Learnability of Restricted Logic Programs

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

December 28, 1993

Abstract

An active area of research in machine learning is learning logic programs from examples; this subarea is sometimes called *inductive logic programming*. This paper investigates this problem formally, using as our primary technical tool the method of *prediction-preserving reducibilities* and the model of *polynomial predictability* introduced by Pitt and Warmuth [1990]. We focus on the learnability of various generalizations of the language of constant-depth determinate clauses, which is used by several practical learning systems. We show that a single determinate clause of logarithmic depth is not polynomially predictable, under cryptographic assumptions. We then establish a close connection between the learnability of a single clause with k "free" variables and the learnability of DNF; a close connection is also shown between the learnability of a single clauses with bounded indeterminacy and the learnability of k-term DNF, leading to a prediction algorithm for a class of clauses with bounded indeterminacy. We then define two new classes of logic programs that allow indeterminacy, but are easily pac-learnable. Finally we present a series of results showing that allowing recursion makes some simple logic programming languages hard to learn against an arbitrary distribution: in particular, one-clause constant depth determinate programs with arbitrary recursion are hard to learn, as are multi-clause constant depth determinate programs with linear recursion.

1 Introduction

Most machine learning systems in current use represent concepts using some restricted form of propositional logic, and represent examples as a vector of attribute values. Recently, however, there has been an increasing amount of research on the problem of extending the representational power of machine learning systems by learning concepts expressed in restricted first-order logics. While some of this research has considered the use of special-purpose logics such as description logics as a representation for concepts and examples [Cohen and Hirsh, 1992; Vilain *et al.*, 1990; Kietz and Morik, 1991] most researchers have used standard first-order logic as a representation language; in particular, most have used restricted subsets of the Prolog programming language to represent concepts [Cohen, 1992; Muggleton and Feng, 1992; Pazzani and Kibler, 1992; Quinlan, 1990; Muggleton, 1992a]. The term *inductive logic programming (ILP)* has been used to describe this growing body of research.

One advantage of basing learning systems on Prolog is that its semantics and complexity are mathematically well-understood. This offers some hope that learning systems based on it can also be rigorously analyzed; several formal results have in fact been obtained. For example, Frisch and Page [1991] have shown that a single *constrained atom* can be learned in Valiant's [1984] model of probably approximately correct (pac) learning: a constrained atom is a Horn clause in which every variable appearing in the body of the clause also appears in the head. A stronger result is due to Džeroski, Muggleton and Russell [1992]. They show that a single ij-*determinate* clause is learnable in the pac-model, and that a non-recursive logic program containing k such clauses is learnable against any "simple" distribution: in an ij-determinate clause, new variables may appear in the body, as long as there is a unique binding for these variables, the arity of literals in the body is bounded by a constant j, and the *depth* of the clause is bounded by a constant i.[1] The ij-determinacy restriction was first used in the GOLEM system [Muggleton and Feng, 1992] and has since been incorporated in several other practical learning systems [Cohen, 1993a; Lavrač and Džeroski, 1992; Quinlan, 1991].

Some very recent work [Kietz, 1993] shows that a single clause is *not* pac-learnable if the ij-determinacy condition does not hold; specifically, it is shown that neither the language of indeterminate clauses of fixed depth nor the language of determinate clauses of arbitrary depth is pac-learnable. The proof of these facts is based on showing that there are sets of examples such that finding a single clause in the language consistent with the examples is NP-hard (or worse). These negative results are of limited practical importance because they assume that the learner is required to output a *single* clause consistent with all of the examples. Most ILP learning systems, however, learn a set of clauses, and the results do *not* show that learning using this more expressive representation is intractable. In short, the intractibilities arise because it is difficult to encode the answer into the representation language chosen for hypotheses, not because the learning problem itself is intrinsically difficult. Such negative learnability results are sometimes called *representation-dependent*.[2]

[1] This class of clauses is defined more rigorously in Section 3.

[2] The prototypical example of a learning problem which is hard in a representation-dependent setting but not in a broader setting is learning k-term DNF. Pac-learning k-term DNF is NP-hard if the hypotheses of

This paper seeks to expand the theoretical foundations of ILP by investigating the learnability of restricted logic programs in the absence of representational restrictions. In particular, we consider relaxing the condition of ij-determinacy in various ways, and see what effect this has on learnability. Our principle formal tool in this paper is the method of *prediction-preserving reducibilities* [Pitt and Warmuth, 1990]. This technique allows one to analyze learnability by relating the difficulty of two learning problems. Negative results are obtained by showing that a learning problem is as hard as breaking a (presumably) secure cryptographic system. Positive results are obtained by showing that a learning problem is no harder than some learning problem with a known solution. Use of this method requires introducing a slightly weaker notion of learning than pac-learning; in particular learning algorithms are allowed to generate hypotheses in any language. (Learning algorithms that generate hypotheses in an arbitrary language are sometimes called *prediction algorithms* or *approximation algorithms*.) The advantage of the method is that results are *not* dependent on any particular representational choices. An additional advantage is that applying this method often leads to at least a partial characterization of the expressive power of the language being analyzed.

More specifically, in this paper, we will first investigate the learnability of determinate clauses of log depth, rather than constant depth; we show, via a reduction from log-depth circuits, that this language is not pac-learnable, regardless of the representation language. We then investigate the effect of allowing indeterminacy in a clause, and show that clauses with k "free" variables are as hard to learn as DNF. A class of clauses with bounded indeterminacy is defined and shown to be predictable; however, known prediction algorithms are exponential in the amount of indeterminacy, and pac-learning such clauses can be difficult. Thus we turn to other syntactic restrictions on Horn clauses; we show that bounding either the length or the "locality" of a clause allows pac-learnability even if an arbitrary amount of indeterminacy is allowed. We next turn to the question of allowing recursion in learnable languages, and show that allowing recursion makes some very simple languages hard to pac-learn against an arbitrary distribution, again regardless of the representation language. Finally, we summarize our results, and conclude.

2 Preliminaries

2.1 Learning models

Formal analysis of learnability requires an explicit and formal model of what if means for a language to be "efficiently learnable." In this section, we will describe our basic model of learnability.

Some basic terminology is first necessary. Let X be a set, called the *domain*. Define a *concept* C over X to be a representation of some subset of X, and a *language* $\mathcal{L}$ to be a set of concepts. Associated with X and $\mathcal{L}$ are two *size complexity measures*. We will write the size complexity of some $C \in \mathcal{L}$ or $e \in X$ as $\|C\|$ or $\|e\|$, and we will assume that this measure is polynomially related to the number of bits needed to represent C or e. We use the

the learning system must be k-term DNF; however it is tractable if hypotheses can be expressed in the richer language of k-CNF.

notation X_n (respectively $\mathcal{L}_n$) to stand for the set of all elements of X (respectively $\mathcal{L}$) of size complexity no greater than n. In this paper, we will be rather casual about the distinction between a concept and the set it represents, since there is seldom any risk of confusion.

For example, if X is the domain of binary vectors, interpreted as assignments to boolean variables, and $\mathcal{L}_{\mathrm{DNF}}$ is the language of boolean formula in disjunctive normal form, one might measure the complexity of a vector $e \in X$ as the length of the vector, and measure the complexity of a formula C by the number of literals in C.

In most formalizations, the goal of learning is to find a good approximation to any target concept $C \in \mathcal{L}_{n_t}$ over the domain X_{n_e} using resources (e.g. time and sample size) polynomial in n_t, the complexity of the concept to be learned, and n_e, the size of objects in the domain. 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* K: the task of the learner is then to find a logic program P such that P, together with K, is a good model of the data. We thus introduce the following notation and terminology. If $\mathcal{L}$ is some language of logic programs[3] and K is a logic program, then $\mathcal{L}[K]$ denotes the set of all pairs of the form (P, K) such that $P \in \mathcal{L}$: each such pair represents the set of all atoms e such that $P \wedge K \vdash e$. If $\mathcal{K}$ is a set of background theories, then the *family of languages* $\mathcal{L}[\mathcal{K}]$ represents the set of all languages $\mathcal{L}[K]$ where $K \in \mathcal{K}$. We will consider $\mathcal{L}[\mathcal{K}]$ to be "efficiently learnable" only when every $\mathcal{L}[K] \in \mathcal{L}[\mathcal{K}]$ is "efficiently learnable".

For example, if K contains the usual definition of *adjacent/2*, and $\mathcal{L}_{CON}$ is the language of logic programs containing only "constrained atoms", then $\mathcal{L}[K]$ would include the program

illegal(A,B,C,D,E,F) $\leftarrow$adjacent(A,E) $\wedge$ adjacent(B,D)
illegal(A,B,C,D,C,F) $\leftarrow$
illegal(A,B,C,D,E,D) $\leftarrow$

which would also have the expected semantics.

We can now complete the definition of our learning model. An *example of* C is a pair (x, b) where $b = 1$ if $x \in C$ and $b = 0$ otherwise. If D is a probability distribution function, a *sample of* C *from* X *drawn according to* D is a pair of multisets S^+, S^- drawn from the domain X according to D, S^+ containing only positive examples of C, and S^- containing only negative ones. We can now define the learning model used in this paper:

Definition 1 (Polynomially predictable) *A family of languages* $\mathcal{L}[\mathcal{K}]$ *is* polynomially predictable *iff there is an algorithm LEARN and a polynomial function* $m(\frac{1}{\epsilon}, \frac{1}{\delta}, n_e, n_t, n_b)$ *so that for every* $n_t > 0$, *every* $n_e > 0$, *every* $n_b > 0$, *every background theory* K *of complexity* n_b *or less, every* $C \in \mathcal{L}_{n_t}$, *every* $\epsilon : 0 < \epsilon < 1$, *every* $\delta : 0 < \delta < 1$, *and every probability distribution function* D, *for any sample* S^+, S^- *of* C *from* X_{n_e} *drawn according to* D *containing at least* $m(\frac{1}{\epsilon}, \frac{1}{\delta}, n_e, n_t, n_b)$ *examples*

1. *LEARN, on inputs* S^+, S^-, ϵ, *and* δ, *outputs a hypothesis* H *such that*

$$Prob(D(H - C) + D(C - H) > \epsilon) < \delta$$

[3]We assume that the reader is familiar with the basic elements of logic programming; see Lloyd [1987] for the necessary background.

