal{L}$ is learnable from equivalence queries, then $\mathcal{L}$ is also pac-learnable [Angluin, 1988, Angluin, 1989]. The basic idea behind this fact is that if the goal of learning is a probably approximately correct hypothesis, an equivalence query for the hypothesis H can be emulated by drawing a set of random examples and seeing if that set contains a counterexample.

Both of these learning models have been well-studied, and are fairly well-understood. One reason that they are attractive is that pac-learning algorithms often appear to be "reasonable" learning algorithms. In most cases, for instance, an algorithm will pac-learn $\mathcal{L}$ if it efficiently finds the smallest concept in $\mathcal{L}$ that is consistent with a given set of examples. In fact, in most cases a hypothesis that is sublinear in the size of the examples and no more than polynomially larger than the target C_t suffices for pac-learning [Blumer *et al.*, 1989]. (Such learning algorithms are often called "Occam" algorithms.) A converse to this statement also holds [Helmbold and Warmuth, 1992]; hence there is a close connection between being able to pac-learn a concept and being able to perform data compression on a set of examples.

One simple "Occam" algorithm is to enumerate the concepts in $\mathcal{L}$ in increasing order of complexity and output the first hypothesis consistent with the examples. Although there are exceptions [Minton, 1994, Riddle *et al.*, 1994], such brute-force learning systems are typically not practical for reasons of efficiency. In the pac-learning setting, enumerative learning methods usually satisfy all the requirements of pac-learning *except* for the requirement of efficiency.

2.2 Background knowledge

These learning models are quite appropriate to modelling standard inductive learners. However, the typical ILP system is used in a somewhat more complex setting, as the user will typically provide both a set of examples and a *background theory* B: the task of the learner is then to find a logic program P such that P, together with B, is a good model of the data.

Intuitively, the way to handle this complication is revise the model of how learning is conducted as follows:

1. Some process picks the background knowledge B.
2. Some process picks *target concept* C_t, which is assumed to be encoded using B.
3. The learner receives B, and some information about C_t.
4. The learner produces a hypothesis H.

One can formalize this process (in the context of ILP problems) as follows. If $\mathcal{L}$ is some language of logic programs and B is a logic program, then let $\mathcal{L}[B]$ denote the set of all logic programs of the form $C \wedge B$ such that $C \in \mathcal{L}$. Thus "choosing a target concept C_t encoded using B" becomes simply choosing a concept in $\mathcal{L}[B]$.

In all of the results described in this paper, it will be assumed that the choice of a background theory B is made from a set $\mathcal{B}$ by an adversary. This model is made reasonable by the fact that the learning system can use resources polynomial in the complexity of C_t as encoded in $\mathcal{L}[B]$; hence if the adversary chooses a "useless" background theory, the size of C_t will be large or infinite, and the learner is not obligated to produce an accurate hypothesis quickly.

This leads to a simple definition of learning with background knowledge. If $\mathcal{B}$ is a set of background theories, then define the *language family* $\mathcal{L}[\mathcal{B}]$ to be the set of all languages $\mathcal{L}[B]$ where $B \in \mathcal{B}$. We define the language family $\mathcal{L}[\mathcal{B}]$ to be learnable from equivalence queries (respectively pac-learnable) when every $\mathcal{L}[B] \in \mathcal{L}[\mathcal{B}]$ is learnable from equivalence queries (pac-learnable.) This definition is more meaningful if one makes a slight modification to the definitions of pac-learnability and learning from equivalence queries: specifically, if one assumes a complexity measure for background theories, and if one allows the sample and time complexities of a learner to also grow (polynomially) with the complexity of the background theory provided by the user.

Usually we are interested in the case in which a single algorithm $\textsc{Learn}(B, S^+, S^-)$ pac-learns an approximation to $C_t \in \mathcal{L}[B]$ given any $B \in \mathcal{B}$. If a language family is pac-learnable by such an algorithm we will say that the language family is *uniformly pac-learnable*.

Discussion. There are other ways in which background knowledge could be formalized; for example, one alternative would be to directly extend the definitions of pac-learning and learning from equivalence queries to include a notion of background knowledge. The notion of learnability of language families, however, is quite convenient for technical reasons. The principal advantage is that the results about pac-learnability with background knowledge can be immediately translated into results about the pac-learnability in the standard model, without background knowledge. This makes it easier to apply previous pac-learnability results.

3 Proof techniques for negative learnability results

Many of the most interesting formal results for ILP problems are negative. In this section we will survey the principal proof techniques that are used to establish negative pac-learnability results, and illustrate these techniques by presenting some representative results on the non-learnability of ILP languages.

We will concentrate on negative pac-learnability results, as showing that a language $\mathcal{L}$ is not pac-learnable immediately implies that $\mathcal{L}$ is not learnable from equivalence queries. This is true because pac-learnability is the weaker of the two models.

3.1 Consistency hardness of constant-depth clauses

One common proof technique is to show that the *consistency problem* for a language is intractable. Informally, the consistency problem for $\mathcal{L}$ is to find a hypothesis in $\mathcal{L}$ that is consistent a given set of examples. Formally, the consistency problem for a language $\mathcal{L}$ can be stated as follows:

Given: a set of positive examples S^+ and a set of negative examples S^- labelled according to some target concept $C_t \in \mathcal{L}$,

Find: a concept $H \in \mathcal{L}$ that is consistent with the examples.

Pitt and Valiant [Pitt and Valiant, 1988] showed that if a language $\mathcal{L}$ is pac-learnable, then the consistency problem for $\mathcal{L}$ must be solvable in time polynomial in the size of C_t, S^+, and S^-. Thus one way to show that $\mathcal{L}$ is not pac-learnable is to show that the consistency problem for $\mathcal{L}$ is intractable.

It is straightforward to extend the Pitt and Valiant argument to pac-learning with background knowledge. The *consistency problem with background knowledge B* for a language $\mathcal{L}[B]$ can be defined as follows:

Given: a background theory $B \in \mathcal{B}$, a set of positive examples S^+, and a set of negative examples S^- labelled according to some target concept $C_t \in \mathcal{L}[B]$,

Find: a concept $H \in \mathcal{L}[B]$ that is consistent with the examples.

If there is some polynomial-sized $B \in \mathcal{B}$ such that the consistency problem with background knowledge B is intractable, then the language family $\mathcal{L}[\mathcal{B}]$ is not pac-learnable.

As an early example of this technique in an ILP context, Haussler [1989] showed that the consistency problem for "existential conjunctive concepts" is NP-hard. This proof was later adapted by Kietz to a somewhat more familiar language: the language of constant-depth nonrecursive definite clauses without function symbols. By way of introduction to this proof technique we will summarize Kietz's result.

We begin by defining formally the language considered by Kietz. If $A \leftarrow B_1 \wedge \ldots \wedge B_r$ is an (ordered) definite clause, then the *input variables* of the literal B_i are those variables appearing in B_i which also appear in the clause $A \leftarrow B_1 \wedge \ldots \wedge B_{i-1}$; all other variables appearing in B_i are called *output variables*. The *depth* of variables appearing in a clause can now be defined inductively as follows: variables appearing in the head of a clause have depth zero, and the depth of any output variable V that first appears in the literal B_i is $d+1$, where d is the maximal depth of any input variable of B_i. The depth of a clause is the maximal depth of any variable in the clause.

Let $\mathcal{L}^{k\text{-DEPTH}}[B]$ be the language of depth-k logic programs consisting of a single nonrecursive function free clause (plus the background theory B). Let a-$\mathcal{B}$ be the set of background theories B containing only ground facts of arity a or less. Kietz showed the following:

Theorem 1 (Kietz) *There exists a $B \in 2$-$\mathcal{B}$ such that the consistency problem with background knowledge B for $\mathcal{L}^{1\text{-DEPTH}}[B]$ is NP-hard.*

As with any NP-hardness result, the proof begins with a problem known to be NP-hard (in this case 3-SAT) and reduces it to the consistency problem for $\mathcal{L}^{1\text{-DEPTH}}[B]$, for a particular background theory B. For the sake of completeness, we give a sketch of the proof below.

Proof (sketch): Consider a 3-CNF clause $\phi = \bigwedge_{i=1}^{n}(l_{i_1} \vee l_{i_2} \vee l_{i_3})$ over the boolean variables $x_1, \ldots, x_n$. (Thus each of the l_{i_j}'s above is either x_k or $\overline{x_k}$ for some k.) Determining whether ϕ has any satisfying assignments is NP-hard. Now consider the following examples and background theory.

Positive examples. Let $ALL\_LITS(c)$ be defined as

$$\{true_1(c), \ldots, true_n(c), false_1(c), false_n(c)\}$$

For $k = 1, \ldots, n$, $target(e_k)$ is a positive example in S^+, and the background theory B_ϕ contains the facts

- $q(e_k, a_k)$, and all the facts in the set $ALL\_LITS(a_k) - \{true_k(a_k)\}$;
- $q(e_k, b_k)$, and all the facts in the set $ALL\_LITS(b_k) - \{false_k(b_k)\}$.

A clause covering all the positive example must have a head of the form *target(X)*. However, no clause covering the positive example $target(e_k)$ can contain all of the literals $q(X, Y), true_k(Y), false_k(Y)$ in its body. Thus in any clause H that covers all the positive examples and has the form $target(X) \leftarrow q(X, Y), \ldots$, the variable Y can be interpreted as a (partial) assignment to the variables $x_1, \ldots, x_n$. In particular, let η_Y be the assignment created by letting $x_k = 1$ if $true_k(Y)$ appears in H, and letting $x_k = 0$ if $false_k(Y)$ appears, and giving x_k an arbitrary value (say 0) if neither appears. If H covers all of the positive examples, then η_Y must be well-defined.

Negative examples. Let $LIT(l, c)$ be defined as

$$LIT(l, c) \equiv \begin{cases} true_k(c) & \text{if } l = x_k \\ false_k(c) & \text{if } l = \overline{x_k} \end{cases}$$

S^- contains the single negative example $target(e_*)$, and for $i = 1, \ldots, n$ the background theory B_ϕ contains the fact $q(e_*, c_i)$ and also all of the facts

$$ALL\_LITS(c_i) - \{LIT(l_{i_1}, c_i), LIT(l_{i_2}, c_i), LIT(l_{i_3}, c_i)\}$$

Thus to exclude the negative example the hypothesis H must be of the form

$$
\begin{array}{ll}
\textit{target(X)} \leftarrow & \\
\quad q(X,Y) \wedge & \\
\quad LIT(l_{1_{j1}}, Y) \wedge & \text{\% where } l_{1_{j1}} \text{ is } l_{1_1}, l_{1_2}, \text{ or } l_{1_3} \\
\quad LIT(l_{2_{j2}}, Y) \wedge & \text{\% where } l_{2_{j2}} \text{ is } l_{2_1}, l_{2_2}, \text{ or } l_{2_3} \\
\quad \vdots & \\
\quad LIT(l_{n_{jn}}, Y) \wedge & \text{\% where } l_{n_{jn}} \text{ is } l_{n_1}, l_{n_2}, \text{ or } l_{n_3} \\
\quad \vdots &
\end{array}
$$

It can be easily argued that the assignment η_Y corresponding to such a clause is a satisfying assignment for ϕ. Thus if a consistent clause can be found, a satisfying assignment for ϕ can be immediately constucted.

For example, consider the formula ϕ_0 below (over the variables x_1, x_2, x_3):

$$\phi_0 = (\overline{x_1} \vee \overline{x_2} \vee \overline{x_3}) \wedge (\overline{x_1} \vee \overline{x_2} \vee x_3) \wedge (\overline{x_1} \vee x_2 \vee \overline{x_3})$$

For this formula, the sample would be $S^+ = \{target(e_1), target(e_2), target(e_3)\}$ and $S^- = \{target(e_*)\}$, and the background theory B would contain following facts (abbreviating $true_i$ as t_i and $false_i$ as f_i):

$q(e_1, a_1)$		$t_2(a_1)$	$t_3(a_3)$	$f_1(a_1)$	$f_2(a_1)$	$f_3(a_1)$
$q(e_1, b_1)$	$t_1(b_1)$	$t_2(b_1)$	$t_3(b_3)$		$f_2(b_1)$	$f_3(b_1)$
$q(e_1, a_2)$	$t_1(a_2)$		$t_3(a_3)$	$f_1(a_2)$	$f_2(a_2)$	$f_3(a_2)$
$q(e_1, b_2)$	$t_1(b_2)$	$t_2(b_2)$	$t_3(b_3)$	$f_1(b_2)$		$f_3(b_2)$
$q(e_1, a_3)$	$t_1(a_3)$	$t_2(a_3)$		$f_1(a_3)$	$f_2(a_3)$	$f_3(a_3)$
$q(e_1, b_3)$	$t_1(b_3)$	$t_2(b_3)$	$t_3(b_3)$	$f_1(b_3)$	$f_2(b_3)$	
$q(e_*, c_1)$	$t_1(c_1)$	$t_2(c_1)$	$t_3(c_3)$			
$q(e_*, c_2)$	$t_2(c_2)$	$t_2(c_2)$				$f_3(c_2)$
$q(e_*, c_3)$	$t_1(c_3)$		$t_3(c_3)$		$f_2(c_3)$	

In this case $x_1 = x_2 = x_3 = 0$ is a satisfying assignment to ϕ_0, and the corresponding clause

$$target(X) \leftarrow q(X,Y), f_1(Y), f_2(Y), f_3(Y)$$

covers all the positive examples but not the negative example. ■

Due to the argument of Pitt and Valiant [Pitt and Valiant, 1988] the following is an immediate corollary:

Corollary 1 *For $k \geq 1$ and $a \geq 2$, the language family $\mathcal{L}^{k\text{-DEPTH}}[a\text{-}\mathcal{B}]$ is not pac-learnable.*

Discussion. Theorem 1 shows that it may be expensive to find a hypothesis that is consistent with a given set of data, *when the hypothesis is constrained to be in the language of single non-recursive depth-bounded function-free clauses.* The main weakness of this result—and indeed of every negative result based on consistency-hardness—is that it is heavily dependent on the requirement that the learning system outputs a hypothesis in the target language, in this case the language $\mathcal{L}^{1\text{-DEPTH}}[B]$. However, few practical ILP systems restrict their hypotheses to a single clause—instead most ILP systems hypothesize a set of clauses. Thus the result does not give any information about the computational complexity of real ILP systems. For example, FOIL [Quinlan, 1990] hypothesizes a set of clauses that together are consistent with the positive and negative examples; it is entirely possible (from Theorem 1) that FOIL's hypotheses are generated in polynomial time and are probably approximately correct for target concepts in $\mathcal{L}^{1\text{-DEPTH}}[B]$.

In general, a non-learnability result for a language $\mathcal{L}$ that is based on consistency-hardness does not preclude the existence of a more expressive language $\mathcal{L}'$ that is pac-learnable. As an example of this in an ILP context, Page and Frisch [Page and Frisch, 1992] show that the language of typed atoms is not pac-learnable; however, the more general language of constrained atoms is pac-learnable. To summarize, while a consistency-hardness result for a language $\mathcal{L}$ shows that $\mathcal{L}$ is hard to pac-learn, it gives no indication of how "fix" $\mathcal{L}$—pac-learnability may be achieved by either restricting or by generalizing $\mathcal{L}$.

3.2 Evaluation hardness

A proof technique that gives a much stronger negative result than consistency-hardness is to show that a language contains concepts that cannot be *evaluated* in polynomial time. Schapire [1990] shows that a language $\mathcal{L}$ cannot be pac-learnable (under certain complexity-theoretic assumptions) if it contains any concept C that cannot be evaluated in polynomial time—*i.e.*, if there is a $C \in \mathcal{L}$ such that testing to see if $e \in C$ cannot be performed in time polynomial in the size of e and C. Although the proof that evaluation-hardness implies non-learnability is deep, the result is fairly intuitive: indeed, it is difficult to imagine how an efficient learning algorithm could be implemented if it is impossible to check the consistency of a hypothesis with a single example efficiently.

It can be shown that the language of depth-1 nonrecursive clauses is also hard to evaluate. The following result appears in [Cohen, 1993d]:

Theorem 2 *Recall that $\mathcal{L}^{k\text{-DEPTH}}[B]$ is the language of depth-k logic programs consisting of a single nonrecursive function free clause, and a-$\mathcal{B}$ is the set of background theories B containing only ground facts of arity a or less.*

There exists a theory $B \in 3$-$\mathcal{B}$ and a concept C in $\mathcal{L}^{1\text{-DEPTH}}[B]$ such that evaluating C is NP-hard.

Again for completeness we will sketch the proof, which is also a reduction from 3-SAT.

Proof (sketch): Now, consider a 3-CNF formula $\phi = \wedge_{i=1}^{n}(l_{i_1} \vee l_{i_2} \vee l_{i_3})$ over the n variables $x_1, \ldots, x_n$. To show evaluation hardness, we must show that ϕ can be encoded as an instance e_ϕ, and that there exists some background theory B and some clause $C_{SAT} \in \mathcal{L}^{1\text{-DEPTH}}[B]$ that is true exactly when the instance e_ϕ is satisfiable. To do this, let us encode ϕ as the arity-$3n$ atom

$$e_\phi = sat(m_{1_1}, m_{1_2}, m_{1_3}, \ldots, m_{n_1}, m_{n_2}, m_{n_3})$$

where $m_{i_j} = k$ if $l_{i_j} = x_k$ and $m_{i_j} = -k$ if $l_{i_j} = \overline{x_k}$.

Now, let B contain the following predicates:

- The predicate *boolean(X)* is true if $X \in \{0,1\}$.
- For $k = 1$ through n, the predicate $link_k$*(M,V,X)* is true if $M \in \{-n,\ldots,-1\}$ or $M \in \{1,\ldots,n\}$, $V \in \{0,1\}$, $X \in \{0,1\}$, and one of the following conditions also holds: $M = k$ and $X = V$; $M = -k$ and $X = \neg V$; or $M \neq k$ and $M \neq -k$.
- The predicate $sat(V_1, V_2, V_3)$ is true if each $V_i \in \{0,1\}$ and if one of V_1, V_2, V_3 is equal to 1.

Now consider the clause below:

$$
\begin{aligned}
&sat(M_{1_1}, M_{1_2}, M_{1_3}, \ldots, M_{n_1}, M_{n_2}, M_{n_3}) \leftarrow \\
&\quad \textstyle\bigwedge_{k=1}^{n}\ boolean(X_k) \wedge \\
&\quad \textstyle\bigwedge_{i=1}^{n} \wedge_{j=1}^{3}\ boolean(V_{i_j}) \wedge \\
&\quad \textstyle\bigwedge_{i=1}^{n} \wedge_{j=1}^{3} \wedge_{k=1}^{n}\ link_k(M_{i_j}, V_{i_j}, X_k) \wedge \\
&\quad \textstyle\bigwedge_{i=1}^{n}\ sat(V_{i_1}, V_{i_2}, V_{i_k})
\end{aligned}
$$

It can be easily argued that this clause succeeds exactly when ϕ is satisfiable. Briefly, first set of *boolean* literals introduce two sets of non-deterministic variables, the X_i's and the V_{i_j}'s. The X_k's will correspond to possible assignments for the variables $x_1, \ldots, x_n$ over which ϕ is defined, and the V_{i_j}'s correspond to possible assignments of the literals l_{i_j} that appear in ϕ. The conjunction of the link$_k$ literals ensure that the V_{i_j}'s have values consistent with some assignment to the x_i's; in otherwords, for the conjunction of the link$_k$ variables to succeed, if must be that whenever $l_{i_j} = x_k$, then V_{i_j} and X_k are bound the same value, and whenever $l_{i_j} = \overline{x_k}$, then V_{i_j} and X_k are bound to complementary values. Finally, the *sat* literals ensure that the values given to the V_{i_j}'s are such that every clause in ϕ is satisfied.

For example, consider again the formula ϕ_0 below.

$$\phi_0 = (\overline{x_1} \vee \overline{x_2} \vee \overline{x_3}) \wedge (\overline{x_1} \vee \overline{x_2} \vee x_3) \wedge (\overline{x_1} \vee x_2 \vee \overline{x_3})$$

This would be encoded as the atom

$$e_{\phi_0} = sat(-1,-2,-3,\ -1,-2,+3,\ -1,+2,-3)$$

Now consider the *sat* clause for 3-clause 3-CNF over 3 variables shown in Table 1. This clause has been commented as follows. The underlined *link* literals are the ones that actually restrict the bindings of their arguments when attempting to prove the atom e_{ϕ_0}. After selected groups of literals, comments have been inserted pointing out the relationships between the free variables that must hold if all preceding literals succeed, again when attempting to prove the atom e_{ϕ_0}. It This clause will succeed with the bindings $X_1 = X_2 = X_3 = 0$, corresponding to the satisfying assignment $x_1 = x_2 = x_3 = 0$ for ϕ_0. ■

Due to Schapire's result, the following is an immediate corollary:

Corollary 2 *For $k \geq 1$ and $a \geq 3$, $\mathcal{L}^{k\text{-DEPTH}}[a\text{-}\mathcal{B}]$ is not pac-learnable, and neither is any language at least as expressive as a language in this family.*

$$
\begin{array}{l}
\mathit{sat}(M_{11}, M_{12}, M_{13}, M_{21}, M_{22}, M_{23}, M_{31}, M_{32}, M_{33}) \leftarrow \\
\quad \mathit{boolean}(X_1) \wedge \mathit{boolean}(X_2) \wedge \mathit{boolean}(X_3) \wedge \\
\quad \textit{\% every } X_i \textit{ is bound to 0 or 1} \\
\\
\quad \mathit{boolean}(V_{11}) \wedge \mathit{boolean}(V_{12}) \wedge \mathit{boolean}(V_{13}) \wedge \\
\quad \mathit{boolean}(V_{21}) \wedge \mathit{boolean}(V_{22}) \wedge \mathit{boolean}(V_{23}) \wedge \\
\quad \mathit{boolean}(V_{31}) \wedge \mathit{boolean}(V_{32}) \wedge \mathit{boolean}(V_{33}) \wedge \\
\quad \textit{\% every } V_{ij} \textit{ is bound to 0 or 1} \\
\\
\quad \underline{\mathit{link}_1(M_{11}, V_{11}, X_1)} \wedge \mathit{link}_1(M_{12}, V_{12}, X_1) \wedge \mathit{link}_1(M_{13}, V_{13}, X_1) \wedge \\
\quad \underline{\mathit{link}_1(M_{21}, V_{21}, X_1)} \wedge \mathit{link}_1(M_{22}, V_{12}, X_1) \wedge \mathit{link}_1(M_{23}, V_{23}, X_1) \wedge \\
\quad \underline{\mathit{link}_1(M_{31}, V_{31}, X_1)} \wedge \mathit{link}_1(M_{32}, V_{12}, X_1) \wedge \mathit{link}_1(M_{33}, V_{33}, X_1) \wedge \\
\quad \textit{\% } V_{11}, V_{21}, \textit{ and } V_{31} \textit{ are complements of } X_1 \\
\\
\quad \mathit{link}_2(M_{11}, V_{11}, X_2) \wedge \underline{\mathit{link}_2(M_{12}, V_{12}, X_2)} \wedge \mathit{link}_2(M_{13}, V_{13}, X_2) \wedge \\
\quad \mathit{link}_2(M_{21}, V_{21}, X_2) \wedge \underline{\mathit{link}_2(M_{22}, V_{12}, X_2)} \wedge \mathit{link}_2(M_{23}, V_{23}, X_2) \wedge \\
\quad \mathit{link}_2(M_{31}, V_{31}, X_2) \wedge \underline{\mathit{link}_2(M_{32}, V_{12}, X_2)} \wedge \mathit{link}_2(M_{33}, V_{33}, X_2) \wedge \\
\quad \textit{\% } V_{12} \textit{ and } V_{22} \textit{ are complements of } X_2 \\
\quad \textit{\% } V_{32} \textit{ is equivalent to } X_2 \\
\\
\quad \mathit{link}_1(M_{11}, V_{11}, X_1) \wedge \mathit{link}_1(M_{12}, V_{12}, X_1) \wedge \underline{\mathit{link}_1(M_{13}, V_{13}, X_1)} \wedge \\
\quad \mathit{link}_1(M_{21}, V_{21}, X_1) \wedge \mathit{link}_1(M_{22}, V_{12}, X_1) \wedge \underline{\mathit{link}_1(M_{23}, V_{23}, X_1)} \wedge \\
\quad \mathit{link}_1(M_{31}, V_{31}, X_1) \wedge \mathit{link}_1(M_{32}, V_{12}, X_1) \wedge \underline{\mathit{link}_1(M_{33}, V_{33}, X_1)} \wedge \\
\quad \textit{\% } V_{13} \textit{ and } V_{33} \textit{ are complements of } X_3 \\
\quad \textit{\% } V_{23} \textit{ is equivalent to } X_3 \\
\\
\quad \mathit{sat}(V_{11}, V_{12}, V_{13}) \wedge \mathit{sat}(V_{21}, V_{22}, V_{33}) \wedge \mathit{sat}(V_{31}, V_{32}, V_{33}). \\
\quad \textit{\% the bindings of the } V_{ij}\textit{'s make the clause succeed}
\end{array}
$$

Table 1: A clause testing satisfiability of 3-clause 3-CNF

Discussion. This theorem is superficially quite similar to Theorem 1; specifically, both are based on a reduction from 3-SAT. However, evaluation-hardness gives a much more useful negative result, as if a language is evaluation-hard additional restrictions must be imposed in order to make it pac-learnable.