2. *LEARN runs in time polynomial in* $\frac{1}{\epsilon}$, $\frac{1}{\delta}$, n_e, n_t, *and* n_b, *and the number of examples; and*

3. *H can be evaluated in polynomial time.*

The polynomial function $m(\frac{1}{\epsilon}, \frac{1}{\delta}, n_e, n_t, n_b)$ *is called the sample complexity of the learning algorithm LEARN.*

The first condition in the definition merely states that the error of the hypothesis must (usually) be low, as measured against the probability distribution D from which the training examples were drawn. The second condition, together with the stipulation that the sample size is polynomial, ensures that the total running time of the learner is polynomial. The final condition simply requires that the hypothesis be useable in the very weak sense that it can be used to make predictions in polynomial time. Notice that this definition is quite strong, as it allows an adversarial choice of all the inputs of the learner.[4] Finally note that the parameters n_e, n_b, n_t all measure, in some sense, the size of the learning problem, and we are requiring the learner to be polynomial in all of these size measures; while there is some value in keeping these different measures separate, the casual reader may find it easier to consider the results in terms of a single size measure $n = n_e + n_b + n_t$.

Polynomial predictability is a slight weakening of Valiant's [1984] criterion of *pac-learnability:*

Definition 2 (Pac-learnable) *A language $\mathcal{L}$ is* pac-learnable *iff it is polynomially predictable and the hypothesis H it outputs is in $\mathcal{L}$.*

Thus if a language is pac-learnable it is predictable, but the converse need not be true. Predictability also has the property that if a language is *not* predictable, then no superset of that language is predictable; i.e., one cannot make a non-predictable language predictable by generalizing the language, only by adding additional restrictions.

2.2 Reducibilities among prediction problems

Our main analytic tool in this paper is the use of prediction-preserving reducibilities, as described by Pitt and Warmuth [1988]. This is essentially a method of showing that one language is no harder to predict than another.

Definition 3 (Reducibilities among prediction problems) *Let $\mathcal{L}_1$ be a language over domain X and $\mathcal{L}_2$ be a language over domain Y. We say that* predicting $\mathcal{L}_1$ reduces to predicting $\mathcal{L}_2$, *denoted $\mathcal{L}_1 \trianglelefteq \mathcal{L}_2$, if there is a function $f_i : X \rightarrow Y$, henceforth called the* instance mapping, *and a function $f_c : \mathcal{L}_1 \rightarrow \mathcal{L}_2$, henceforth called the* concept mapping, *so that the following all 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;*

[4]It might seem odd that we allow an adversarial choice of the background theory K; notice however, that if the background theory is such that the target concept cannot be expressed, or is only expressible by an exponentially large concept, then the learning system is not required to output an accurate hypothesis quickly.

2. *the size complexity of $f_c(C)$ is polynomial in the size complexity of C — i.e. the size of concept representations is preserved within a polynomial factor;*

3. *$f_i(x)$ can be computed in polynomial time.*

Note that f_c need not be computable; also, since f_i can be computed in polynomial time, $f_i(x)$ must also preserve size within a polynomial factor.

If predicting $\mathcal{L}_1$ reduces to predicting $\mathcal{L}_2$ and a learning algorithm for $\mathcal{L}_2$ exists, then one possible scheme for learning concepts from $\mathcal{L}_1$ would be the following. First, convert any examples of the unknown concept C_1 from the domain X to examples over the domain Y using the instance mapping f_i. If the conditions of the definition hold, then since C_1 is consistent with the original examples, the concept $f_c(C_1)$ will be consistent with their image under f_i; thus running the learning algorithm for $\mathcal{L}_2$ should produce some hypothesis H that is a good approximation of $f_c(C_1)$. Of course, if may not be possible to map H back into the original language $\mathcal{L}_1$, as computing $f_c{}^{-1}$ may be difficult or impossible. However, H can still be used to 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 $f_i(x) \in H$.

Pitt and Warmuth [1988] give a more rigorous argument that this approach leads to a prediction algorithm for $\mathcal{L}_1$, leading to the following theorem.

Theorem 1 (Pitt and Warmuth) *If $\mathcal{L}_1 \trianglelefteq \mathcal{L}_2$ and $\mathcal{L}_2$ is polynomially predictable, then $\mathcal{L}_1$ is polynomially predictable. Conversely, if $\mathcal{L}_1 \trianglelefteq \mathcal{L}_2$ and $\mathcal{L}_1$ is* not *polynomially predictable, then $\mathcal{L}_2$ is not polynomially predictable.*

Notice that the first case of the theorem gives a means of obtaining learning algorithms for $\mathcal{L}_1$. A restriction is that the hypotheses of these algorithms, while they must be accurate predictors, need not be expressed in a convenient language: in particular, they need not be expressible in $\mathcal{L}_1$. Such learning algorithms are often called *approximation algorithms*, as in the general case an approximation to the target concept may be produced.

The second case of the theorem allows one to transfer hardness results from one language to another; this is useful because for a number of languages, it is known that predicting that language is as hard as breaking cryptographic schemes that are widely assumed to be secure.

A useful special case of this theorem is when the inverse of the concept mapping, $f_c{}^{-1}$, is tractably computable: in this case, if $\mathcal{L}_2$ is pac-learnable, then $\mathcal{L}_1$ will also be pac-learnable. Such concept mappings will be called *reversible concept mappings* in this paper. Another useful case is when f_i preserves distributional properties, such as uniformity: in this case, distribution-specific learnability and predictability results can often be transferred from $\mathcal{L}_1$ to $\mathcal{L}_2$ (or vice versa.)

2.3 Constant-depth determinate clauses

Muggleton [1992] has introduced several useful restrictions on Horn clauses, which we will now describe. If $A \leftarrow B_1 \wedge \ldots \wedge B_r$ is an (ordered) Horn 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*. A literal B_i is *determinate* (with respect to K and X) if for every possible substitution σ that unifies A with some $e \in X$ such that $K \vdash (B_1 \wedge \ldots \wedge B_{i-1})\sigma$ there is at most one substitution θ so that $K \vdash B_i\sigma\theta$. Less formally, a literal is determinate if its output variables have only one possible binding, given K and the binding of the input variables. A clause is determinate if all of its literals are determinate.

Next, define the *depth* of a variable appearing in a clause $A \leftarrow B_1 \wedge \ldots \wedge B_r$ as follows. Variables appearing in the head of a clause have depth zero. Otherwise, let B_i be the first literal containing the variable V, and let d be the maximal depth of the input variables of B_i; then the depth of V is $d+1$. The depth of a clause is the maximal depth of any variable in the clause.

To provide some examples (from Muggleton and Cheng [1992]), the clause

lte(X,Y) $\leftarrow$ successor(X,W) $\wedge$ lte(W,Y).

is determinate whenever *successor* is functional. The maximum depth of a variable is one for C, and hence the clause has depth one. The clause

multiply(X,Y,Z) $\leftarrow$ decrement(Y,W) $\wedge$ multiply(X,W,V) $\wedge$ plus(X,V,Z).

has depth two. It should be emphasized that many programs that are *nondeterministic* when used as Prolog programs contain *determinate* clauses: for example, in the following function-free version of list membership, the first clause is is determinate with depth zero, and the second clause is determinate with depth one. However, the program itself is non-deterministic.

member(X,Ys) $\leftarrow$ head(Ys,X).
member(X,Ys) $\leftarrow$ tail(Ys,Zs) $\wedge$ member(X,Zs).

An interesting class of logic programs is the following.

Definition 4 (*ij*-determinate) *A determinate clause of depth bounded by a constant i over a ground background theory $K \in j$-$\mathcal{K}$ is called ij-*determinate.

The learning program GOLEM, which has been applied to a number of practical problems [Muggleton and Feng, 1992; Muggleton, 1992b], learns ij-determinate programs. Closely related restrictions also have been adopted by several other inductive logic programming systems, including FOIL [Quinlan, 1991], LINUS [Lavrač and Džeroski, 1992], and GRENDEL [Cohen, 1993a]. On the formal side, it has also been shown that for fixed i and j, predicting a single ij-determinate clause can be reduced to predicting a monomial [Džeroski *et al.*, 1992]. The reduction is reversible—i.e., given the learned monomial, an equivalent ij-determinate clause can be found. Let $\mathcal{L}^1_{ij\text{-DET}}$ denote the language of logic programs that contain a single (nonrecursive) ij-determinate clause; an immediate consequence is that $\mathcal{L}^1_{ij\text{-DET}}$ is pac-learnable.

2.4 Notational conventions

In this paper, the domain X for a propositional learning problem will always be a set of variable assignments encoded as binary vectors (as in the example above.) The domain X for a inductive logic programming problem will always be a set of ground atomic facts with the same principle functor and arity; for example, it might be the set of literals of the form *illegal*$(t_1, t_2, t_3, t_4, t_5, t_6)$ where the t_i's range over the set $\{0, \ldots, 7\}$. The complexity of $e \in X$ will be the arity of e. We will consider only background theories K which contain ground unit clauses of arity j or less, for some fixed j; this set of background theories will be written as j-$\mathcal{K}$, or $\mathcal{K}$ when j is an arbitrary constant. When it is more convenient, we will think of $K \in j$-$\mathcal{K}$ as a set of atomic facts (i.e., as a model). Finally, languages $\mathcal{L}$ that are not propositional languages (such as DNF) will always be some subset of the language of function-free Horn clauses; thus we are really considering the learnability of restricted classes of Datalog.

Finally, in all of the Horn clause languages that we consider, the clauses in a program P will all have heads with the same principle functor and arity: we will sometimes refer to such programs as *predicate definitions*, following common parlance among Prolog programmers. The complexity measure on logic programs $P \in \mathcal{L}$ is the sum of the lengths of the clauses in the program; for ground atomic background theories $K \in j$-$\mathcal{K}$, complexity is measured as the number of atoms in K, which we will usually denote n_b.

In this paper, we will denote a background theory by the symbol K, and a language by the symbol $\mathcal{L}$ with an appropriate subscript. If $\mathcal{L}$ is a language of logic programs, the superscript will be used to indicate the number of clauses allowed in a program: $\mathcal{L}^1$ will denote a language of one-clause programs, $\mathcal{L}^k$ will denote a language of k-clause programs, and $\mathcal{L}^*$ will denote a language of programs with any number of clauses. If recursion is allowed it will also be indicated with a superscript: for example, $\mathcal{L}^{k,rec}$ denotes a language of k-clause logic programs that are (possibly) recursive.

For the convenience of the reader, Table 2 on page 22 gives a quick overview of the various languages studied in this paper.