Nonetheless, evaluation-hardness is not a very useful proof technique. The main problem is that it usually not applicable, as most reasonable concept classes can be evaluated in polynomial time. A related point is that it is always easy to restrict a concept class so as to avoid a evaluation-hardness result—for example, one could restrict the concept class of depth-k clauses to the concept class of depth-k clauses that can be evaluated in polynomial time. By definition the latter class cannot be hard to evaluate; however, it is likely to be just as useful in practical learning situations, since typically one is not interested in finding a hypothesis that cannot be evaluated easily.

3.3 Predictability hardness

A third important proof technique that has been used for ILP problems is *prediction-preserving reducibility.* Intuitively, consistency-hardness and evaluation-hardness results rely on taking a computationally difficult problem (such as 3-SAT) and reducing it to some stage of a pac-learning algorithm. In contrast, prediction-preserving reducibility involves directly reducing one pac-learning problem to another. In this section we will outline the ideas behind this proof technique and give some examples of its application.

3.3.1 Pac-predictability

To describe this proof technique, let us first introduce a slight variation of the pac-learning model. We define a language $\mathcal{L}$ to be *pac-predictable* iff there is an algorithm PREDICT that satisfies all of the requirements for pac-learning, save that its hypothesis is (perhaps) not in $\mathcal{L}$: instead the hypothesis of PREDICT is simply required to be some classifier that can be evaluated in polynomial time.[1] Pac-predictability of a language family $\mathcal{L}[\mathcal{B}]$ is defined analogously to uniform pac-learnability.

We note that in ILP applications, it has often been important that the output of the learning system is expressed in an understandable language [King *et al.*, 1992, Cohen, 1994c]. Because of this a pac-prediction algorithm may be much less desirable than a pac-learning algorithm. For purposes of analysis, however, this learning model is quite useful. Previous results in computational learning theory have shown that there are two different reasons that a language $\mathcal{L}$ may fail to be pac-learnable. It may be that it is computationally difficult to find a probably approximate hypothesis of any sort in polynomial time. On the other hand, it may be possible to find a probably approximate hypothesis quickly, but difficult to encode

[1] More formally, $\mathcal{L}$ is *pac-predictable* iff there is an algorithm PREDICT and a polynomial function for every $n_t > 0$, for every $n_e > 0$, for every $C_t \in \mathcal{L}_{n_t}$, for every $\epsilon : 0 < \epsilon < 1$, for every $\delta : 0 < \delta < 1$, for every probability distribution function D, if S^+, S^- is a sample of C_t from X_{n_e} drawn according to D that contains at least $m(\frac{1}{\epsilon}, \frac{1}{\delta}, n_e, n_t)$ examples, then (a) PREDICT(S^+,S^-,ϵ,δ) outputs a hypothesis H such that $Prob(D(H-C)+D(C-H) > \epsilon) < \delta$ (b) PREDICT runs in time polynomial in $\frac{1}{\epsilon}$, $\frac{1}{\delta}$, n_e, n_t, and the number of examples if S^+ and S^-, and (c) the hypothesis H can be evaluated in time polynomial in the size of its input.

this hypothesis in the desired target language $\mathcal{L}$. The definition of pac-predictability allows one to separate out these two issues, and thus gain insight into *why* a language is hard to pac-learn.

Languages that are pac-predictable also have a useful closure property, as noted below.

3.3.2 Reducibilities between prediction problems

Consider two languages $\mathcal{L}_1$ over domain X_1 and $\mathcal{L}_2$ over domain X_2, and suppose that there are two functions $f_i : X_1 \to X_2$ and $f_c : \mathcal{L}_1 \to \mathcal{L}_2$ such that the following hold:

1. $x \in C$ if and only if $f_i(x) \in f_c(C)$ — i.e., concept membership is preserved by the mappings;
2. the size complexity of $f_c(C)$ is polynomial in the size complexity of C — i.e. the size of concepts is preserved within a polynomial factor; and
3. $f_i(x)$ can be computed in polynomial time.

Intuitively, the function $f_c(C_1)$ (called the *concept mapping*) returns a concept $C_2 \in \mathcal{L}_2$ that will "emulate" C_1 (i.e., make the same decisions about concept membership) on examples that have been "preprocessed" with the function f_i (the *instance mapping*). If these functions exist, then one possible scheme for pac-learning a concept $C_1 \in \mathcal{L}_1$ is to preprocess all examples of C_1 with f_i, and then use these preprocessed examples to pac-learn some H_2 that is a good approximation of $C_2 = f_c(C_1)$. H_2 can then be used to pac-predict membership in C_1: given an example x from the original domain X_1, one can simply predict $x \in C_1$ to be true whenever $f_i(x) \in H_2$. It is easy to show that if H_2 is a probably approximate correct classifier for C_2, then the function $(f_i \circ H_2)$ is a probably approximate correct classifier for C_1.

More formally, let us make the following definition: if $\mathcal{L}_1$ is a language over the domain X_1 and $\mathcal{L}_2$ is a language over the domain X_2, then *predicting* $\mathcal{L}_1$ *is reducible to predicting* $\mathcal{L}_2$ (written $\mathcal{L}_1 \trianglelefteq \mathcal{L}_2$) if there exists a polynomial p and functions $f_i : X_1 \to X_2$ and $f_c : \mathcal{L}_1 \to \mathcal{L}_2$ such that (a) $x \in C$ iff $f_i(x) \in f_c(C)$, (b) for all $C \in \mathcal{L}_1$ $\|f_c(C)\| \leq p(\|C\|)$, and (c) $f_i(x)$ can be computed in time bounded by $p(\|x\|)$. Pitt and Warmuth [1990] give a rigorous proof of the following theorem.

Theorem 3 (Pitt and Warmuth) *If $\mathcal{L}_1 \trianglelefteq \mathcal{L}_2$ and $\mathcal{L}_2$ is pac-predictable, then $\mathcal{L}_1$ is pac-predictable.*

Taking the contrapositive of this theorem and noting that pac-learnability is a special case of pac-predictability leads to the following observation: if $\mathcal{L}_1 \trianglelefteq \mathcal{L}_2$ then $\mathcal{L}_2$ is pac-learnable only if $\mathcal{L}_1$ is pac-predictable. A number of languages have been discovered that are unlikely to be pac-predictable, including arbitrary boolean functions, arbitrary deterministic finite state machines (DFAs), and log-depth boolean formulae [Kearns and Valiant, 1989]. (We say "unlikely" because most of these negative results have been obtained by reducing certain cryptographic encodings to learning problems; thus the prediction-hardness of the languages usually depends on cryptographic assumptions.) Prediction-preserving reducibilities are thus a powerful technique for proving negative results: to show the non-learnability of a language $\mathcal{L}_2$, one must simply find a language $\mathcal{L}_1$ known to be hard and show that $\mathcal{L}_1 \trianglelefteq \mathcal{L}_2$.

Another application of prediction-preserving reducibilities is best explained by analogy to complexity theory. Although little is known in absolute terms about the complexity of computational problems, much progress has been made in establishing relationships between the complexity of certain problems—for instance, it is useful to know that graph coloring is (within a polynomial) exactly as difficult as finding a Hamiltonian path in a graph. Similarly, prediction-preserving reductions allow one to obtain a sense of the relative difficulty of various learning problems.

3.3.3 Predicting clauses with one free variable

To illustrate this technique we will present a proof showing that the problem of pac-predicting boolean formulae in disjunctive normal form (DNF) can be reduced to pac-predicting a certain highly restricted class of depth-1 nonrecursive clauses. The pac-predictability of DNF is a long-standing open problem in computational learning theory. No provably correct pac-learning algorithms for DNF are known; further, most experimental algorithms that use heuristics to learn DNF rely on assumptions about the distribution of examples. It seems unlikely that DNF is learnable in the distribution-free sense required by pac-learnability (although it may be learnable under certain classes of natural distributions).

Let the *free variables* of a clause be those variables that appear in the body of the clause but not in the head, and let the language $\mathcal{L}^{k\text{-FREE}}$ be the language of nonrecursive function-free clauses containing at most k free variables. The language of k-free clauses is a further restriction of the language of depth-k clauses; however, restricting the number of free variables to a constant ensures that clauses can be evaluated in polynomial time.

The following result appears in [Cohen, 1993a]:

Theorem 4 *For all n, there exists a background database $B_n \in 2\text{-}\mathcal{B}$ of size polynomial in n such that predicting n-term DNF formulae can be reduced to predicting $\mathcal{L}^{1\text{-FREE}}[B_n]$.*

Again for completeness we will sketch the proof.

Proof (sketch): Let B_n contain sufficient atomic facts to define the binary predicates $true_1$, $false_1$, …, $true_n$, $false_n$ that behave as follows:

- $true_i(X, Y)$ succeeds if $X = 1$, or if $Y \in \{1, \ldots, n\} - \{i\}$.
- $false_i(X, Y)$ succeeds if $X = 0$, or if $Y \in \{1, \ldots, n\} - \{i\}$.

Note that DNF is defined over the domain of assignments to n boolean variables.[2] These instances can be encoded as bit vectors $b_1 \ldots b_n$. Define the instance mapping f_i to map an

[2]Strictly speaking, the number of terms in the DNF and the number of variables can be different, so properly speaking we should use two variables distinct variables n_1 and n_2 for these quantities. However, since the reduction is must be polynomial in both quantities, there's no harm in using a single n.

assignment $b_1 \ldots b_n$ to the atom *dnf*$(b_1, \ldots, b_n)$, and define the concept mapping f_c to map a formula of the form

$$\phi \equiv \bigvee_{i=1}^{n} \bigwedge_{j=1}^{s_i} l_{ij}$$

(where each l_{ij} is either x_k or $\overline{x_k}$) to the clause

$$f_c(\phi) \equiv dnf(X_1, \ldots, X_n) \leftarrow \bigwedge_{i=1}^{n} \bigwedge_{j=1}^{s_i} Lit_{ij}$$

where Lit_{ij} is defined as follows:

$$Lit_{ij} \equiv \begin{cases} true_i(X_k, Y) & \text{if } l_{ij} = x_k \\ false_i(X_k, Y) & \text{if } l_{ij} = \overline{x_k} \end{cases}$$

As an example of the concept mapping, the following formula over the variables x_1, x_2, x_3, x_4

$$(v_1 \wedge v_2 \wedge \overline{v_3}) \;\vee\; (v_1 \wedge \overline{v_2} \wedge v_3) \;\vee\; (\overline{v_1} \wedge v_2 \wedge v_3)$$

would be mapped to the clause

$dnf(X_1,X_2,X_3,X_4) \leftarrow$
$\quad true_1(X_1, Y) \wedge true_1(X_2, Y) \wedge false_1(X_3, Y) \wedge$
$\quad true_2(X_1, Y) \wedge false_2(X_2, Y) \wedge true_2(X_3, Y) \wedge$
$\quad false_3(X_1, Y) \wedge true_3(X_2, Y) \wedge true_3(X_3, Y).$

It is easy to show that f_i is polytime evaluable and that f_c preserves size within a polynomial. Let us consider the final criterion for a prediction-preserving reducibility, the requirement that

$$x \in C \Leftrightarrow f_i(x) \in f_c(C)$$

If ϕ is true for an assignment $b_1 \ldots b_n$, then some term $T_i = \bigwedge_{j=1}^{s_i} l_{ij}$ must be true. In this case $\bigwedge_{j=1}^{s_i} Lit_{ij}$ succeeds with Y bound to the value i and $\bigwedge_{j=1}^{s_{i'}} Lit_{i'j}$ for every $i' \neq i$ also succeeds with Y bound to i; hence the clause $f_c(\phi)$ succeeds.