3 Log-depth determinate clauses are not predictable

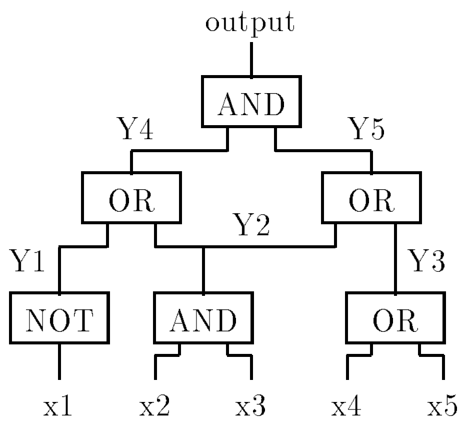
We will first consider generalizing the definition of ij-determinacy by relaxing the restriction that clauses have constant depth. The key result of this section is that any boolean circuit of depth d can be emulated by a determinate clause of depth d; more formally,

Theorem 2 *Let $\mathcal{L}_{d\text{-CIRC}}$ be the language of depth d boolean circuits over n binary variables containing only AND, OR and NOT gates with fan-in two, with the usual semantics and complexity measures.*[5]

There exists a ground background theory $K_{\text{CIR}} \in 3$-$\mathcal{K}$ containing only eleven atomic facts such that

$$\mathcal{L}_{d\text{-CIRC}} \trianglelefteq \mathcal{L}^1_{dj\text{-DET}}[K_{\text{CIR}}]$$

[5]I.e., a circuit C represents the set of all variable assignments such that C outputs a "1", the complexity of an instance is the number of inputs (i.e., the length of the binary vector representing an assignment to those inputs) and the complexity of a circuit is the number of gates in the circuit.



circuit(X1,X2,X3,X4,X5) $\leftarrow$
not(X1,Y1) $\wedge$
and(X2,X3,Y2) $\wedge$
or(X4,X5,Y3) $\wedge$
or(Y1,Y2,Y4) $\wedge$
or(Y2,Y3,Y5) $\wedge$
and(Y4,Y5,Output)$\wedge$
true(Output)

Figure 1: Constructing a determinate clause equivalent to a circuit

A proof will be given shortly; however, the basic idea of the proof is illustrated by example in Figure 1. The importance of this reduction is that the expressive power of depth-bounded boolean circuits, as well as their learnability, has been well studied [Boppana and Sipser, 1990]. This reduction thus has a number of immediate corollaries regarding the learnability of determinate clauses, the most important of which is the following.

Corollary 1 *For $j \geq 3$, the family of languages $\mathcal{L}^1_{(\log n_e)j\text{-DET}}$ is not polynomially predictable, and hence not pac-learnable, under cryptographic assumptions.*[6]

This is a direct corollary of Theorem 2 and Theorem 4 of Kearns and Valiant [1989]. Interestingly, recent work shows that this result holds even if the examples *circuit($X_1, \ldots, X_n$)* are drawn from a uniform distribution [Kharitonov, 1992]: i.e., if every *circuit/n* atom with arguments taken from the set $\{0, 1\}$ has an equal chance of being drawn.[7] This shows that making fairly strong assumptions about the distribution of examples does not make this prediction problem tractable.

We now present a formal proof of the theorem.

Proof: The construction used in the proof is illustrated in Figure 1. An example for the circuit language is a binary vector $b_1 \ldots b_n$; it is converted by the instance mapping f_i to an atom of the form *circuit($b_1, \ldots, b_n$)*. For example the vector 10011 would be converted to *circuit(1,0,0,1,1)*. The background theory K_{CIR} contains definitions of the boolean functions *and*, *or*, and *not* (for example, it would contain the atoms *and(0,0,0)*, *and(0,1,0)*, *and(1,0,0)*, and *and(1,1,1)* as a definition of *and*) and also contains the atom *true(1)*.

We will now define a concept mapping which produces a clause equivalent to a given circuit C; this clause can be constructed as indicated in the figure. For each gate G_i in the

[6]Specifically, this prediction problem is intractable if one or more of the following are intractable: solving the quadratic residue problem, inverting the RSA encryption function, or factoring Blum integers [Kearns and Valiant, 1989]. These problems are widely conjectured to be intractable.

[7]This case requires the additional cryptographic assumption that solving the $n \times n^{1+\epsilon}$ subset sum is hard.

circuit there is a single literal L_i with a single output variable Y_i, defined as

$$L_i \equiv \begin{cases} and(Z_{i1}, Z_{i2}, Y_i) & \text{if } G_i \text{ is an AND gate} \\ or(Z_{i1}, Z_{i2}, Y_i) & \text{if } G_i \text{ is an OR gate} \\ not(Z_i, Y_i) & \text{if } G_i \text{ is an NOT gate} \end{cases}$$

where in each case the Z_{ij}'s are the variables that correspond to the input(s) to G_i. Assume without loss of generality that numbering for the G_i's always puts all the inputs to a gate G_j before G_j in the ordering; then the clause $f_c(C)$ is simply

$$f_c(C) \equiv circuit(X_1, \ldots, X_n) \leftarrow (\bigwedge_{i=1}^{n} L_i) \wedge true(Y_n)$$

Notice that the construction preserves depth. ■

We conjecture that a partial converse also holds: namely, any depth d determinate clause over a fixed background theory K can be emulated by a circuit of depth linear in d.

4 Predictability of indeterminate clauses

4.1 Predictability of k-free clauses

The results of Section 3 indicate that one is not likely to be able to generalize the class of ij-determinate clauses by increasing the depth bound. We will now consider the effect of relaxing the determinacy condition. We will show that the most reasonable relaxation of this condition leads to a language that is as hard to learn as DNF. Further, allowing even a small amount of determinacy makes the learning problem surprisingly hard; informally, for constant l, the problem learning a single l-indeterminate clause is quite similar to the problem of learning l-term DNF.

The most obvious way of relaxing ij-determinacy would be to consider of constant-depth clauses that are not determinate. Unfortunately, these clauses are almost certainly too expressive to learn, as they can encode any NP-complete problem. (For example, if K defines a predicate *node(X)* that is true if X is a node of some graph G and a predicate *edge(X, Y)* that is true if there is an edge from X to Y, then the clause

$$clique \leftarrow (\bigwedge_{i=1}^{n} node(X_i)) \wedge (\bigwedge_{i \neq j} edge(X_i, X_j))$$

succeeds if and only if G contains a clique of size n.) Thus we will consider a slightly more restricted language. Let the *free variables* of a Horn clause be those variables that appear in the body of the clause but not in the head; we will consider the learnability of the language $\mathcal{L}^1_{k\text{-FREE}}$, defined as all nonrecursive programs containing one clause with at most k free variables. Clauses in $\mathcal{L}^1_{k\text{-FREE}}$ are necessarily of depth at most k, but also restricting the number of free variables ensures that clauses can be evaluated in polynomial time.

Notice that $\mathcal{L}^1_{1\text{-FREE}}$ is the *most restricted language possible* that contains indeterminate clauses. We begin with an observation about the expressive power of this language.

Background theory:

for $i = 1, \ldots, k$

$\text{true}_i(b, y)$ for all $b, y : b = 1$ or $y \in 1, \ldots, r$ but $y \neq i$

$\text{false}_i(b, y)$ for all $b, y : b = 0$ or $y \in 1, \ldots, r$ but $y \neq i$

DNF formula: $(v_1 \wedge \overline{v_3} \wedge v_4) \vee (\overline{v_2} \wedge \overline{v_3}) \vee (v_1 \wedge \overline{v_4})$

Equivalent clause:

dnf(X_1,X_2,X_3,X_4) $\leftarrow$

true$_1$(X_1,Y) $\wedge$ false$_1$(X_3,Y) $\wedge$ true$_1$(X_4,Y) $\wedge$

false$_2$(X_2,Y) $\wedge$ false$_2$(X_3,Y)$\wedge$

true$_3$(X_1,Y) $\wedge$ false$_3$(X_4,Y).

Figure 2: *Constructing an indeterminate clause equivalent to a DNF formula*

Theorem 3 *Let $\mathcal{L}_{r\text{-term-DNF}}$ denote the language of r-term DNF formulae. There is a background theory K_r of size polynomial in r such that $\mathcal{L}_{r\text{-term-DNF}} \trianglelefteq \mathcal{L}^1_{\text{1-FREE}}[K_r]$.*

The proof is based on the observation that a clause $p(X) \leftarrow q(X, Y)$ classifies an example $p(a)$ as true exactly when $K \vdash q(a, b_1) \vee \ldots \vee K \vdash q(a, b_r)$, where $b_1, \ldots, b_r$ are the possible bindings of the (indeterminate) variable Y; thus indeterminate variables allow some "disjunctive" concepts to be expressed by a single clause.

Proof: Let K_r contain sufficient atomic facts to define the binary predicates $true_1$, $false_1$, $\ldots$, $true_r$, $false_r$ which behave as follows:

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

We now define the instance mapping f_i to map an assignment $b_1 \ldots b_n$ to the atom $dnf(b_1, \ldots, b_n)$. The concept mapping f_c is defined to map a formula of the form

$$\phi \equiv \bigvee_{i=1}^{r} \bigwedge_{j=1}^{r_i} l_{ij}$$

to the Horn clause

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

where Lit_{ij} is defined as

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

Notice that there is only one variable Y not appearing in the head, and it can be bound to only the r values $1, \ldots, r$. (I.e., any resolution proof of $dnf(b_1, \ldots, b_n)$ must use a substitution θ that replaces Y with 1 or $\ldots$or r.) It is easy to see that if ϕ is true for an assignment

$b_1 \ldots b_n$, then some term $T_i = \bigwedge_{j=1}^{r_i} l_{ij}$ must be true; in this case $\bigwedge_{j=1}^{r_i} Lit_{ij}$ succeeds (with Y bound to the value i) and $\bigwedge_{j=1}^{r'_i} Lit_{i'j}$ for every $i' \neq i$ also succeeds with Y bound to i. On the other hand, if ϕ is false for an assignment, then each T_i fails, and hence for every possible binding of Y some conjunction $\bigwedge_{j=1}^{r'_i} Lit_{ij}$ will fail. Thus concept membership is preserved by the mapping. ■

Since a DNF term of complexity n can have at most n terms, we the following as a corollary:

Corollary 2 *The family of languages $\mathcal{L}^1_{k\text{-FREE}}[\mathcal{K}]$ is predictable only if $\mathcal{L}_{\text{DNF}}$ is predictable.*

It also should be noted that every clause in $\mathcal{L}^1_{1\text{-FREE}}[K_r]$ can be easily translated into an r-term DNF expression; thus together with existing hardness results for minimizing DNF expressions (e.g. [Kearns *et al.*, 1987]) the reduction above also leads to a number of representation *dependent* hardness results for pac-learning clauses in $\mathcal{L}^1_{k\text{-FREE}}$, somewhat along the lines of Theorem 1 of Kietz [Kietz, 1993]. It is easy to show, for example, that for the background theory K_r shown above the language $\mathcal{L}^1_{1\text{-FREE}}[K_r]$ is not pac-learnable for $r > 2$.

An important question is whether there are languages in $\mathcal{L}^1_{k\text{-FREE}}$ that are *harder* to learn than DNF. The answer to this questions is no:

Theorem 4 *For every k and every background theory $K \in a\text{-}\mathcal{K}$, $\mathcal{L}^1_{k\text{-FREE}}[K] \trianglelefteq \mathcal{L}_{\text{DNF}}$. Thus the family of languages $\mathcal{L}^1_{k\text{-FREE}}$ is predictable if and only if DNF is predictable.*

Proof: Let $Cl = A \leftarrow B_{c_1} \wedge \ldots \wedge B_{c_l}$ be a clause in $\mathcal{L}^1_{k\text{-FREE}}[K]$. As we assume clauses are nonrecursive, Cl covers an example e iff

$$\exists \sigma : K \vdash (B_{c_1} \wedge \ldots \wedge B_{c_l})\sigma\theta_e \qquad (1)$$

where θ_e is the most general unifier of A and e. However, since the background theory K is of size n_b, and all predicates are of arity a or less, there are at most an_b constants in K, and hence only $(an_b)^k$ possible substitutions $\sigma_1, \ldots, \sigma_{(an_b)^k}$ to the k free variables. Also (as we assume clauses are function-free) if K defines l different predicates, there are at most $l \cdot (n_e + k)^a < n_b \cdot (n_e + k)^a$ possible literals $B_1, \ldots, B_{n_b \cdot (n_e+k)^a}$ that can appear in the body of a $\mathcal{L}^1_{k\text{-FREE}}$ clause.

So, let us introduce the boolean variables v_{ij} where i ranges from one to $n_b \cdot (n_e + k)^a$ and j ranges from one to $(an_b)^k$. We will define f_i of an example e to return an assignment η_e to these variables: v_{ij} will be true in η_e if and only if $K \vdash B_i\sigma_j\theta_e$. This means that Equation 1 is true exactly when the DNF formula

$$\bigvee_{j=1}^{(an_b)^a} \bigwedge_{i=1}^{l} v_{c_i j}$$

is true; thus the clause Cl can be emulated by DNF over the v_{ij}'s. ■

This does not actually settle the question of whether indeterminate clauses are predictable, but does show that answering the question will require substantial advances in computational learning theory, as the predictability of DNF has been an open problem in computational learning theory for several years.

4.2 Clauses with bounded indeterminacy

We will now consider the effects of allowing a constant amount of indeterminacy. We will want to talk about clauses that are almost, but not quite, deterministic; thus the following definition.

Definition 5 (l-indeterminate) *A clause $A \leftarrow B_1 \wedge \ldots \wedge B_r$ is called* l-indeterminate *(with respect to K and X) iff for every possible substitution σ that unifies the head of A to some ground instance $e \in X$ and for all $i = 1, \ldots, r$ there are at most l distinct substitutions θ such that $K \vdash (B_1 \wedge \ldots \wedge B_i)\theta\sigma$.*

Informally, a clause is l-indeterminate if there are at most l substitutions that can be used to prove the clause, or any prefix of it. Determinate clauses are 1-indeterminate.

Let $\mathcal{L}_{k\text{-term-DNF}}$ be the language of k-term DNF formulae. It is trivial to extend Theorem 3 to show the following.

Corollary 3 *For any constant l, $\mathcal{L}_{l\text{-term-DNF}} \trianglelefteq (\mathcal{L}^1_{1\text{-FREE}} \cap \mathcal{L}^1_{l\text{-INDET}})$.*

We will next define a particular class of languages; this definition is, intended as a generalization of the core property that, together with determinism, makes ij-determinate clauses easy to pac-learn.

Definition 6 (Polynomial literal support) *A language $\mathcal{L}^1[K]$ is a* Horn clause language *with* polynomial literal support (pls) *if every $C \in \mathcal{L}^1$ is a logic program containing a single Horn clause, and if for every X and $K \in \mathcal{K}$ there is a set of literals LIT and a partial order $\preceq$ such that*

- *the cardinality of LIT is polynomial in n_e and n_b;*
- *$\mathcal{L}^1[\mathcal{K}]$ is exactly those clauses $A \leftarrow B_1 \wedge \ldots \wedge B_r$, where A is fixed, all the B_i's are members of LIT, and each B_i either immediately precedes or is equivalent to B_{i+1} in the partial order $\preceq$.*

We have the following result, which generalizes the result that ij-determinate languages are pac-learnable.

Theorem 5 *Let $\mathcal{L}_{\text{MONOMIAL}}$ be the language of monomials, with the usual complexity measures, and let $\mathcal{L}^1_{\text{DET-PLS}}$ be any language of determinate clauses with polynomial literal support. Then for all $K \in j\text{-}\mathcal{K}$,*

$$\mathcal{L}^1_{\text{DET-PLS}}[K] \trianglelefteq \mathcal{L}_{\text{MONOMIAL}}$$

and hence is polynomially predictable.

Proof: The proof is a slight generalization of the proof of Theorem 1 in Džeroski, Muggleton and Russell [1992]. Since $\mathcal{L}^1_{\text{DET-PLS}}$ is determinate, the following process will yield a unique substitution θ given an example e and the background knowledge K.

- Let $LIT = \{L_1, \ldots, L_n\}$. Start with an empty substitution θ, and consider each of the L_i's in an order consistent with $\preceq$:
- For each L_i:
  - if L_i contains output variables and some binding for those variables exists that makes L_i true (relative to K) then add that (unique) binding to θ;
  - otherwise, bind the output variables of L_i to some constant not appearing in K (i.e., a null value).

Now let us construct a set of n boolean variables $v_{L_1}, \ldots, v_{L_n}$ and define an instance mapping f_i that maps an example e to an assignment to the v_{L_i}'s that makes v_{L_i} true iff $L_i\theta \in K$, for the θ constructed by the process described above. Finally define a concept mapping f_c that maps the clause $A \leftarrow L_{i1} \wedge \ldots \wedge L_{ir}$ to the monomial $v_{L_{i1}} \ldots v_{L_{1r}}$. It is easy to show that these mappings preserve concept membership (the proof follows the proof of Theorem 1 of Džeroski, Muggleton and Russell [1992].) Since monomials are pac-learnable, it follows that $\mathcal{L}^1_{\text{DET-PLS}}$ is predictable. ■

We now consider extending Theorem 5 from determinate languages to l-indeterminate languages with pls. One example of a language that is l-indeterminate and has pls is the language of constant-depth clauses containing a polynomial number of determinate literals and indeterminate literals from a fixed, constant-size set; if the indeterminacy of these indeterminate literals was also bounded, it is not hard to show that the indeterminancy of such a clause would be bounded by a constant.

The principle result is the following.

Theorem 6 *Let $\mathcal{L}^1_{k\text{-INDET-PLS}}$ be any language of l-indeterminate clauses with polynomial literal support. Then for any fixed j and any $K \in j$-$\mathcal{K}$,*

$$\mathcal{L}^1_{k\text{-INDET-PLS}}[K] \trianglelefteq \mathcal{L}_{l\text{-term-DNF}}$$

Proof: We extend the previous proof by supposing that an alternative process is used to generate a set of substitutions given an example e and the background knowledge K.

- Start with an empty substitution θ, and consider each of the L_i's in order.
- If L_i contains output variables and bindings for those variables exist that make L_i true (relative to K) then add some such binding to θ.

  Otherwise, bind the output variables of L_i to some constant not appearing in K.

Since the clauses of $\mathcal{L}^1_{k\text{-INDET-PLS}}$ are l-indeterminate, this nondeterministic process can produce at most l different substitutions for any instance e; thus backtracking or some similar mechanism can be used to generate in polynomial time all l substitutions, which we will denote $\theta_1, \ldots, \theta_l$. The ordering of these substitutions can be arbitrary (we will see why shortly.)

Now let us define a set of ln boolean variables

$$v_{1L_1}, \ldots, v_{lL_1}, \ldots, v_{1L_n}, \ldots, v_{lL_n}$$

and define an instance mapping f_i that maps an example e to an assignment to the v_{jL_i}'s that makes v_{jL_i} true iff $L_i\theta_j \in K$ for θ_j, the j-th substitution constructed by the process above. Finally define a concept mapping f_c that maps the clause $C \equiv (A \leftarrow L_{i_1}, \ldots, L_{i_r})$ to the l-term DNF formula

$$f_c(C) \equiv \bigvee_{j=1}^{l} (v_{jL_{i_1}} \wedge \ldots \wedge v_{jL_{i_r}})$$

Note that when a clause C covers an example e, then it must be that some θ_j makes the clause true, and hence one of the terms of $f_c(C)$ will be true; conversely, when C doesn't cover e, no terms of $f_c(C)$ are true. So these mappings preserve concept membership.

Notice that the ordering of the θ_i's is irrelevant, and can even be different for different examples. ■

Notice that this is a positive result, as l-term DNF is predictable, for constant l. However, all known algorithms for predicting k-term DNF require time exponential in k. (The standard algorithm is to approximate k-term DNF with k-CNF; this requires time $O(n^k)$ where n is the number of boolean features.)

4.3 Discussion

To summarize, we have shown that strong parallels exist between predicting DNF and predicting indeterminate clauses. In particular, the language of k-free clauses is predictable if and only if DNF is predictable. This does not actually settle the question of whether such clauses are predictable, but does show that answering the question will require substantial advances in computational learning theory; the learnability and predictability of DNF has been an open problem in computational learning theory for several years.

Furthermore, l-indeterminate clauses can express l-term DNF concepts, and conversely, predicting any l-indeterminate clause with pls can be reduced to predicting k-term DNF. Thus predicting these clauses with bounded indeterminacy is possible; however, known prediction algorithms are exponential in k. Pac-learning this class is harder; notice that since the concept mapping used in the proof is not reversible, the proof of predictability can not be adapted to produce a pac-learning algorithm even for those classes of distributions (e.g. bounded distributions) for which k-term DNF is pac-learnable.

5 Pac-learnable indeterminate languages

We will now consider some alternative restrictions on indeterminate clauses; our aim is to find reasonably expressive languages which are not only predictable but also pac-learnable. The first restriction we consider, called *locality*, is suggested by the observation that the construction in Theorem 3 requires a free variable that appears in every literal.

Definition 7 (Locality) *Let V_1 and V_2 be two variables appearing in a clause $A \leftarrow B_1 \wedge \ldots \wedge B_r$. We say that V_1* touches *V_2 if they appear in the same literal, and that V_1* influences *V_2 if it either touches V_2, or if it touches some variables V' that influences V_2. (Thus* influences *and* touches *are both symmetric and reflexive relations, and* influences *is the transitive closure of* touches*.) The* locale *of a variable V is the set of literals $\{B_{i_1}, \ldots, B_{i_l}\}$ that contain variables influenced by V. The* locality *of a variable is the cardinality of its locale. Finally, the* locality *of a clause is the maximum locality of any free variable in that clause, where a* free variable *is one that appears in the body but not the head of the clause.*

For example, consider the following clause.

high_risk_driver(Dr) ←
single(Dr) ∧ male(Dr) ∧
years_old(Dr,Age) ∧ less_than_25(Age) ∧
offense(Dr,Off) ∧ moving_violation(Off) ∧
date(Off,Yr1) ∧ current_year(Yr2) ∧ recent(Yr1,Yr2).