On the other hand, if ϕ is false for an assignment, then each T_i fails, and hence for every binding of Y in the range $1, \ldots, n$ some conjunction $\bigwedge_{j=1}^{s_i} Lit_{ij}$ will fail. Thus concept membership is preserved by the mapping. ■

One formal consequence of this reduction is the following.

Corollary 3 *For $k \geq 1$ and $a \geq 2$, the language family $\mathcal{L}^{k\text{-FREE}}[a\text{-}\mathcal{B}]$ is pac-learnable only if boolean DNF is pac-predictable. The same is true for any any language at least as expressive as a language in this family.*

Discussion. To be interpreted as a negative result, Corollary 3 requires a much stronger assumption than Corollaries 1 and 2 (namely, the prediction-hardness of DNF). However, Corollary 3 is much stronger in other ways. In particular, it holds regardless of the hypothesis language used by the learning system, and hence applies to actual ILP systems; also, it holds for a language restricted enough to be plausibly considered as the hypothesis language for a learning system. The result also holds for all languages at least as expressive as the language of 1-free clauses.

Another benefit of Theorem 4 is that this reduction clarifies our understanding of the problem of learning Prolog clauses. Because of the superficial syntactic similarity of first-order definite clauses and Boolean conjunctions, it is tempting to believe that the same methods that work for learning Boolean conjunctions can be applied to first-order definite clauses, and that the same methods that (experimentally) work for learning DNF can be applied to learning predicate definitions consisting of several definite clauses. Theorem 4 shows that the expressive power of a single first-order definite clause is very different from that of a single Boolean conjunction; in fact, a single clause is at least as expressive as a DNF formulae with polynomially many terms.

This suggests that very different learning algorithms will be needed to learn first-order definite clauses, and conversely, that straightforward adaptations of zeroth-order learning algorithms are likely to effectively learn only a restricted subset of first-order definite clauses.

3.3.4 Predicting atomic Datalog schemata

Thus far the languages we have considered have strictly limited the number of clauses, but have allowed relatively powerful clauses, and have imposed few restrictions on background knowledge. In this section we will analyze a different type of logic program: programs that may contain any number of very simple clauses. In particular, we will require all clauses to have empty bodies, and hence to make no use of background knowledge. We will show that even this class of logic programs is as hard to pac-predict as DNF.

In fact, we will even require that these unit clauses, or atoms, contain only variables—that is, contain no constants or function symbols. More precisely, define an *atomic Datalog schema* to be a logic program that consists of a set of unit clauses (aka a conjunction of universally-closed atoms)[3] that contain no function symbols or constants. Let $\mathcal{L}^{\mathrm{ADS}}$ be the concept class of atomic Datalog schemata. (Note that this is not a language family, as no background knowledge is allowed.) We have the following result:

Theorem 5 $\mathcal{L}^{\mathrm{ADS}}$ *is pac-learnable only if DNF is pac-learnable.*

Proof: We will show that predicting r-term DNF over n variables can be reduced to predicting an atomic Datalog schema consisting of r unit clauses of arity $n+2$. The argument is similar to the earlier prediction-preserving reductions, with the following exception. In the earlier reductions, the choice of a particular background theory was crucial; in this reduction, no background theory is available. (We will also carry out the argument in somewhat more detail, since we are not simply sketching a proof that appears elsewhere.)

The domain for the DNF language is truth assignments to a set of n boolean variables $x_1, \ldots, x_n$. Each assignment can be represented as a length-n bit vector. Let f_i map a given truth assignment $\eta = b_1 \ldots b_n$ to the ground atom $p(b_1, ..., b_n, 0, 1)$. For example, the assignments 10101, 00110, and 11100 are mapped by f_i to the atoms $p(1,0,1,0,1,0,1)$,

[3]The universal closure of a formula ϕ is the result of universally quantifying all free variables in ϕ. Throughout this section, when we refer to atoms we mean universally closed atoms.

$p(0,0,1,1,0,0,1)$, and $p(1,1,1,0,0,0,1)$, respectively. Trivially $f_i(\eta)$ can be computed in polynomial time.

We define the concept mapping f_c using a function f_t that maps terms of a DNF formula (over n propositional variables) to atoms built from the $(n+2)$-ary predicate p and variables. Let f_t map a term T_i to an atom $p(t_1, ..., t_n, Y_0, Y_1)$ where for all $1 \leq j \leq n$, t_j is:

- the variable Y_0 if T_i contains the boolean literal $\overline{x_j}$;
- the variable Y_1 if T_i contains the boolean literal x_j;
- the variable X_j if T_i contains neither $\overline{x_j}$ nor x_j.

For example, the conjunction $\overline{x_3} \wedge x_4 \wedge \overline{x_5}$ (over five variables) is mapped by f_t to the atom

$$p(X_1, X_2, Y_0, Y_1, Y_0, Y_0, Y_1)$$

The function f_c maps a DNF formula $\phi = T_1 \vee ... \vee T_r$ to an atomic Datalog schema consisting of the atoms $f_t(T_1), f_t(T_2), ..., f_t(T_r)$. As an example, again take $n = 5$. The DNF formula

$$(x_1 \wedge \overline{x_2}) \vee (\overline{x_3} \wedge x_4 \wedge \overline{x_5}) \vee (x_2 \wedge x_3)$$

is mapped by f_c to the atomic Datalog schema

$p(Y_1, Y_0, X_3, X_4, X_5, Y_0, Y_1)$
$p(X_1, X_2, Y_0, Y_1, Y_0, Y_0, Y_1)$
$p(X_1, Y_1, Y_1, X_4, X_5, Y_0, Y_1)$

It remains to verify that, for any DNF formula $\phi = T_1 \vee ... \vee T_n$ and any truth assignment η over propositional variables $x_1, ..., x_n$, the assignment η satisfies ϕ if and only if $f_c(\phi)$ entails $f_i(\eta)$. A property useful in establishing this is *strong compactness* [Lassez *et al.*, 1988]. By strong compactness, a conjunction of atoms entails a given atom β just if some atom in the conjunction subsumes β. Hence we need only to show that ϕ is satisfied by η if and only if $f_c(\phi)$ contains an atom that subsumes $f_i(\eta)$. This means that it is sufficient to show that a given disjunct T_i in ϕ is satisfied by η iff $f_t(T_i)$ subsumes $f_i(\eta)$.

Before showing this property, we illustrate it using the preceding DNF formula. Let η be the truth assignment 10101. The assignment η satisfies the first disjunct, $x_1 \wedge \overline{x_2}$, but neither of the other two disjuncts, $\overline{x_3} \wedge x_4 \wedge \overline{x_5}$ or $x_2 \wedge x_3$. Similarly, the ground atom $f_i(\eta) = p(1,0,1,0,1,0,1)$ is subsumed by the first atom in the corresponding database, $p(Y_1, Y_0, X_3, X_4, X_5, Y_0, Y_1)$, but by neither of the other two.

In general, let ϕ be a DNF formula over n propositional variables, let T_i be an arbitrary disjunct in ϕ, and let η be an arbitrary truth assignment to the n propositional variables. Let α be the atom in $f_c(\phi)$ corresponding to T_i (*i.e.* $\alpha = f_t(T_i)$), and let $\beta = p(c_1, ..., c_n, 0, 1)$ be the ground atom $f_i(\eta)$.

Note that α subsumes β iff there is a substution θ such that $\alpha\theta = \beta$. Let us consider under what circumstances such a θ exists. Clearly, it is always possible to map each of the X_j's to the corresponding constant value b_j. Clearly also if θ exists, it must map Y_0 to 0 and Y_1 to 1. This is possible if and only if for every position $j : 1 \leq j \leq n$ in which Y_1 (respectively

Y_0) appears in α, the corresponding position in β contains the value 1 (respectively 0). By the definition of α and β, this happens precisely when for every literal x_j ($\overline{x_j}$) in T_i the bit b_j in η is set to 1 (0). But this is precisely the case in which the truth assignment η makes T_i true.

Discussion. All of our previous negative results have required constructing a background theory that makes learning hard. This suggests that restricting the types of background knowledge might lead to a learnable family of languages.

Theorem 5 shows that a language that allows no background knowledge and no function symbols can still be as hard to learn as DNF. This result shows that simply restricting background knowledge is not enough to make logic programs learnable. It also emphasizes the expressive power available in a program consisting of a polynomially-large set of clauses, rather than a small set of clauses.

4 Selected learning techniques

Not all of the formal results for ILP problems are negative. In this section we will survey some of the more interesting positive learnability results. We will present two main groups of results, one on the learnability of recursive programs, and one on the learnability of non-recursive programs.

In Section 2.1.2 we introduced the model of learning from equivalence queries, and in Section 3.3.1 we introduced the notion of pac-predictability. We can define a prediction variant of learning by equivalence queries in the same manner. Recall that in the original definition, "an *equivalence query* is a concept $C_q \in \mathcal{C}$, which is sent to an oracle..." In the prediction variant, the concept C_q need not be a member of $\mathcal{C}$, but may be any classifier that can be evaluated in polynomial time; such an equivalence query is sometimes called an *extended equivalence query*. The results in this section will focus on the stronger model of learning from equivalence queries.

4.1 Example-guided enumeration

Most proofs of learnability from equivalence queries are constructive: that is, they work by describing an equivalence-query learning algorithm, and then proving that the algorithm is correct. Probably the most straightforward learning algorithm (and hence one of the easiest to analyze) is an enumerative algorithm. In its simplest form, an enumerative algorithm checks the members of its hypothesis space in some order, and then conjectures the first hypothesis that is consistent with the data that it has so far.

Of course, we are interested in an algorithm only if it will run in time polynomial in the size of its data set and of the target hypothesis. For a simple enumerative algorithm to do so, the number of hypotheses preceding any given hypothesis h in the enumeration must be small, specifically, polynomial in the size of h. But this requirement is drastic, eliminating many interesting concept classes. Can it be relaxed by using a slightly smarter enumerative algorithm?

Note that the simple enumerative algorithm never uses its data to alter or eliminate a portion of the enumeration, but faithfully plods along through the enumeration, checking hypotheses against the data. One possible modification to this algorithm is to use portions of the data to dynamically determine the enumeration that will be used.

This is the form of the equivalence query learners described in this subsection. (Additional details regarding these learning algorithms can be found in [Frazier and Page, 1993a, Frazier and Page, 1993b, Frazier, 1994].) Although the size of the hypothesis spaces are exponential, the learners can always find the correct hypothesis by doing one or more enumerative searches in small (polynomial-sized) spaces. The small search spaces are determined by analyzing a small group of examples.

4.1.1 Binary logic programs with unary constructors

We begin by motivating and defining the concept classes to be learned. There are a large number of negative results on the learnability of recursive concepts [Cohen, 1993b, Cohen, 1993a]; hence a reasonable starting point is the simplest possible form of recursive program that can do anything interesting at all. Therefore, we consider two-clause *binary* programs, or programs with at most two literals per clause, built from unary predicates, unary functions, constants, and variables. (Further tightening any of these restrictions makes recursion completely uninteresting, or in some cases impossible.) We call this concept class $H_{2,1}$. For simplicity in this presentation, we assume that every atom is built from the same predicate, p, although this is not necessary to ensure learnability.

As an example, the following is a concept in $H_{2,1}$.

Example 6 *An example of an $H_{2,1}$ concept*

$$\begin{array}{l} p(f(g(g(a)))) \\ p(f(h(h(Y)))) \leftarrow p(f(Y)) \end{array}$$

We will sometimes refer to $p(f(g(g(X))))$ as the *base atom* and $p(f(h(h(Y)))) \leftarrow p(f(Y))$ as the *recursive clause.* This concept "generates" the following atoms.

$$\begin{array}{l} p(f(g(g(a)))) \\ p(f(h(h(g(g(a)))))) \\ p(f(h(h(h(h(g(g(a)))))))) \\ p(f(h(h(h(h(h(h(g(g(a)))))))))) \\ \vdots \end{array}$$

Thus, this concept would classify $p(f(g(g(f(a)))))$ as *positive* and $p(g(f(a)))$ as *negative.*

Since function symbols (other than constants) are unary, we can omit the parentheses separating unary function symbols without loss of clarity. For example, we can write the preceding concept as

$$\begin{array}{l} p(fgg(a)) \\ p(fhh(Y)) \leftarrow p(f(Y)) \end{array}$$

and the preceding list of atoms as

$p(fgg(a))$
$p(fhhgg(a))$
$p(fhhhhgg(a))$
$p(fhhhhhhgg(a))$
$\ldots$

We will usually view the unary function symbols in an atom as symbols in a string. Therefore, we will sometimes employ Greek letters to represent such strings, so that the following

$p(\alpha\beta(c))))$
$p(\alpha\gamma(Y)))) \leftarrow p(\alpha(Y))$

represents a concept in which the string in the antecedent of the recursive clause is a prefix of both the string in the consequent and the string in the base atom. In the concept of Example 6, α is f, β is gg, and γ is hh. It is worth noting that an atom $p(\alpha(X))$ is more general than a second atom $p(\beta(X))$ or $p(\beta(e))$, for a constant e, just if α is a prefix of β.