The variable *Dr* touches the variables *Age* and *Off*, and influences every variable in the clause. The variable *Age* influences no other variables, has the locale {*years_old(Dr,Age)*, *less_than_25(Age)*}, and hence has locality two. The variable *Yr1* touches *Yr2* and also has locality two. The variable *Off* touches *Yr1* and, through it, influences *Yr2*, has the locale

{*offense(Dr,Off)*,*moving_violation(Off)*,*date(Off,Yr1)*,*current_year(Yr2)*,*recent(Yr1,Yr2)*}

and hence has locality five. The locality of the clause is thus five, the maximum locality of the output variables *Age*, *Off*, *Yr1*, and *Yr2*.

Finally, let $\mathcal{L}^1_{l\text{-LOCAL}}$ represent the language of non-recursive logic programs containing a single clause with locality l or less. We have the following result:

Theorem 7 *For any fixed l and j, the family of languages $\mathcal{L}^1_{l\text{-LOCAL}}[j\text{-}\mathcal{K}]$ is pac-learnable from positive examples alone.*

Proof: As there are only n_e variables in the head of the clause, and every new literal in the body can introduce at most j new variables, any length l locale can contain at most $n_e + jl$ distinct variables. Since there are at most n_b distinct predicates represented in the background theory K, a simple counting argument thus establishes a polynomial upper bound of

$$p = (n_b(n_e + jl)^j)^l \tag{2}$$

on the number of different[8] locales. Let us denote these as $LOC_1, \ldots, LOC_p$. Now, notice that every clause of locality l can be written in the form $A \leftarrow LOC_{i_1} \wedge \ldots \wedge LOC_{i_r}$ where each LOC_{i_j} is one of the possible locales, and all of these locales LOC_{i_j} are "disjoint"—i.e., they share no free variables. One can thus reduce pac-learning such a clause to pac-learning a conjunction over a set of boolean variables $v_1, \ldots, v_p$, where each variable represents a locale. (More precisely, an instance $e = A\theta$ is mapped to assignment where each v_i is

[8]Up to renaming of variables.

true iff $K \vdash LOC_i\theta$, and the clause $A \leftarrow LOC_{i_1} \wedge \ldots \wedge LOC_{i_r}$ is mapped to the conjunction $v_{i_1} \ldots v_{i_r}$. It is easy to show that these mappings preserve concept membership, and that the concept mapping is reversible.) The theorem thus follows from the fact that monomials are pac-learnable from positive examples only. ■

This restriction does not correspond very well to the sorts of clauses typically used in logic programs for list manipulation and other programming tasks. However, it may be useful when logic programs are being used to represent other sorts of knowledge [Muggleton, 1992b; Quinlan, 1990]. An advantage of locality over l-indeterminacy is that when constructing a background theory, the user does not need to think about the degree to which individual relations are indeterminate. We also note that it is straightforward to extend this result somewhat; for example, it is easy to pac-learn the language of clauses of the form

$$A \leftarrow B_1 \wedge \ldots \wedge B_r \wedge D_1 \wedge \ldots \wedge D_s$$

where $A \leftarrow B_1 \wedge \ldots \wedge B_r$ is ij-determinate and $A \leftarrow D_1 \wedge \ldots \wedge D_s$ is l-local.

While restricting locality makes a single clause pac-learnable, there is still no provably tractable method of pac-learning a program containing a number of local clauses. By considering a much stronger restriction we obtain the following result.

Theorem 8 *Let $\mathcal{L}^*_{l\text{-LENGTH}}$ represent the language of non-recursive logic programs containing any number of clauses, each of which contains l or fewer literals in its body. For any fixed l and j, the family of languages $\mathcal{L}^*_{l\text{-LENGTH}}[j\text{-}\mathcal{K}]$ is pac-learnable from negative examples only.*

Proof: Using the counting argument used to compute Equation 2, for a fixed head A, there are only a polynomial number $p = (n_b(n_e + jl)^j)^l$ of clauses of length l or less. Since the language is nonrecursive, every concept in $\mathcal{L}^*_{l\text{-LENGTH}}[K]$ is simply a disjunction of these clauses, and can be pac-learned by the same method used to pac-learn k-DNF: return as a hypothesis a program C containing all clauses in $\mathcal{L}^*_{l\text{-LENGTH}}[K]$ that do not cover any negative examples. ■

$\mathcal{L}^*_{l\text{-LENGTH}}$ is a very restricted language, arguably too restricted to be useful; however, it is the most expressive one (we know of) for which programs with any number of clauses are provably pac-learnable. The language is also useful for studying the effect of recursion on learnability; we will see in the next section that adding recursion makes even $\mathcal{L}^*_{l\text{-LENGTH}}$ hard to predict.

6 Most recursive programs are not predictable

In this section, we will consider the effects of adding recursion to languages that are either known to be predictable (like $\mathcal{L}^*_{l\text{-LENGTH}}$ and $\mathcal{L}^1_{l\text{-LOCAL}}$) or which might be predictable (like the language of ij-determinate programs, which for nonrecursive programs is equivalent to DNF.) First, we will show that adding recursion to either $\mathcal{L}^*_{l\text{-LENGTH}}$ or to the language of ij-determinate programs makes them hard to predict. Next, we will adapt this construction to show that adding recursion show that even the language of one-clause ij-determinate programs[9] with high locality is hard to predict. All of these results hold only for predicting against an arbitrary distribution.

6.1 Preliminaries

Investigation of the learnability of recursive programs requires a slight change in our formalism. It is often useful to assume (in writing learning algorithms) that the literals in the body of a clause are of small arity. For nonrecursive predicates, requiring that the atoms in the background theory be of bounded arity j ensures that this is true; however, if recursion is used, and the examples are atoms of arity n_e, there will be literals of arity n_e in the body of a clause. To avoid this problem we will (in this section of the paper only) represent an example as a ground *goal atom*, e, plus a set of up to n_e ground *description atoms* $D = \{d_1, \ldots, d_{n_e}\}$ of arity bounded by j; a concept C now classifies an example (e, D) as true iff $C \wedge K \wedge D \vdash e$.

This also allows structured objects like lists to be used in examples. For example, if we are trying to learn the predicate *member(X, Y)* used as an example in Section 2.3, then the example *member(b,[a,b,c])* might be represented as the goal atom $e =$*member(b,list_abc)* together with the description

$D = \{$ head(list_abc,a), tail(list_abc,list_bc),
head(list_bc,b), tail(list_bc,list_c),
head(list_c,c), tail(list_c,nil) $\}$

This formalization follows the actual use of learning systems like FOIL.

We will also extend our notion of a "family of languages" slightly, and let $\mathcal{L}_{\mathrm{DLOG}}[n]$ represent the language of log-space bounded deterministic Turing machines accepting inputs of size n or less, with the usual semantics and complexity measure.[10]

6.2 Recursion and short local determinate clauses

The first result of this section is the following, which shows that recursive programs with a polynomial number of clauses are hard to predict against an arbitrary distribution, even for relatively simple languages of clauses.

Theorem 9 *Let $\mathcal{L}^{*,rec}_{ij\text{-DET}}$ denote the language of recursive logic programs containing any number of ij-determinate clauses, let $\mathcal{L}^{*,rec}_{l\text{-LENGTH}}$ represent the language of recursive logic programs containing any number of length l clauses, and let $\mathcal{L}^{*,rec}_{l\text{-LOCAL}}$ represent the language of recursive logic programs containing any number of l-local clauses.*

For every n, there exists a ground background theory K_n of size polynomial in n such that

$$\mathcal{L}_{\mathrm{DLOG}}[n] \trianglelefteq (\mathcal{L}^{*,rec}_{12\text{-DET}}[K_n] \cap \mathcal{L}^{*,rec}_{4\text{-LENGTH}}[K_n] \cap \mathcal{L}^{*,rec}_{4\text{-LOCAL}}[K_n])$$

[9]For languages of one-clause recursive programs, we will assume that the base case for the recursion is known—i.e., that it is part of the background theory K.

[10]I.e. a machine represents the set of all inputs that it accepts, and its complexity is the number of internal states.

Proof: We will show that there is a single fixed background theory that allows any log-space bounded Turing machine to be emulated by a logic program that is 12-determinate and contains only clauses of length 4.

Recall that a log-space bounded Turing machine has an input tape of length n, a work tape of length $\log_2 n$ which initially contains all zeros, and a finite state control with state set Q. To simplify the proof, we assume without loss of generality that the tape and input alphabets are binary, that there is a single accepting state $q_f \in Q$, and that the machine will always erase its work tape and position the work tape head at the far left after it decides to accept its input. At each time step, the machine will read the tape squares under its input tape head and work tape head, and based on these values and its current state q, it will

- write either a 1 or a 0 on the work tape,
- shift the input tape head left or right,
- shift the work tape head left or right, and
- transition to a new internal state q'

A deterministic machine can thus be specified by a transition function

$$\delta : \{0,1\} \times \{0,1\} \times Q \longrightarrow \{0,1\} \times \{L,R\} \times \{L,R\} \times Q$$

Let us define the *internal configuration* of a TM to consist of the string of symbols written on the worktape, the position of the tape heads, and the internal state q of the machine: thus a configuration is an element of the set

$$CON \equiv \{0,1\}^{\log_2 n} \times \{1,\ldots,\log_2 n\} \times \{1,\ldots,n\} \times Q$$

A simplified specification for the machine is the transition function

$$\delta' : \{0,1\} \times CON \rightarrow CON$$

where the component $\{0,1\}$ represents the contents of the input tape at the square below the input tape head.

Notice that for a machine whose worktape size is bounded by $\log n$, the cardinality of CON is only $p = |Q| n^2 \log_2 n$, a polynomial in n. We will use this fact in our constructions of the background theory, the instance mapping, and the concept mapping.

The background theory K_n consists the following. First, for $i = 0,\ldots,p$, an atom of the form $con_i(c_i)$ is present; each constant c_i represents a different internal configuration of the Turing machine. We will arbitrarily select c_1 to represent the (unique) accepting configuration, and thus add to K_n the atom $accepting(c_1)$. Thus

$$K_n \equiv \{con_i(c_i)\}_{i=1}^{p} \cup \{accepting(c_1)\}$$

Now, we define the mapping functions. An instance in the Turing machine's domain is a binary string $X = b_1 \ldots b_n$; this is mapped by f_i to the goal atom $tm(new_X, c_0)$, where new_X is some new constant not appearing in K_n, and a set of n description atoms defining the predicate $true_i(X)$ to be true iff the i-th bit of X is a "1", and the predicate $false_i(X)$ to be true iff the i-th bit of X is "0". In other words,

$$\begin{aligned} e &\equiv tm(new_X, c_0) \\ D &\equiv \{true_i(new_X)\}_{b_i \in X : b_i = 1} \;\cup\; \{false_i(new_X)\}_{b_i \in X : b_i = 0} \end{aligned}$$

The constant c_0 will represent the start configuration of the Turing machine, and the predicate $tm(X,C)$ will be defined so that it is true iff the Turing machine accepts input X starting from state C.

For the concept mapping f_c, let us assume some arbitrary one-to-one mapping η between the internal configurations of a Turing machine M and the predicate names $con_0, \ldots, con_{p-1}$ such that the start configuration $(0^{\log_2 n}, 1, q_0)$ maps to con_0 and the accepting configuration $(0^{\log_2 n}, 1, q_f)$ maps to con_1. We will construct the program $f_c(M)$ as follows. For each transition $(1, c) \rightarrow c'$ in δ', where c and c' are in CON, construct a clause of the form

$$\mathrm{tm(X,C)} \leftarrow \mathrm{con}_j\mathrm{(C)} \wedge \mathrm{true}_i\mathrm{(X)} \wedge \mathrm{con}_{j'}\mathrm{(C1)} \wedge \mathrm{tm(X,C1)}.$$

where i is the position of the input tape head which is encoded in c, $con_j = \eta(c)$, and $con_{j'} = \eta(c')$. For each transition $(0, c) \rightarrow (c')$ in δ' construct an analogous clause, in which $true_i(X)$ is replaced with $false_i(X)$. Finally, construct a clause

$$\mathrm{tm(X,C)} \leftarrow \mathrm{accepting(C)}.$$

It is easy to show that these clauses are 12-determinate, of length (and hence locality) less than four, and that this construction preserves concept membership. ■

The class of all concepts that can be tested in deterministic logspace is known to be hard to predict against an arbitrary distribution [Pitt and Warmuth, 1990; Kearns and Valiant, 1989]. Thus we have the following as an immediate corollary:

Corollary 4 *The families of languages* $\mathcal{L}^{*,rec}_{ij\text{-DET}}[\mathcal{K}]$ *for* $i \geq 1$ *and* $j \geq 2$, $\mathcal{L}^{*,rec}_{l\text{-LENGTH}}[\mathcal{K}]$ *for* $l \geq 4$, *and* $\mathcal{L}^{*,rec}_{l\text{-LOCAL}}$ *for* $l \geq 4$ *are not polynomially predictable, and hence are not pac-learnable, under cryptographic assumptions.*[11]

6.3 Recursion and one local determinate clause

Next, we will extend this construction to show that recursion also makes prediction impossible if the number of clauses, rather than the length of each clause, is bounded. This is true even if locality remains bounded.

Let $\mathcal{L}^{1,rec}_{l\text{-LOCAL}}$ be the language of recursive logic programs containing a single clause with locality l or less,[12] and let $\mathcal{L}^{1,rec}_{ij\text{-DET}}$ denote the language of recursive logic programs containing a single ij-determinate clauses. We have the following result.

[11]This prediction problem is intractable if one or more of the following are intractable: solving the quadratic residue problem, inverting the RSA encryption function, or factoring Blum integers [Kearns and Valiant, 1989].

[12]More accurately, we will assume that the background theory already contains a clause defining the base case for the target recursive program; thus only one recursive clause needs to be learned.

Theorem 10 *For every n, there exists a ground background theory $K'_n \in$ 3-$\mathcal{K}$ of size polynomial in n such that*

$$\mathcal{L}_{\text{DLOG}}[n] \trianglelefteq (\mathcal{L}^{1,rec}_{\text{4-LOCAL}}[K'_n] \cap \mathcal{L}^{1,rec}_{\text{33-DET}}[K'_n])$$

Thus the families of languages $\mathcal{L}^{k,rec}_{l\text{-LOCAL}}$ for $l \geq 4$ and $k \geq 1$ and $\mathcal{L}^{k,rec}_{ij\text{-DET}}$ for $i \geq 3$, $j \geq 3$, and $k \geq 1$ are not polynomially predictable, under cryptographic assumptions.

Proof: The proof is an extension of the proof of Theorem 9 that makes use of a slightly different background theory K'_n, and slightly different concept and instance mappings.

Let us first make some simplifying assumptions about the Turing machine M. Since M is deterministic, for every pair (b, c), where $b \in \{0, 1\}$ and $c \in CON$ there is either exactly one or zero transitions in δ' for which (b, c) is the right-hand side. If there are no transitions from (b, c) then M will always fail once it reaches this configuration. One may assume without loss of generality that there is exactly one such "failing" configuration, and represent it by c_{p-1}. Thus all configurations (b, c) will have a single transition, unless $c = c_1$ (the accepting configuration) or $c = c_{p-1}$ (the failing configuration).

We will first define the instance mapping f_i. Instead of defining an *accepting* predicate in the background theory, we will insert the base case of the *tm/2* predicate directly into the description atoms for an instance: thus every string $X = b_1 \ldots b_n$ will be mapped to (e, D'), where D' contains the atom *tm(new$_X$,c$_1$)* (with c_1 representing the accepting configuration). Also, instead of the predicates *true$_i$* and *false$_i$*, D' will contain atoms that define the predicate *bit$_i$(X,B)*, which is true whenever B is the i-th bit of the input string X. Thus D will be the $n + 1$ atoms

$$\begin{aligned} e &\equiv tm(new_X, c_0) \\ D &\equiv \{bit_i(new_X, B)\}_{B \text{ is } i\text{-th bit of } X} \cup \{tm(new_X, c_1)\} \end{aligned}$$

Now we will define the background theory K'_n. For $b \in \{0, 1\}$ and $i = 1, \ldots, p - 2$, this theory contains the p atoms needed to define the predicate *con$_{b,i}$(C,B,Y)*, which is true if $B = b$, $C = c_i$ and $Y =$ *active*, or if $B \neq b$ or $C \in CON - \{c_i, c_{p-1}\}$ and $Y =$ *inactive*. Informally, given values C and B, *con$_{b,i}$(C,B,Y)* will fail if C is the failing configuration c_{p-1}; otherwise, it will succeed, binding Y to *active* if $C = c_i$ and $B = b$ and binding Y to *inactive* otherwise. Also, for $i = 1, \ldots, p$, the theory contains two atoms defining the predicate *con$_i$(Y,C)* to be true if $Y =$ *active* and $C = c_i$, or if $Y =$ *inactive* and $C = c_1$ (the accepting configuration.) Notice that the complexity of K'_n is polynomial in p, and hence n.

Now, let η be any one-to-one mapping between the internal configurations of a Turing machine M and the predicate names *con*$_0$,…,*con*$_{p-1}$ that maps the start configuration to *con*$_0$, the accepting configuration to *con*$_1$, and the failing configuration to *con*$_{p-1}$. The concept mapping will map the Turing machine M to a program containing the single clause

$$\text{tm(Input,C)} \leftarrow \bigwedge_{((b,c) \rightarrow c') \in \delta'} (\text{bit}_i(\text{Input,B}_{bc}) \wedge \text{con}_{b,j}(\text{C,B}_{bc},\text{Y}_{bc}) \wedge \text{con}_{j'}(\text{Y}_{bc},\text{C1}_{bc}) \wedge \text{tm(Input,C1}_{bc}))$$

where in each conjunction

$$\text{AND}_{bc} \equiv \text{bit}_i(\text{Input},\text{B}_{bc}) \wedge \text{con}_{b,j}(\text{C},\text{B}_{bc},\text{Y}_{bc}) \wedge \text{con}_{j'}(\text{Y}_{bc},\text{C1}_{bc}) \wedge \text{tm}(\text{Input},\text{C1}_{bc})$$

the predicates have the following meaning:

- i is the position of the input tape head encoded in c,
- $con_j = \eta(c)$ and $con_{j'} = \eta(c')$, and
- the variables Y_{bc}, B_{bc} and $C1_{bc}$ are used only locally in each such conjunction (i.e., for conjunctions corresponding to different transitions from δ, different variables Y_{bc}, B_{bc}, and $C1_{bc}$ are used.)

It is easy to see that this program is 4-local and 33-determinate. We claim that it also properly emulates the Turing machine M; i.e., that *tm(X,C)* is true iff M accepts input X starting on configuration C. This is easiest to see by induction on the number of steps s required by M to accept. We omit the details of the proof, but will sketch the reasoning behind the inductive step. Notice that for some fixed $C = c_j$ and some fixed input X, most of the conjuncts in the antecedent of the clause defining *tm/2* can be easily proved: for each conjunction $AND_{b'c'}$ where (b', c') is the not the right-hand side of the transition that M would actually make, $AND_{b'c'}$ is true with $Y_{b'c'}$ = *inactive* and $C1_{b'c'} = c_1$. However, the conjunction AND_{bc} for that (b, c) corresponding to the Turing machine's actual configuration is provable iff *tm(X,*$c_{j'}$*)* is provable; by the inductive hypothesis, this is true iff M accepts X starting in configuration $c_{j'}$ which is precisely when M accepts X starting in the configuration c_j. ■

6.4 Linear recursion and one recursive clause

So far, the results of this section have been discouraging: they show that if recursion is allowed, then some extremely simple languages of logic programs become hard to predict. In particular, even when the length of every clause is bounded by a constant, or when a program must consist of only a single clause that both deterministic and local, prediction is cryptographically hard.

There are some additional restrictions which might improve learnability. One possibility is that learning is possible in a distribution-specific setting; notice that the instance mapping used in Theorems 9 and 10 lead to a distribution in which most of the instance space has a zero weight (in particular, every instance *tm(X,C)* where $C \neq c_0$ has zero probability of being chosen.) Another natural restriction is to consider only *linear recursion:* a predicate definition is linearly recursive if the body of every clause contains only a single literal that has the same principle functor and arity as the head of a clause. The construction of Theorem 9 requires only linear recursion, so a corollary is that programs with an unbounded number of deterministic short clauses are not predictable; however, the results leave open the possibility that linearly recursive programs with a bounded number of clauses are learnable.

Define a recursive program be be *closed* if every recursive literal has no output variables; in another paper [Cohen, 1993b], we present a positive result for the following (extremely restricted) class of programs.