Further considering concepts of the general form

$p(\alpha\beta(e))))$
$p(\alpha\gamma(Y)))) \leftarrow p(\alpha(Y))$

where e is an arbitrary constant, notice that the base atom will unify with the antecedent of the recursive clause, binding the variable Y in the recursive clause to $\beta(e)$. This binding produces the atom $p(\alpha\gamma\beta(e))$. Running the recursive clause again, by unifying the antecedent with the newly produced atom, yields the atom $p(\alpha\gamma\gamma\beta(e))$. In general, the concept produces the following atoms.

$p(\alpha\beta(e))$
$p(\alpha\gamma\beta(e))$
$p(\alpha\gamma\gamma\beta(e))$
$p(\alpha\gamma\gamma\gamma\beta(e))$
$\vdots$

In short, the effect of running the recursive rule is to insert additional copies of γ into an atom. For the obvious reason, we call any such a concept an *expanding* concept. Example 6 is an expanding concept. Below is another example of an expanding concept, where α is f, β is f, and γ is ff.

Example 7 *Expanding Concept*

$p(ff(a))$
$p(fff(Y)))) \leftarrow p(f(Y))$

A careful analysis of $H_{2,1}$ reveals that each of its concepts can have one of three interesting forms.[4] One of these three forms is that of an expanding concept, seen already. The second is a non-recursive form, that is, consisting of two atoms. The following concept has this form.

[4]Technically, several other forms are possible. But these are less interesting, and their treatment here would only obscure the fundamental ideas. Please see [Frazier and Page, 1993a, Frazier and Page, 1993b, Frazier, 1994] for additional details.

Example 8 *Nonrecursive Concept*

$$p(fgf(X))$$
$$p(hff(Y))$$

This concept classifies a ground atom as *positive* just if the ground atom is an instance of one of the two atoms in the concept. The third form is that of a *contracting* concept, as follows.

$$p(\alpha\beta^k\gamma(X))))$$
$$p(\alpha(Y)))) \leftarrow p(\alpha\beta(Y))$$

(β^k denotes the result of concatenating k copies of the string β.) Running this rule once generates the atom $p(\alpha\beta^{k-1}\gamma(X))$. In general, running the rule has the effect of removing copies of β from an atom. The base atom in a contracting concept may end in a constant rather than a variable, if desired. The following is an example of a contracting concept, where α is gg, β is fg, and γ is h.

Example 9 *Contracting Concept*

$$p(ggfgfgfgfgfgh(a))$$
$$p(gg(Y)) \leftarrow p(ggfg(Y))$$

We can now make several useful observations about each of the three concept forms. For each concept form, these observations lead to an example-guided enumeration algorithm that learns, by equivalence queries, the subclass of concepts having that form. Afterward, we will see that these algorithms can be combined into a single example-guided enumeration algorithm that learns the entire class $H_{2,1}$ by equivalence queries.

We begin with the expanding concepts. First, an expanding concept is determined entirely by the strings α, β, and γ; if a learning algorithm determines these from the examples, then it has determined the concept. Second, given any two distinct positive examples of an expanding rule, at least one of the two examples is *not* the base atom. This example has the form $p(\alpha\gamma^k\beta(e))$, for some $k \geq 1$. Therefore, this example has "all the pieces", α, β, and γ, of the concept. Thus, the number of possibilities for the string α, given this example, is simply the number of substrings of this example. So where the example has length n, the number of possibilities for α is at most n^2 (choose the two endpoints of α within the example). Similarly, there are at most n^2 possible choices for β and γ (actually even fewer, given a particular choice of α).

So in fact the following example-guided enumeration algorithm learns the subclass of expanding concepts by (extended) equivalence queries.

1. Conjecture the empty concept (the concept that labels all examples *negative*). The counterexample A is necessarily a positive example.

2. Conjecture A. The counterexample B is necessarily a positive example.

3. In any order, conjecture the concepts of expanding form that result from each possible guess of α, β, and γ from A, halting if correct.

4. In any order, conjecture the concepts of expanding form that result from each possible guess of α, β, and γ from B, halting if correct.

Having considered the expanding concepts, we next consider the non-recursive concepts. A nonrecursive concept of the form

$$
\begin{array}{l}
p(\alpha(X)) \\
p(\beta(Y))
\end{array}
$$

is completely determined by α and β. (In general, either of the atoms could be ground, in which case the final constant must also be obtained. Obtaining it is simple, but the description is clearer if we ignore this possibility and assume both atoms are non-ground.) The example-guided enumeration algorithm for this subclass begins by conjecturing the empty concept and necessarily obtaining a positive counterexample $A = p(\alpha\gamma(e))$, where e is some constant. (Without loss of generality, we may assume that A is an instance of $p(\alpha(X))$.) The algorithm then conjectures, in any order, each atom that is more general than A, that is, each atom of the form $p(\alpha_i(X))$ where α_i is a prefix of $\alpha\beta$. Where n is the size of A, only n distinct conjectures can be made. The algorithm then retains each conjecture $p(\alpha_i)$ and the counterexample B_i, $1 \leq i \leq n$, to the conjecture. One of these conjectured atoms is the correct one, $p(\alpha(X))$, though it is not the entire concept; hence, the counterexample B to this conjecture must necessarily be a positive example which is an instance of the other atom in the concept, $p(\beta(Y))$. The algorithm then attempts to guess β from each B_i. For each guess α_i of α from A, and each guess β_j of β from B_i, the algorithm conjectures

$$
\begin{array}{l}
p(\alpha_i(X)) \\
p(\beta_j(Y))
\end{array}
$$

One of these conjectures must be correct. The total number of conjectures is $1 + n + n'$, where n' is the size of the largest B_i.

Finally, we consider the contracting concepts. Because a contracting concept has the form

$$
\begin{array}{l}
p(\alpha\beta^k\gamma(X)))) \\
p(\alpha(Y)))) \leftarrow p(\alpha\beta(Y))
\end{array}
$$

(β nonempty and $k \geq 2$), any such concept is uniquely determined by α, β, γ, and k. (The base atom could end in a constant rather than a variable, which makes learning simpler. For clarity of presentation, we assume that the base atom necessarily ends in a variable.) Notice that such a concept generates exactly the following atoms.

$$
\begin{array}{l}
p(\alpha\beta^k\gamma(X)) \\
p(\alpha\beta^{k-1}\gamma(X)) \\
\ldots \\
p(\alpha\beta\gamma(X)) \\
p(\alpha\gamma(X))
\end{array}
$$

The concept classifies a ground atom *positive* just if that ground atom is an instance of one of these generated atoms.

The example-guided enumeration algorithm for concepts of the contracting form begins by conjecturing the empty concept and necessarily obtaining a positive counterexample A. A is an instance of some generated atom $A' = p(\alpha\beta^j\gamma(X))$. Notice that if j were known to be at least 1, the algorithm could, from A, guess α, β, and γ, and then guess successively larger values of k. In fact, this is what will be done by the algorithm, but only after it ensures that it has a copy of β (an example with $j \geq 1$), as follows. Since A is an instance of the generated atom A', the algorithm can guess A' from A (in the same manner as did the algorithm for concepts of the non-recursive form). Given A, there are only $\|A\|$ guesses of A'. The algorithm then conjuctures each guess A'_i of A', and to each conjecture receives a counterexample B_i. One of the guesses A'_i of A' is correct, and to this conjecture the algorithm necessarily receives a positive counterexample B_i which is an instance of an atom, other than A', generated by the target concept. Since only one atom generated by the target concept has $j = 0$, either A or this B_i must have $j \geq 1$, that is, must contain a copy of β.

The algorithm now has $\|A\| + 1$ examples, at least one of which contains α, β, and γ. For each example, the algorithm makes all possible guesses of α, β, and γ. The number of combinations of guesses for α, β, and γ, given an example of size n, is at most n^6 (at most n^2 possible guesses for each), so the total number of such guesses by the algorithm is at most $(\|A\| + 1)(n^6) \leq (n + 1)(n^6)$, where n is the size of the largest example seen to this point. The algorithm now guesses successively larger values of k, beginning with 2. The algorithm will have to make $k - 1$ such guesses; this number of guesses is clearly less than the size of the target concept, since the base atom of the target has k copies of a nonempty string β. For each guess k' of k, the algorithm combines k' with all possible guesses of α, β, and γ, and conjectures each of the resulting concepts. From the preceding arguments, at any point the number of conjectures and time taken by the algorithm is polynomial in the size of the target and the size of the largest counterexample seen thus far. Thus, while any full enumeration of the contracting concepts (or of the concepts of the other two forms) is in general too time-consuming, an example-guided enumeration algorithm for this class is sufficient for polynomial-time learning.

Finally, we note that the three example-guided enumeration algorithms described to this point can be combined easily into a single algorithm that learns $H_{2,1}$ by equivalence queries, by interleaving the three algorithms. This interleaving can be done on a step-by-step basis, in which each algorithm executes one instruction in turn, or on a conjecture-by-conjecture basis.

Putting all this together gives the following result.

Theorem 10 *The language $H_{2,1}$ is learnable from extended equivalence queries.*

4.1.2 Increasing the predicate symbol arity

Following the previous result, it is natural to ask whether the arity of the function symbols or predicate symbols can be increased without losing learnability. Increasing the predicate symbol arity is a strictly weaker relaxation and, therefore, the obvious one to consider next. We will introduce one additional restriction, which we now motivate and define, that variables must be *stationary.* Notice that because functions are unary, each argument in a literal has at most one variable (it may have a constant instead), and that variable must be the last symbol of the argument. A concept meets the *stationary variables* restriction if for every variable x, if x appears in argument i of the consequent of the recursive clause, then x also appears in argument i of the antecedent. The following arithmetic concepts are examples have stationary variables.

Example 11 *Definition of plus using zero and successor*

$$plus(X, 0, X)$$
$$plus(X, s(Y), s(Z)) \leftarrow plus(X, Y, Z)$$

Example 12 *Definition of greater-than using zero and successor*

$$greater(s(X), 0)$$
$$greater(s(X), s(Y)) \leftarrow greater(X, Y)$$

The following concept does not meet the stationary variables restriction, because variables "shift positions" in the recursive clause.

Example 13 *A concept with variables that are not stationary*

$$p(a, b, c)$$
$$p(Z, X, Y) \leftarrow p(X, Y, Z)$$

The class of two-clause, binary logic programs built unary functions, constants, variables, and predicates of arbitrary arity is denoted by $H_{2,*}$.

It has been shown that the class $H_{2,*}$ is not learnable [Frazier and Page, 1993a, Frazier and Page, 1993b]. It is not known whether this class is pac-predictable. But it is known that if the predicate symbol arities in this class are restricted to be at most k, for any fixed k, and if variables are stationary, then the class is learnable by extended equivalence queries [Frazier and Page, 1993a, Frazier and Page, 1993b]. This restricted language is denoted by $H_{2,k}$. We now briefly outline how the learning algorithm for $H_{2,1}$ can be used to predict $H_{2,k}$ for any fixed k.

Let the following be a concept in $H_{2,k}$.

$$p(\alpha_1(X_1), ..., \alpha_k(X_k))$$
$$p(\gamma_1(Y_1), ..., \gamma_k(Y_k)) \leftarrow p(\beta_1(Y_1), ..., \beta_k(Y_k))$$

(Some of the variables may be replaced by constants under certain conditions, but we will not describe these conditions here.) Then we say the *subconcept at argument* i, for $1 \leq i \leq k$, of this concept is

$$p_i(\alpha_i(X_i))$$
$$p_i(\gamma_i(Y_i)) \leftarrow p_i(\beta_i(Y_i))$$

Each of the (at most) k subconcepts of a concept in $H_{2,k}$ is itself a concept in $H_{2,1}$ (although some may be of relatively uninteresting forms that were not described in the previous section). For example the concept *plus* as defined in Example 11 breaks into the following three subconcepts.