$\mathcal{L}_{\text{MONOMIAL}}$	$\trianglelefteq$	$(\mathcal{L}^1_{ij\text{-DET}})$		$\boxed{\mathcal{L}^1_{l\text{-LOCAL}}}$		$\trianglelefteq$	$\mathcal{L}_{\text{MONOMIAL}}$
$\mathcal{L}_{k\text{-DNF}}$	$\trianglelefteq$				$\boxed{\mathcal{L}^*_{l\text{-LENGTH}}}$	$\trianglelefteq$	$\mathcal{L}_{k\text{-DNF}}$
$\mathcal{L}_{k\text{-term-DNF}}$	$\trianglelefteq$	$(\mathcal{L}^k_{ij\text{-DET}})$	$\boxed{\mathcal{L}^1_{k\text{-INDET-PLS}}}$	$\mathcal{L}^k_{l\text{-LOCAL}}$		$\trianglelefteq$	$\mathcal{L}_{k\text{-term-DNF}}$
			$\mathcal{L}^1_{*\text{-INDET-PLS}}$				
$\mathcal{L}_{\text{DNF}}$	$\trianglelefteq$	$(\mathcal{L}^*_{ij\text{-DET}})$	$\mathcal{L}^*_{k\text{-INDET-PLS}}$	$\mathcal{L}^*_{l\text{-LOCAL}}$		$\trianglelefteq$	$\mathcal{L}_{\text{DNF}}$
			$\mathcal{L}^*_{*\text{-INDET-PLS}}$				
			$\boxed{\mathcal{L}^1_{k\text{-FREE}}}$				
		$\boxed{\mathcal{L}^1_{(\log n)3\text{-DET}}}$					
$\mathcal{L}_{\log n\text{-CIRC}}$	$\trianglelefteq$	$\mathcal{L}^k_{(\log n)3\text{-DET}}$					
		$\mathcal{L}^*_{(\log n)3\text{-DET}}$					
		$\boxed{\mathcal{L}^{*,linrec}_{12\text{-DET}}}$	$\mathcal{L}^{*,linrec}_{1\text{-INDET-PLS}}$		$\boxed{\mathcal{L}^{*,linrec}_{4\text{-LENGTH}}}$		
$\mathcal{L}_{\text{DLOG}}[n]$	$\trianglelefteq$						
		$\boxed{\mathcal{L}^{1,rec}_{33\text{-DET}}}$		$\boxed{\mathcal{L}^{1,rec}_{4\text{-LOCAL}}}$			

Table 1: Overview of known reducibilities

Theorem 11 (From [Cohen, 1993b]) *The language of one-clause closed linear recursive ij-determinate logic programs is pac-learnable.*

This result can be extended somewhat if one assumes that some additional information is available. For example, two-clause closed linear recursive ij-determinate programs are learnable if one assumes the existence of an oracle determining when an instance is an example of the recursive clause, and when it is an example of the base clause. This language includes many common ILP benchmarks.

7 Summary and related work

7.1 Summary

Table 1 gives a quick overview our results, together with some immediate corollaries and observations. The boxed results are the "key" results of the paper—those requiring a non-trivial proof—and unboxed results are straightforward corollaries. For completeness, we also include in the table previous learnability results; these are enclosed in parentheses.

In the table, we place $\mathcal{L}_{\mathrm{FOO}} \trianglelefteq$ on the far left of a row if for every family of languages $\mathcal{L}_{\mathrm{BAR}}$ in the row, there is some background theory K such that $\mathcal{L}_{\mathrm{FOO}} \trianglelefteq \mathcal{L}_{\mathrm{BAR}}[K]$. For example, the first row says that there exist background theories K_1 and K_2 such that $\mathcal{L}_{\mathrm{MONOMIAL}} \trianglelefteq \mathcal{L}^1_{ij\text{-}\mathrm{DET}}[K_1]$ and $\mathcal{L}_{\mathrm{MONOMIAL}} \trianglelefteq \mathcal{L}^1_{l\text{-}\mathrm{LOCAL}}[K_2]$. Thus the left-hand column gives lower bounds on the expressiveness of a language, and hence on its learnability. We place $\trianglelefteq \mathcal{L}_{\mathrm{FOO}}$ on the far right of a row if for every family of languages $\mathcal{L}_{\mathrm{BAR}}$ in the row and *every* $K \in \mathcal{K}$, $\mathcal{L}_{\mathrm{BAR}}[K] \trianglelefteq \mathcal{L}_{\mathrm{FOO}}$. Thus this column gives upper bounds on expressiveness. The individual languages appearing in the table are summarized in Table 2.

The top third of the table lists languages restricted enough to be polynomially predictable; the bottom third lists languages not predictable, and hence not pac-learnable; and the middle third lists languages equivalent in expressive power to DNF. The predictability and learnability of languages in the middle third of the paper is thus equivalent to a difficult open problem in computational learning theory.

In more detail, the paper's contributions are in four principle areas. First, we investigated the learnability of determinate clauses of logarithmic depth. It was shown that this language is *not* predictable, even if one restricts logic programs to contain a single clause, and even if examples are taken from a uniform distribution. This results follows immediately from the fact that these programs can express any concept computed by a log-depth circuit.

Second, we considered relaxing the condition of determinacy. In particular, we analyzed the expressive power of indeterminate clauses with k free variables, and showed that this makes a single clause as hard to predict as DNF. Bounding indeterminacy by a constant l and imposing the additional restriction of polynomial literal support does lead to a predictable language: under these restrictions, a single clause is as hard to learn as l-term DNF. These results show a strong parallel between learning DNF and learning indeterminate clauses.

Third, we have defined two additional classes of logic programs that allow restricted use of indeterminacy, but are pac-learnable. In particular, we defined $\mathcal{L}^1_{l\text{-}\mathrm{LOCAL}}$, the class of logic programs consisting of a single clause with locality bounded by a constant l, and proved it to be pac-learnable; we also defined $\mathcal{L}^*_{l\text{-}\mathrm{LENGTH}}$, the class of logic programs consisting of any number of clauses of length l or less, and showed that this class is pac-learnable. These classes can easily be shown to be incomparable to the class of ij-determinate programs defined by Muggleton.

Finally, we obtained a number of results on the learnability of logic programs using recursion. Recursion turns out to be a surprisingly powerful construct, making even highly restricted languages of logic programs very hard to predict. If even linear recursion is allowed, logic programs with an arbitrary number of clauses are not predictable even if the clauses are both ij-determinate and of bounded length. If arbitrary recursion is allowed, logic programs with a *single* clause are hard to predict, even if the clause is ij-determinate and of bounded locality.

One positive result is presented elsewhere: if one defines a recursive program to be

$\mathcal{L}_{\text{MONOMIAL}}$ *(monomials)*: boolean functions of the form $l_{i_1} \wedge l_{i_2} \wedge \ldots \wedge l_{i_r}$ where each l_{i_j} is either a variable v or its negation.

$\mathcal{L}_{\text{DNF}}$ *(Disjunctive normal form):* boolean functions of the form $M_{i_1} \wedge l_{i_2} \wedge \ldots \wedge M_{i_r}$ where each M_{i_j} is a monomial of any length.

$\mathcal{L}_{k\text{-DNF}}$ *(k-DNF)*: boolean functions of the form $M_{i_1} \vee M_{i_2} \vee \ldots \vee M_{i_r}$ where each M_{i_j} is a monomial of length k or less.

$\mathcal{L}_{k\text{-term-DNF}}$ *(k-term DNF)*: boolean functions of the form $M_{i_1} \vee M_{i_2} \vee \ldots \vee M_{i_k}$ where each M_{i_j} is a monomial of any length (but there no more than k such monomials.)

$\mathcal{L}_{\log n\text{-CIRC}}$ *(log depth circuits)*: boolean functions computed by a circuit or binary AND, OR and unary NOT gates that has depth $\log n$ or less (where n is the number of inputs.)

$\mathcal{L}_{\text{DLOG}}[n]$ *(deterministic log space)*: functions over strings of length n or less that can be computed by a log-space bounded Turing machine

$\mathcal{L}^1_{ij\text{-DET}}$ *(ij-determinate one-clause logic programs)*: logic programs containing a single nonrecursive ij-determinate clause. Related languages are

$\mathcal{L}^k_{ij\text{-DET}}$: allows up to k nonrecursive ij-determinate clauses

$\mathcal{L}^1_{\log n_e j\text{-DET}}$: allows a single determinate clauses with depth bounded by $\log maxi$

$\mathcal{L}^{1,rec}_{ij\text{-DET}}$: allows a single recursive ij-determinate clauses

$\mathcal{L}^{*,linrec}_{ij\text{-DET}}$: allows any number of linearly recursive ij-determinate clauses

$\mathcal{L}^1_{l\text{-LOCAL}}$ *(local one-clause logic programs)*: logic programs containing a single nonrecursive clause of locality l or less. Related languages are

$\mathcal{L}^k_{l\text{-LOCAL}}$: allows up to k nonrecursive clauses of locality l or less

$\mathcal{L}^{1,rec}_{l\text{-LOCAL}}$: allows a single recursive clause of locality l or less

$\mathcal{L}^*_{l\text{-LENGTH}}$ *(length bounded one-clause logic programs)*: logic programs containing a single clause of length l or less. Related languages are

$\mathcal{L}^{*,linrec}_{l\text{-LENGTH}}$: allows up any number of recursive clauses of length l or less

$\mathcal{L}^1_{k\text{-FREE}}$ *(k-free one-clause logic programs)* logic programs containing a single nonrecursive clause with at most k "free" variables.

$\mathcal{L}^1_{k\text{-INDET-PLS}}$ *(one-clause logic programs with polynomial literal support)*: logic programs containing a single nonrecursive k-indeterminate clause from some language with pls. Related languages are

$\mathcal{L}^{k'}_{k\text{-INDET-PLS}}$: allows up to k such clauses, as long as they are not recursive.

$\mathcal{L}^{*,linrec}_{k\text{-INDET-PLS}}$: allows any number of such such clauses, and also allows linear recursion.

Table 2: Languages studied and their definitions

closed if every recursive literal contains no output variables, then the class of one-clause ij-determinate closed linear recursive programs is pac-learnable. We leave open the learnability of less restricted classes of recursive logic programs: for example, linearly recursive programs with a bounded number of clauses may be predictable. We also leave open the learnability of recursive programs under distributional assumptions, as our results for recursive programs hold only in a distribution-free setting.

7.2 Related work

Much research has been done in the area of inductive logic programming; a substantial fraction of this work is formal in nature. However, almost all formal analyses of learnability have either assumed the ability to make queries, or have considered only weak criteria of convergence (such as learnability in the limit); there have been relatively few formal analyses in the more stringent model of pac-learnability from examples alone.