$plus_1(X)$	$plus_2(0)$	$plus_3(X)$
$plus_1(X) \leftarrow plus_1(X)$	$plus_2(s(Y)) \leftarrow plus_2(Y)$	$plus_3(s(Z)) \leftarrow plus_3(Z)$

It has been shown that, with several additional technicalities, these subconcepts can be learned simultaneously using the learning algorithm for $H_{2,1}$. In particular, any example $p(\alpha(e_1), \beta(e_2), \gamma(e_3))$ given to the learning algorithm for $H_{2,k}$ is divided into the following three examples, which are used in learning three distinct subconcepts in $H_{2,1}$: $p_1(\alpha(e_1))$, $p_2(\beta(e_2))$, and $p_3(\gamma(e_3))$. Conjectures for each of the three subconcepts, by the learning algorithm for $H_{2,1}$, are in turn combined into conjectures of the target concept in $H_{2,k}$. Because of the technicalities noted above, the conjectures are actually of a slightly more general concept class. Therefore, the resulting algorithm learns $H_{2,k}$ by *extended* equivalence queries and hence can be used, as well, to pac-predict $H_{2,k}$, though not to pac-learn $H_{2,k}$. As noted earlier, details can be found in [Frazier and Page, 1993a, Frazier and Page, 1993b].

This gives us the following result:

Theorem 14 *The language $H_{2,k}$ is learnable from extended equivalence queries.*

Discussion. By the same arguments as for learnability, uniform pac-predictability of a language family by equivalence queries (learnability by extended equivalence queries) implies uniform pac-predictability of that language family. Notice that every uniformly pac-learnable (learnable by equivalence queries) language family is, trivially, also uniformly pac-predictable (predictable by equivalence queries).

4.2 One-sided learning

4.2.1 The general idea

We will now turn our attention to efficient special-purpose learning algorithms; algorithms that take advantage of special properties of a concept class to quickly find a consistent hypothesis. Suppose that $\mathcal{L}$ has the following structure: every concept $C \in \mathcal{L}$ can be expressed as a disjunction

$$C = P_{i_1} \vee \ldots \vee P_{i_r}$$

where each P_{i_j} has polynomial size, and is an element of some set $\mathcal{P}$ that is of polynomial cardinality. One algorithm that learns $\mathcal{L}$ from equivalence queries is the following.

Algorithm LEARNDISJUNCTION($\mathcal{P}$):

1. Let $\mathcal{P}' = \mathcal{P}$ and let the current hypothesis be $C = \bigvee_{P_{i_j} \in \mathcal{P}'} P_{i_j}$
2. Ask the equivalence oracle if the current hypothesis is equivalent to the target concept.
   (a) If the answer is "equivalent" then return the current hypothesis C.

(b) Otherwise, the oracle returns a counterexample e, which must be a negative example of the target C_t. In this case, remove from $\mathcal{P}'$ all P' that cover e, replace the current hypothesis with

$$C = \bigvee_{P_{i_j} \in \mathcal{P}'} P_{i_j}$$

and return to Step 2.

To briefly sketch the argument for the correctness of this procedure, notice that the initial hypothesis is the most general concept in $\mathcal{L}$, and hence the most general hypothesis that is consistent with what is known about the target at this point. Also, after seeing a negative example e^-, it is clear if P' is a disjunct that covers e^-, then every hypothesis of the form

$$C = P_{i_1} \vee \ldots \vee P' \vee \ldots \vee P_{i_r}$$

will also cover e^-. Thus after seeing a negative example e^-, the updated hypothesis $C = \bigvee_{P_{i_j} \in \mathcal{P}'} P_{i_j}$ is again the most general concept in $\mathcal{L}$ consistent with what is known about the target. It is easy to see that since at every stage the most general consistent concept in $\mathcal{L}$ is hypothesized, LEARNDISJUNCTION will always receive negative counter-examples, and will eventually converge to the target concept C_t. Further the number of equivalence queries needed for convergence can be easily bounded by $|\mathcal{P}|$.

Notice that if the intermediate hypotheses of LEARNDISJUNCTION are used for prediction, then every time they predict an example to be negative, it must be negative (assuming that the target concept is in $\mathcal{L}$); thus the only errors made are on examples that are assigned positive labels. For this reason LEARNDISJUNCTION is sometimes called a *one-sided* learning algorithm, or a learning algorithm with *one-sided error*.

4.2.2 Programs of small non-recursive clauses

To apply the LEARNDISJUNCTION technique in an ILP context, let $\mathcal{P}^k$ be the set of all function-free clauses P that satisfy the following conditions:

- P is nonrecursive;
- the number of literals in the body of P is no greater than k;
- the head of P contains only variables, all of which are distinct.

It can be shown easily that if one fixes a background theory B and considers only those elements of $\mathcal{P}^k$ that have non-empty extensions over B, the cardinality of $\mathcal{P}^k$ is polynomial in $\|B\|$ and the number of variables in the head of P. Note also that semantically, a program of clauses from $\mathcal{P}^k$ behaves like a disjunction of these clauses: *i.e.,* if $M(P)$ denotes the least Herbrand model of a program $C = \{P_1, \ldots, P_r\}$, then

$$M(\{P_1, \ldots, P_r\}) = M(\{P_1\}) \cup \ldots \cup M(\{P_r\})$$

Thus the LEARNDISJUNCTION algorithm can be used to learn programs made up of clauses from $\mathcal{P}^k$. This leads to the following result (which first appeared in [Cohen, 1993b].)

Theorem 15 *Let $\mathcal{L}^{k\text{-LENGTH}}[B]$ be the set of logic programs composed of clauses from $\mathcal{P}^k$. Then for all a the language family $\mathcal{L}^{k\text{-LENGTH}}[a\text{-}\mathcal{B}]$ is uniformly learnable from equivalence queries using the learning algorithm* LEARNDISJUNCTION($\mathcal{P}^k$).

Discussion. As noted above, this result implies immediately that the language family $\mathcal{L}^{k\text{-LENGTH}}[B]$ is uniformly pac-learnable. One pac-learning algorithm for this class is imply to simply collect appropriate-sized sample S^+, S^- and hypothesis a logic program that contains every clause $P \in \mathcal{P}^k$ that does not cover any negative examples.

We also note that the requirement that the head of each clause must contain only variables, all of which are distinct, is necessary. Theorem 5 shows that if repeated variables are allowed, then even the language $\mathcal{L}^{0\text{-LENGTH}}$ (programs of unit clauses) is as hard to learn as DNF. A similar (and simpler) result shows that if no variables repeat, but constants are allowed in the heads, then again $\mathcal{L}^{0\text{-LENGTH}}$ is as hard to learn as DNF.

4.2.3 A dual result

There is also a dual form of the LEARNDISJUNCTION algorithm, in which disjunction is replaced by conjunction, and negative examples are replaced with positive examples. Like LEARNDISJUNCTION this algorithm, which we will call LEARNCONJUNCTION, learns from equivalence queries: for the sake of completeness we give the code for the algorithm below.

Algorithm LEARNCONJUNCTION($\mathcal{P}$):

1. Let $\mathcal{P}' = \mathcal{P}$ and let the current hypothesis be $C = \bigwedge_{P_{i_j} \in \mathcal{P}'} P_{i_j}$
2. Ask the equivalence oracle if the current hypothesis is equivalent to the target concept.
   (a) If the answer is "equivalent" then return the current hypothesis C.
   (b) Otherwise, the oracle returns a positive counterexample e; in this case, remove from $\mathcal{P}'$ all P' that do not cover e, replace C with $C = \bigwedge_{P_{i_j} \in \mathcal{P}'} P_{i_j}$ and return to Step 2.

In applying this technique in an ILP context, a natural starting point is to consider the algorithm LEARNCONJUNCTION($\mathcal{P}^k$), which is the dual form of the algorithm developed in the previous section. The first question to consider is the issue of the hypotheses of LEARNCONJUNCTION($\mathcal{P}^k$): is it in general possible to express the semantic conjunction of clauses from $\mathcal{P}^k$ as a logic program?

The answer to this question is affirmative. Let $A{\leftarrow}B_1, \ldots, A{\leftarrow}B_r$ be r clauses from $\mathcal{P}^k$. Without loss of generality one can assume that the heads of these clauses are identical, not just simply alphabetic variants. To form a program C such that

$$M(C) = M(\{A{\leftarrow}B_1\}) \cap \ldots \cap M(\{A{\leftarrow}B_r\})$$

simply rename the free variables in each B_i so that they are all disjoint, and let C be the clause

$$A{\leftarrow}B_1 \wedge \ldots \wedge B_r$$

This leads to the following result. (For a detailed proof, see [Cohen, 1994b].)

Theorem 16 *Let $\mathcal{L}^{k\text{-LENGTH-DUAL}}[B]$ be the language of logic programs C such that*

$$M(C) = M(\{P_1\}) \cap \ldots \cap M(\{P_r\})$$

where $P_1,\ldots,P_r$ are clauses in $\mathcal{P}^k$.

For all a and k, the language family $\mathcal{L}^{k\text{-LENGTH-DUAL}}[a\text{-}\mathcal{B}]$ is uniformly learnable from equivalence queries.

4.2.4 Local clauses

The importance of Theorem 16 is a somewhat hard to assess because the characterization of the language $\mathcal{L}^{k\text{-LENGTH-DUAL}}$ is somewhat hard to understand. It turns out, however, that there is a simpler characterization of the language $\mathcal{L}^{k\text{-LENGTH-DUAL}}$.

Let us first note that if the background theory B contains an *equality predicate* (that is, a predicate *equal(X,Y)* that is defined to be true precisely when $X = Y$) then the syntactic restriction that all variables in the head be distinct does not semantically restrict the language.[5] Further an equality predicate can be added to any background theory B with only a polynomial increase in size. Henceforth we will assume that B contains an equality predicate.

Now, let X_1 and X_2 be two free variables appearing in a clause $A \leftarrow B_1 \wedge \ldots \wedge B_r$. We say that X_1 *touches* X_2 if they appear in the same literal, and that X_1 *influences* X_2 if it either touches X_2, or if it touches some variables X_3 that influences X_2. The *locale* of a variable X is the set of literals $\{B_{i_1}, \ldots, B_{i_l}\}$ that contain either X, or some variable influenced by X. Finally, let the *locality* of a clause be the cardinality of the largest locale of any free variable in that clause, and let $\mathcal{L}^{k\text{-LOCAL}}$ denote the language of nonrecursive clauses with locality k or less.

It can be shown that if a background theory B contains an equality predicate, then $\mathcal{L}^{k\text{-LOCAL}}[B] = \mathcal{L}^{k\text{-LENGTH-DUAL}}[B]$. Hence:

Corollary 4 *Let $\mathcal{L}^{k\text{-LOCAL}}$ be the language of logic programs consisting of single nonrecursive k-local clause. For all a and k, the language family $\mathcal{L}^{k\text{-LOCAL}}[a\text{-}\mathcal{B}]$ is uniformly learnable from equivalence queries.*

Discussion. As before, this result implies that the language family is also uniformly pac-learnable.

The learnability of k-local clauses was first proved in [Cohen, 1993b], but was not noted as the dual of the case of length-k clauses.

The algorithm LEARNCONJUNCTION is one example of a *bottom-up* or *general-to-specific* learning algorithm; such algorithms are common, both in learning theory and in practise. Here we have described a special-purpose bottom-up learning algorithm; however, in ILP contexts bottom-up algorithms are often implemented using the *least-general-generalization* (lgg) algorithms introduced by Plotkin [Plotkin, 1969]. Use of lgg (or its extensions *e.g.*

[5] To see that this is true, observer that a clause like *p(X,X)*$\leftarrow$*B* can be equivalently written *p(X_1,X_2)*$\leftarrow$*equal(X_1,X_2)*$\wedge$*B*.

[Buntine, 1988]) usually leads to algorithms that extend to clauses with arbitrary function symbols, but which are harder to analyze from a learning theory prospective.

The language $\mathcal{L}^{k\text{-LOCAL}}$ is more powerful than it would at first appear, as is discussed in the section below.

5 Summary of learning theory results

5.1 Technical background

In the previous sections of the paper, we have reviewed several proof techniques from computational learning theory that have been applied to ILP problems, and presented a number of the more interesting formal results. In this section, we will summarize some other relevant results in which polynomial learning models have been applied to inductive logic programming problems. We will focus on the traditional models of pac-learnability and learnability from equivalence queries; other formal models are discussed later, in Section 6.