One previous result in this model shows the pac-learnability of a single "constrained atom" [Frisch and Page, 1991]; this was later generalized to a single ij-determinate Horn clause, or (against simple distributions) a nonrecursive program containing k such clauses [Džeroski *et al.*, 1992]. The latter result is of interest because the property of ij-determinacy is made use of by several practical learning systems [Muggleton and Feng, 1992; Cohen, 1993a; Lavrač and Džeroski, 1992; Quinlan, 1991], and is also used extensively in this paper: indeed, this paper can be considered an investigation of the degree to which this language can be generalized while preserving pac-learnability.[13]

Some previous negative results also exist. There are a number of results on the tractability of the *relative least general generalization (rlgg)* operator for Horn clauses, which can be used as a learning algorithm; with general background knowledge, even of restricted kind described here, rlgg is known to be intractable [Buntine, 1988; Plotkin, 1969]. Some algorithm-independent negative learnability results are also known. Frisch and Page [1990] describe a special logic consisting of atoms over typed variables and show that it is not pac-learnable; Haussler [1989] has also shown that the language of "existential conjunctive concepts", which are closely related to indeterminate Horn clauses, are not pac-learnable. Recently, some additional negative results also have been obtained for more conventional logics: Kietz [1993] has shown that a single clause is not pac-learnable if it is determinate but of unbounded depth, or of bounded depth but indeterminate.

However, while these results do not depend on a particular learning algorithm (such as rlgg) being used, they do rely heavily on the constraint that the learner must produce a single clause in the designated language as its hypothesis; this constraint is not obeyed by most practical ILP learning systems. Such representation-dependent results can lead to an apparent paradox, in which pac-learning is made possible by generalizing, rather than restricting, the language to be learned: for example, the pac-learnable language of constrained atoms is strictly more general than the language of typed atoms. Another example is k-term DNF, which is NP-hard to learn if the hypotheses of the learning system must be k-term DNF, but is tractably learnable if hypotheses are expressed in k-CNF.

[13]Džeroski, Muggleton and Russell [1992] also consider query algorithms for pac-learning recursive programs. In this paper, however, we have considered only "passive" learning algorithms that make no queries.

Our hardness results, in contrast, make no such representational assumptions; thus if a language $\mathcal{L}$ is not predictable, no language $\mathcal{L}'$ with greater expressive power than $\mathcal{L}$ can be predictable. Also, while like Kietz we consider conventional logic programs, our analysis considers finer-grained extensions of the ij-determinacy conditions: in particular we analyze the learnability of clauses of logarithmic depth, rather than unbounded depth, and clauses with indeterminacy bounded by a constant, rather than unbounded indeterminacy. Finally, we have also analyzed the effect of adding recursion to a single ij-determinate clause.

8 Conclusions

Most implemented first-order learning systems use restricted logic programs to represent concepts. An obvious advantage of this representation is that its semantics and complexity are mathematically well-understood; this suggests that learning systems using such logics can also be mathematically analyzed. This paper has sought to expand the theoretical foundations of this subfield, *inductive logic programming*, by formally investigating the learnability of restricted logic programs. Most of our analysis is using the model of *polynomial predictability* introduced by Pitt and Warmuth [1990]. This model encourages analyzing the learnability of a language $\mathcal{L}$ by characterizing the *expressive power* of $\mathcal{L}$.

In the paper we have (at least partially) characterized several extensions of the language of determinate clauses of constant depth [Muggleton and Feng, 1992; Džeroski *et al.*, 1992]. Via a reduction from log-depth circuits, we showed that a single log-depth determinate clause is not pac-learnable. Relaxing instead the condition of determinacy, we showed that a single clause with k free variables is as hard to learn as DNF, but that additionally bounding the indeterminacy of the clause and adding the restriction of polynomial literal support leads to a predictable (but not pac-learnable) language. It was also shown, via a reduction from log-space Turing machines, that a single *recursive* constant depth determinate clause is not pac-learnable, under cryptographic assumptions. Taken together, these results provide some upper bounds on the sorts of logic programs that can be tractably learned. Importantly, these negative results are not dependent on some particular choice of representation for the hypotheses.

In obtaining these results, several previous results from the literature have been extended. Haussler [1989] raises the question of the learnability of existential conjunctive concepts in a representation-independent (i.e. predictability) setting. It is not hard to show that every existential conjunctive concept can be expressed by a single indeterminate clause; thus an immediate result of Theorem 6 is that these concepts are in general as hard to predict as DNF, but are predictable if the number of possible bindings for the variables is bounded by a constant. Also, Džeroski, Muggleton and Russell [1992] describe a query-based algorithm for pac-learning k-clause ij-determinate recursive programs, but leave open the question of the learnability of this language without queries. A consequence of Theorem 10 is that no such algorithm exists, given cryptographic assumptions.

The paper also gives positive results for several new classes of logic programs. As stated above, language of clauses with bounded indeterminacy can be *predicted*, meaning that highly predictive hypotheses can be formed in a language other than the language of k-indeterminate clauses. The prediction algorithms are based on simple transformations of the problem, and could be implemented easily. Additionally, based on some insights obtained in the analysis of indeterminate clauses, two additional classes of logic programs were defined that allow indeterminacy, but that are pac-learnable: single-clause logic programs with bounded locality, and logic programs containing clauses of bounded length. We have also discussed (briefly) the possibility of combining the restriction of bounded locality with ij-determinism, leading to a pac-learnable language more powerful than either.

Two substantial open problems are suggested by this work. The first is finding additional syntactic restrictions on logic programs, such as locality, that extend expressive power without sacrificing learnability. One reason that this problem is difficult is that to be useful in a practical system, such syntactic restrictions must be relatively simple to describe; it is possible that to obtain restrictions that are both syntactically simple and closely related to learnability, it will be necessary to consider a substantially different syntax [Cohen and Hirsh, 1992].

Second, the results on learning recursive logic programs leave open the possibility that interesting classes of recursive programs are learnable; for example, the learnability of linearly recursive ij-determinate programs with a fixed number of clauses is open. Obtaining a more precise understanding of the precise boundaries of learnability for recursive programs appears to be an interesting and challenging formal problem.

References

(Boppana and Sipser, 1990) R. Boppana and M. Sipser. The complexity of finite functions. In *Handbook of Theoretical Computer Science*, pages 758–804. Elsevier, 1990.

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

(Cohen and Hirsh, 1992) W. Cohen and H. Hirsh. Learnability of description logics. In *Proceedings of the Fourth Annual Workshop on Computational Learning Theory*, Pittsburgh, Pennsylvania, 1992. ACM Press.

(Cohen, 1992) William W. Cohen. Compiling knowledge into an explicit bias. In *Proceedings of the Ninth International Conference on Machine Learning*, Aberdeen, Scotland, 1992. Morgan Kaufmann.

(Cohen, 1993a) William Cohen. Explicit biases and inductive logic programming. In preparation, 1993.

(Cohen, 1993b) William Cohen. A pac-learning algorithm for a restricted class of recursive logic programs. Submitted to AAAI-93, 1993.

(Džeroski *et al.*, 1992) Savso Džeroski, Stephen Muggleton, and Stuart Russell. Pac-learnability of determinate logic programs. In *Proceedings of the 1992 Workshop on Computational Learning Theory*, Pittsburgh, Pennsylvania, 1992.

(Frisch and Page, 1990) A. Frisch and C. D. Page. Generalization with taxonomic information. In *Proceedings of the Eighth National Conference on Artificial Intelligence*, Boston, Massachusetts, 1990. MIT Press.

(Frisch and Page, 1991) A. Frisch and C. D. Page. Learning constrained atoms. In *Proceedings of the Eighth International Workshop on Machine Learning*, Ithaca, New York, 1991. Morgan Kaufmann.

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

(Kearns and Valiant, 1989) Micheal Kearns and Les Valiant. Cryptographic limitations on learning Boolean formulae and finite automata. In *21th Annual Symposium on the Theory of Computing.* ACM Press, 1989.

(Kearns *et al.*, 1987) Micheal Kearns, Ming Li, Leonard Pitt, and Les Valiant. On the learnability of boolean formulae. In *19th Annual Symposium on the Theory of Computing.* ACM Press, 1987.

(Kharitonov, 1992) Michael Kharitonov. Cryptographic lower bounds on the learnability of boolean functions on the uniform distribution. In *Proceedings of the Fourth Annual Workshop on Computational Learning Theory*, Pittsburgh, Pennsylvania, 1992. ACM Press.

(Kietz and Morik, 1991) Jorg-Uwe Kietz and Katharina Morik. Constructive induction of background knowledge. In *Proceedings of the Workshop on Evaluating and Changing Representation in Machine Learning (at the 12th International Joint Conference on Artificial Intelligence)*, Sydney, Australia, 1991. Morgan Kaufmann.

(Kietz, 1993) Jorg-Uwe Kietz. Some computational lower bounds for the computational complexity of inductive logic programming. In *Proceedings of the 1993 European Conference on Machine Learning*, Vienna, Austria, 1993. To appear.

(Lavrač and Džeroski, 1992) Nada Lavrač and Sašo Džeroski. Background knowledge and declarative bias in inductive concept learning. In K. P. Jantke, editor, *Analogical and Inductive Inference: International Workshop AII'92.* Springer Verlag, Daghstuhl Castle, Germany, 1992. Lecture in Artificial Intelligence Series #642.

(Lloyd, 1987) J. W. Lloyd. *Foundations of Logic Programming: Second Edition.* Springer-Verlag, 1987.

(Muggleton and Feng, 1992) Steven Muggleton and Cao Feng. Efficient induction of logic programs. In *Inductive Logic Programming.* Academic Press, 1992.

(Muggleton, 1992a) S. H. Muggleton, editor. *Inductive Logic Programming.* Academic Press, 1992.

(Muggleton, 1992b) Steven Muggleton. Inductive logic programming. In *Inductive Logic Programming.* Academic Press, 1992.

(Pazzani and Kibler, 1992) Michael Pazzani and Dennis Kibler. The utility of knowledge in inductive learning. *Machine Learning*, 9(1), 1992.

(Pitt and Warmuth, 1988) L. Pitt and M. K. Warmuth. Reductions among prediction problems: On the difficulty of predicting automata. In *Proceedings of the 3rd Annual IEEE Conference on Structure in Complexity Theory*, Washington, D.C., 1988. Computer Society Press of the IEEE.

(Pitt and Warmuth, 1990) Leonard Pitt and Manfred 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. Ross Quinlan. Learning logical definitions from relations. *Machine Learning*, 5(3), 1990.

(Quinlan, 1991) J. Ross Quinlan. Determinate literals in inductive logic programming. In *Proceedings of the Eighth International Workshop on Machine Learning*, Ithaca, New York, 1991. Morgan Kaufmann.

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

(Vilain *et al.*, 1990) Marc Vilain, Phyllis Koton, and Melissa Chase. On analytical and similarity-based classification. In *Proceedings of the Eighth National Conference on Artificial Intelligence*, Boston, Massachusetts, 1990. MIT Press.