Recall that a-$\mathcal{B}$ denotes the class of all ground Datalog databases (that is, conjunctions of ground atoms) built from predicates of arity at most a. For clarity we will focus our discussion on such background theories, although some of the results we will discuss extend to more general forms of background theories [Page and Frisch, 1991, Page and Frisch, 1992].

Unless otherwise specified, we will also assume a "Datalog" representation for examples and constants. When target concepts are non-recursive, we will assume that concepts contain no function symbols, and that examples are ground atoms of size n_e containing constants but no other function symbols. For technical reasons, we will assume a slightly more complex sort of example when target concepts are recursive: in particular we will assume that examples are pairs (e, D) where e is an atom, D is a set of n_e atoms, and all atoms in $D \cup \{e\}$ are ground, of arity a or less, and contain no function symbols. [6]

To our knowledge, all of the positive results in this area hold in the stronger model of learning from equivalence queries as well as the weaker model of pac-learning; therefore, rather than saying *uniformly learnable (predictable) by equivalence queries* and *uniformly pac-learnable* (*pac-predictable*), we will simply say *learnable* (*predictable*) in stating the results.

Some other technical terms we will require are *depth*, first defined in Section 3.1; *locality*, first defined in Section 4.2.4; and *determinacy*, which we now define. An ordered definite clause $A \leftarrow B_1 \wedge \ldots \wedge B_r$ is *determinate* relative to a background theory if and only if the

[6] For instance, an example of a recursive program for list membership might be represented as the goal atom $e =$*member(b,list_ab)* together with the description

$D = \{$ head(list_ab,a), tail(list_ab,list_b),
head(list_b,b), tail(list_b,nil) $\}$

A program P is considered to classify an example (e, D) as positive iff $P \wedge B \wedge D \vdash e$ and the size of an instance (e, D) is simply the cardinality of D. There is a close relationship between this formalization of instances and "flattening" [Rouveirol, 1994]. The rationale for adopting examples of this type is that most practical ILP systems are able to handle high-arity literals in the head by not the body of target clauses. Thus in analyzing learning problems in which the target relation is recursive—and hence must appear in both the head and the body of hypothesized clauses—it is less appropriate to encode large examples as high-arity atoms.

following is true. For any literal B_i, $1 \leq i \leq r$, and any grounding substitution θ for the variables in A, $B_1, ..., B_{i-1}$, where $B_1\theta, ..., B_{i-1}\theta$ are provable from the background theory, there exists at most one substitution σ such that: $\theta\sigma$ is a grounding substitution for B_i, and $B_i\theta\sigma$ is provable from the background theory. An ordered definite clause is ij-determinate just if it is determinate, has depth at most i, and has body literals built from predicates of arity at most j.

Given these preliminaries, we can now list the major results.

5.2 Positive results for non-recursive program

For non-recursive concepts, the language families $\mathcal{L}^{k\text{-LOCAL}}[a\text{-}\mathcal{B}]$ and $\mathcal{L}^{j\text{-LENGTH}}[a\text{-}\mathcal{B}]$ (as defined in Section 4) are both learnable; further, no language strictly more expressive than these languages is known to be learnable. The result for $\mathcal{L}^{k\text{-LOCAL}}$ subsumes a number of previous results. Specifically, this language can be shown to be more strictly expressive than the language of nonrecursive constant-depth determinate (aka "ij-determinate") clauses, which is also learnable. The basic idea behind this result is that a determinate but highly non-local clause such as

$$p(X) \leftarrow q(X,Y) \wedge r_1(Y) \wedge r_2(Y) \wedge \ldots \wedge r_n(Y)$$

can always be rewritten as a slightly larger highly local clause, such as

$$p(X) \leftarrow q(X,Y_1) \wedge r_1(Y_1) \wedge q(X,Y_2) \wedge r_2(Y) \wedge \ldots \wedge q(X,Y_n) \wedge r_n(Y)$$

Details of this proof can be found in [Cohen, 1994b]. The language of ij-determinate clauses is analyzed in [Džeroski *et al.*, 1992] and is shown in turn to generalize several other learnable languages, including the class of all concepts consisting of a single atom, and the class of all concepts consisting of a single non-recursive definite clause such that every variable in the body also appears in the head. (Such a clause is also called a *constrained atom.*) We note, however, that unlike the result for $\mathcal{L}^{k\text{-LOCAL}}$ clauses, the results for constrained and unconstrained atoms have also been extended to function symbols and arbitrary background knowledge [Page and Frisch, 1991, Page and Frisch, 1992]. [7]

If pac-predictability rather than pac-learnability is considered, then the language $\mathcal{L}^{k\text{-LOCAL}}$ can be extended further to the language of programs that contain at most c non-recursive k-local clauses (for constant c).

Also, certain classes of clauses that allow bounded indeterminacy are pac-predictable, but not pac-learnable [Cohen, 1993b].

5.3 Negative results for non-recursive programs

A number of negative learnability results have also been obtained for programs without recursion, some of which were presented in this paper. Indeterminate non-recursive clauses of constant depth are not pac-predictable because they are evaluation-hard, as shown in

[7]More precisely, certain kinds of consequences of the background theory must be efficiently decidable, but no restrictions beyond this are imposed.

Theorem 2. Also, k-free clauses are not pac-predictable unless DNF is pac-predictable, as shown in Theorem 4. It is also known that log-depth determinate non-recursive clauses are not pac-predictable, if certain cryptographic assumptions are made [Cohen, 1993a].

Some languages are known to be pac-predictable, but not pac-learnable. In addition to the languages mentioned above, the class of concepts consisting of a single typed (sorted) atom is not learnable, although it is pac-predictable [Page and Frisch, 1992].

5.4 Recursive progams

In Section 4.1 we showed that the language $H_{2,k}$ (which is not restricted to Datalog, but allows only unary functions) is predictable; further the language needed for prediction is also a simple, understandable logic programming language. In addition, the language $H_{2,1}$ is learnable, as well as being predictable.

Another positive result for recursive languages is given in [Cohen, 1993c]. Here it is shown that a single linear "closed" recursive constant-depth determinate clause is pac-learnable.[8] This result can be extended to programs containing two clauses (under the same restrictions) if a special "basecase oracle" is allowed. More recent work has also extended this result to k-ary recursive clauses [Cohen, 1994a].

However, recursion is a very powerful construct, and many languages allowing even limited use of recursion are hard to learn. It has been shown that the language of single recursive constant-depth determinate clauses is not pac-predictable, under cryptographic assumptions [Cohen, 1993a]. Furthermore, the languages $\mathcal{L}^{k\text{-LENGTH}}$ and $\mathcal{L}^{k\text{-LOCAL}}$ are not pac-predictable if even linear recursion is allowed[9] [Cohen, 1993b, Cohen, 1994a]. Other recent results have described other highly constrained recursive languages that are hard to learn: for example, the language of linearly recursive two-clause constant-depth determinate programs is pac-predictable only if DNF is pac-predictable [Cohen, 1994a].

Finally, Arimura, Ishizaka, and Shinohara [1992] and Nienhuys-Cheng and Polman [1994] have given some positive results on the learnability of recursive programs with function symbols and no background knowledge. They show that certain classes of programs are learnable in a relatively weak learning model in which the computational complexity of the learning algorithm is ignored. All of their positive results are for programs with a constant number of clauses, each containing a constant number of literals.

6 Conclusions

In this paper, we have explained and illustrated the primary techniques for proving positive and negative learnability results in ILP, and also summarized existing results on ILP problems in polynomial learnability models. We will now discuss briefly our view of the implications of these results.

[8] A recursive program is "closed" if no recursive literal contains any output variables.

[9] A program is *linearly recursive* if the body of every clause contains at most recursive literal, where a recursive literal is one that has the same principle functor and arity as the head of the clause.

Results on polynomial learnability in general have been disappointing. Even formulae of propositional logic are not even predictable, nor are finite state automata.[10] Propositional DNF formulae still are not known to be predictable or pac-learnable, but are known not to be learnable by equivalence queries [Angluin, 1989].

In spite of these results, ILP is based in part on the claim that often, in practical machine learning problems, propositional logic and finite state automata do not provide appropriate expressivity. However, strictly increasing the expressivity will lead only to additional negative results for prediction. Therefore, if one wants strong positive learnability results for ILP, one must hope that various classes of restricted logic programs will provide efficiently learnable concept classes that are very different from propositional formulae, propositional DNF formulae, or finite state automata.

Existing formal results show that such restricted-form logic programs are quite difficult to find. Specifically, the kinds of logic programs for which we have positive results (e.g., constrained atoms, i, j-determinate clauses, k-local clauses, or conjunctions of some bounded number of these) can be learned (predicted) by transformations to propositional logic. Further, those languages that are not learned by reduction to propositional form (e.g., $H_{2,k}$) are usually restricted enough to be learned or predicted by techniques related to enumeration.

More importantly, the various natural relaxations of these learnable or predictable classes are closely related to propositional DNF, in such a way that if these classes can be predicted then so can DNF. (One exception is an extension of $H_{2,1}$ that allows arbitrarily many clauses built from unary functions and predicates. Nevertheless, this extension is closely related to finite state automata, although the size relationship remains to be determined [Frazier and Page, 1993b].) Thus learnability in ILP is constrained largely by the limits on learnability within propositional logic.

That ILP should be bound by the limits on learning propositional formulae undermines the motivation for ILP. Why extend the representation when we must immediately restrict it again in order to have efficient, effective learning?

There are in fact at least three responses to the preceding question, each of which suggests a significant topic for further research. We now consider each response in turn.

The first response is that we are not studying learnability within ILP in the only possible way, and other settings might yield positive results that go beyond results for propositional logic. Recall that for all the results described in this paper, examples are formulae which are classified *positive* just if they are entailed by the target concept. However, in studies of learnability within propositional logic, the examples are usually truth assignments, or models, which are *positive* just if they satisfy the target. Therefore, a second natural approach to studying learnability within ILP is to use models, rather than formulae, as examples. Of course in general the models of a logic program may be infinite. Nevertheless, for Datalog the models are finite; moreover, DeRaedt and Džeroski [DeRaedt and Džeroski, 1994] describe a reasonable way to use finite approximations of infinite models when the Datalog restriction is removed. DeRaedt and Džeroski have established a promising result within this alternative setting, although again the result is based on a transformation to propositional logic. It seems likely that this alternative setting will provide strictly stronger positive learnability

[10]These results are based on cryptographic assumptions [Kearns and Valiant, 1989, Kearns and Valiant, 1994].

results for ILP, because for a given language:

- a model is (in some ways, at least) more informative than an entailed atom, and
- models usually are substantially larger (often exponentially larger) than atoms over the same language. This allows a polynomial-time learning algorithm more processing time.

The second response is that the equivalence query and pac-learnability models themselves are not necessarily the only reasonable models of polynomial learnability, and other models with more realistic expectations might yield stronger positive results for ILP. What properties should such alternative models have? One property that has been advocated for several years is the use of an *average case accuracy requirement* [Buntine, 1990, Haussler *et al.*, 1991, Haussler *et al.*, 1994], rather than the worst case accuracy requirement of pac-learning or the total-accuracy requirement of the equivalence query model. A second, related property is the use of an average case time complexity requirement rather than the worst-case time-complexity requirement of these standard models. Of course, to speak of average-case performance, one needs a distribution over inputs, and for inductive learning problems this means a distribution (or distributions) over target hypotheses as well as examples. In addition, it may be extreme to expect that a learner knows exactly the distributions from which examples and target hypotheses are drawn. Much more reasonable is the expectation that the learner has a rough idea of the underlying distributions, or knows some family of potential distributions and is able to perform well with each. One model with these desirable properties that has been proposed recently is called *Universal Learnability*, or *U-learnability* [Muggleton and Page, 1994]. It has been shown that under very strong distributional assumptions, the general class of linear-time (or alternatively, quadratic-time, cubic-time, etc.) logic programs is learnable in this model. Hence, at least with some assumptions U-learnability provides strong positive learnability results for ILP. Nevertheless, it remains to be seen whether strong positive results can be obtained with significantly weaker distributional assumptions. The eventual utility of this model depends on whether it is possible to prove positive results for families of distributions that actually arise in practical problems. Therefore, research into the U-learnability of various restricted classes of logic programs under reasonable distributional assumptions is an inviting area for further ILP research.

A third possible response is that the setting and learnability models used in this paper are the most appropriate for ILP—that ILP is indeed bound by the limits on learning propositional formulae in these models. However, even if this is true, it may well be that ILP provides other kinds of advantages over using pure propositional learners. One such advantage is the "automatic" generation of new propositions that the typical user of a propositional learner might not consider creating.

This behavior is best described by example. Consider the mutagenesis domain [Srinivasan *et al.*, 1994], in which various potential pharmaceutical chemicals are represented simply by their atom and bond structure, and the goal is to predict the mutagenicity (related to carcinogenicity) of chemicals based on structural properties. In preparing the data for a propositional learner, the most natural propositions identify which types of atoms and bonds appear in a given chemical. (Atom types might be more informative than simply *carbon*, e.g., *alpiphatic*—an aliphatic carbon atom—or *aromatic-6*—a carbon atom in a 6-membered aromatic ring.) In ILP, the natural representation is to provide a background theory that uses predicates *atom* and *bond* to describe the structures of various chemicals, and then to use the predicate *mutagenic* to build examples specifying which chemicals are mutagenic. Now consider applying the learning algorithm for $\mathcal{L}^{k\text{-LOCAL}}$ to this problem, represented in ILP form. Where X denotes the chemical in question, the algorithm generates *locales*, or propositions, such as

$$atom(X, A, aliphatic), atom(X, B, aromatic\text{-}6), bond(X, A, B, single)$$

specifying the existence in chemical X of an aliphatic carbon atom attached by a single bond to a ring carbon.[11] Such a proposition cannot be built by conjoining or disjoining simpler propositions, but requires the use of shared variables X, A, and B as allowed by a first-order representation.

Thus even though the foundation of the learning algorithm is propositional, starting with a first-order representation provides more complex propositions automatically; furthermore, it provides a frame of reference for describing properties of and relationships among propositions. This may be of importance in determining which features are important for learning. We note that although it is widely accepted that appropriate choice of features is of central importance in practical learning problems, little formal progress has been made on this problem.

We close by noting that in computational learning theory it is also common to use additional types of queries, such as *membership queries*, which ask whether a particular example is *positive* or *negative*, and *subset queries*, which ask (in one query) whether all the examples in a particular set of examples are positive. A number of additional positive results within propositional logic can be proven when such queries are allowed (see for example [Angluin *et al.*, 1992, Aizenstein, 1992]). Nevertheless, the resulting algorithms do not provide corresponding positive results for ILP. (The kinds of "ILP to propositional logic" transformations usually employed do not work in the context of such additional types of queries.) In other words, the relationship between learning in propositional logic and first-order logic changes when such additional queries are introduced. Similarly, much more powerful positive results for ILP can be proven when subset queries are allowed [Haussler, 1989, Page, 1993]. These results appear to have no analogues within propositional logic. The use of membership, subset, or other kinds of queries in ILP is another important area for further research.

References

[Aizenstein, 1992] H. Aizenstein. *On the Learnability of Disjunctive Normal Form Formulas and Decision Trees.* PhD thesis, University of Illinois at Urbana-Champaign, 1992.

[Angluin *et al.*, 1992] D. Angluin, M. Frazier, and L. Pitt. Learning conjunctions of horn clauses. *Machine Learning*, 9:147–164, 1992.

[11] This is in fact a useful combination of literals found by the ILP program *Progol* in this domain. The literals have been simplified somewhat in this presentation.

[Angluin, 1988] D. Angluin. Queries and concept learning. *Machine Learning*, 2(4), 1988.

[Angluin, 1989] D. Angluin. Equivalence queries and approximate fingerprints. In *Proceedings of the 1989 Workshop on Computational Learning Theory*, Santa Cruz, California, 1989.

[Arimura *et al.*, 1992] H. Arimura, H. Ishizaka, and T. Shinohara. Polynomial time inference of a subclass of context-free transformations. In *Proceedings of the Fifth Workshop on Computational Learning Theory (COLT-92)*, pages 136–143, New York, 1992. The Association for Computing Machinery.

[Blumer *et al.*, 1989] A. Blumer, A. Ehrenfeucht, D. Haussler, and M. Warmuth. Classifying learnable concepts with the Vapnik-Chervonenkis dimension. *Journal of the Association for Computing Machinery*, 36(4):929–965, 1989.

[Buntine, 1988] W. Buntine. Generalized subsumption and its application to induction and redundancy. *Artificial Intelligence*, 36(2):149–176, 1988.

[Buntine, 1990] W. Buntine. *A Theory of Learning Classification Rules*. PhD thesis, School of Computing Science, University of Technology, Sydney, 1990.

[Cohen, 1993a] W. W. Cohen. Cryptographic limitations on learning one-clause logic programs. In *Proceedings of the Tenth National Conference on Artificial Intelligence*, Washington, D.C., 1993.

[Cohen, 1993b] W. W. Cohen. Learnability of restricted logic programs. In *Proceedings of the Third International Workshop on Inductive Logic Programming*, Bled, Slovenia, 1993.

[Cohen, 1993c] W. W. Cohen. A pac-learning algorithm for a restricted class of recursive logic programs. In *Proceedings of the Tenth National Conference on Artificial Intelligence*, Washington, D.C., 1993.

[Cohen, 1993d] W. W. Cohen. Pac-learning non-recursive prolog clauses. To appear *Artificial Intelligence*, 1993.

[Cohen, 1994a] W. W. Cohen. The pac-learnability of recursive logic programs. In preparation, 1994.

[Cohen, 1994b] W. W. Cohen. Pac-learning nondeterminate clauses. In *Proceedings of the Eleventh National Conference on Artificial Intelligence*, Seattle, WA, 1994.

[Cohen, 1994c] W. W. Cohen. Recovering software specifications with inductive logic programming. In *Proceedings of the Eleventh National Conference on Artificial Intelligence*, Seattle, WA, 1994.

[DeRaedt and Džeroski, 1994] L. DeRaedt and S. Džeroski. First-order jk-clausal theories are PAC-Learnable. *Artificial Intelligence*, 70:375–392, 1994.

[Džeroski *et al.*, 1992] S. Džeroski, S. Muggleton, and S. Russell. Pac-learnability of determinate logic programs. In *Proceedings of the 1992 Workshop on Computational Learning Theory*, Pittsburgh, Pennsylvania, 1992.

[Frazier and Page, 1993a] M. Frazier and C. D. Page. Learnability in inductive logic programming: Some basic results and techniques. In *Proceedings of the 11th National Conference on Artificial Intelligence (AAAI-93)*, Menlo Park, CA, 1993. AAAI Press.

[Frazier and Page, 1993b] M. Frazier and C. D. Page. Learnability of recursive, non-determinate theories: Some basic results and techniques. In *Proceedings of the Third International Workshop on Inductive Logic Programming*, pages 103–126, Ljubljana, Slovenia, 1993. J. Stefan Institute Technical Report IJS-DP-6707.

[Frazier, 1994] M. Frazier. *Matters Horn and Other Features in the Computational Learning Theory Landscape: The Notion of Membership.* PhD thesis, University of Illinois at Urbana-Champaign, 1994.

[Haussler *et al.*, 1991] D. Haussler, M. Kearns, and R. Schapire. Bounds on the sample complexity of bayesian learning using information theory and vc dimension. In *Proceedings of the Fourth Annual Workshop on Computational Learning Theory*, pages 61–74, San Mateo, CA, 1991. Morgan Kaufmann.

[Haussler *et al.*, 1994] D. Haussler, M. Kearns, and R. Schapire. Bounds on the sample complexity of bayesian learning using information theory and vc dimension. *Machine Learning*, 14(1):83–113, January 1994.

[Haussler, 1989] D. Haussler. Learning conjunctive concepts in structural domains. *Machine Learning*, 4(1), 1989.

[Helmbold and Warmuth, 1992] D. Helmbold and M. Warmuth. Some weak learning results. In *Proc. 5th Annu. Workshop on Comput. Learning Theory*, pages 399–412. ACM Press, New York, NY, 1992.

[Kearns and Valiant, 1989] M. Kearns and L. Valiant. Cryptographic limitations on learning Boolean formulae and finite automata. In *21th Annual Symposium on the Theory of Computing*. ACM Press, 1989.

[Kearns and Valiant, 1994] M. Kearns and L. Valiant. Cryptographic limitations on learning boolean formulae and finite automata. *Journal of the Association for Computing Machinery*, 41(1):67–95, 1994.

[King *et al.*, 1992] R. D. King, S. Muggleton, R. A. Lewis, and M. J. E. Sternberg. Drug design by machine learning: the use of inductive logic programming to model the structure-activity relationships of trimethoprim analogues binding to dihydrofolate reductase. *Proceedings of the National Academy of Science*, 89, 1992.

[Lassez *et al.*, 1988] J-L. Lassez, M. J. Maher, and K. Marriott. Unification revisited. In Jack Minker, editor, *Foundations of Deductive Databases and Logic Programming*, chapter 15, pages 587–625. Morgan Kaufmann Publishers, 1988.

[Minton, 1994] S. Minton. Small is beautiful: a brute-force approach to finding first-order formulas. In *Proceedings of the Twelfth National Conference on Artificial Intelligence*, Seattle, Washington, 1994. MIT Press.

[Muggleton and Page, 1994] S. H. Muggleton and C. D. Page. A learnability model for universal representations. In S. Wrobel, editor, *Proceedings of the Fourth International Workshop on Inductive Logic Programming*, pages 139–160, Sankt Augustin, Germany, 1994. GMD. Published as GMD-Studien Nr. 237.

[Muggleton, 1994a] S. Muggleton. Bayesian inductive logic programming. In *Proceedings of the Eleventh International Conference on Machine Learning (ML-94)*, pages 371–379, San Francisco, 1994. Morgan Kaufmann.

[Muggleton, 1994b] S. Muggleton. Bayesian inductive logic programming. In *Proceedings of the Seventh Annual ACM Conference on Computational Learning Theory (COLT-94)*, pages 3–11, New York, 1994. The Association for Computing Machinery.

[Nienhuys-Cheng and Polman, 1994] S.H. Nienhuys-Cheng and M. Polman. Sample pac-learnability in model inference. In *Machine Learning: ECML-94*, Catania, Italy, 1994. Springer-Verlag. Lecture notes in Computer Science # 784.

[Page and Frisch, 1991] C. D. Page and A. M. Frisch. Generalizing atoms in constraint logic. In J. Allen, R. Fikes, and E. Sandewall, editors, *Second Int. Conf. on Principles of Knowledge Representation and Reasoning*, San Mateo, CA, April 1991. Morgan Kaufmann.

[Page and Frisch, 1992] C. D. Page and A. M. Frisch. Generalization and learnability: A study of constrained atoms. In S. H. Muggleton, editor, *Inductive Logic Programming*, pages 29–61. Academic Press, London, 1992.

[Page, 1993] C. D. Page. *Anti-unification in constraint logics: foundations and applications to learnability in first-order logic, to speed-up learning, and to deduction*. PhD thesis, University of Illinois at Urbana-Champaign, 1993.

[Pitt and Valiant, 1988] L. Pitt and L. Valiant. Computational limitations on learning from examples. *Journal of the ACM*, 35(4):965–984, 1988.

[Pitt and Warmuth, 1990] L. Pitt and M. Warmuth. Prediction-preserving reducibility. *Journal of Computer and System Sciences*, 41:430–467, 1990.

[Plotkin, 1969] G. D. Plotkin. A note on inductive generalization. In *Machine Intelligence 5*, pages 153–163. Edinburgh University Press, 1969.

[Quinlan, 1990] J. R. Quinlan. Learning logical definitions from relations. *Machine Learning*, 5(3), 1990.

[Riddle *et al.*, 1994] P. Riddle, R. Segal, and O. Etzioni. Representation design and brute-force induction in a Boeing manufacturing domain. *Applied Artificial Intelligence*, 8:125–147, 1994.

[Rouveirol, 1994] C. Rouveirol. Flattening and saturation: two representation changes for generalization. *Machine Learning*, 14(2), 1994.

[Schapire, 1990] R. Schapire. The strength of weak learnability. *Machine Learning*, 5(2), 1990.

[Srinivasan *et al.*, 1994] A. Srinivasan, S. Muggleton, R. King, and M. Sternberg. Mutagenesis: Ilp experiments in a non-determinate biological domain. In S. Wrobel, editor, *Proceedings of the Fourth International Workshop on Inductive Logic Programming*, pages 217–232, Sankt Augustin, Germany, 1994. GMD. Published as GMD-Studien Nr. 237.

[Valiant, 1984] L. G. Valiant. A theory of the learnable. *Communications of the ACM*, 27(11), November 1984.