ELSEVIER

Artificial Intelligence 79 (1995) 1–38

Artificial Intelligence

Pac-learning non-recursive Prolog clauses

William W. Cohen *

AT&T Bell Laboratories, 600 Mountain Avenue, Murray Hill, NJ 07974, USA

Received August 1993; revised April 1994

Abstract

Recently there has been an increasing amount of research on learning concepts expressed in subsets of Prolog; the term *inductive logic programming* (*ILP*) has been used to describe this growing body of research. This paper seeks to expand the theoretical foundations of ILP by investigating the pac-learnability of logic programs. We focus on programs consisting of a single function-free non-recursive clause, and focus on generalizations of a language known to be pac-learnable: namely, the language of determinate function-free clauses of constant depth. We demonstrate that a number of syntactic generalizations of this language are hard to learn, but that the language can be generalized to clauses of constant locality while still allowing pac-learnability. More specifically, we first show that determinate clauses of log depth are not pac-learnable, regardless of the language used to represent hypotheses. We then investigate the effect of allowing indeterminacy in a clause, and show that clauses with k indeterminate variables are as hard to learn as DNF. We next show that a more restricted language of clauses with bounded indeterminacy is learnable using k-CNF to represent hypotheses, and that restricting the "locality" of a clause to a constant allows pac-learnability even if an arbitrary amount of indeterminacy is allowed. This last result is also shown to be a strict generalization of the previous result for determinate function-free clauses of constant depth. Finally, we present some extensions of these results to logic programs with multiple clauses.

Keywords: Machine learning; inductive logic programming; pac-learning

1. Introduction

Recently there has been an increasing amount of research on learning concepts expressed in first-order logics. While some researchers have considered special-purpose logics such as description logics as a representation for concepts and examples [15, 29, 51] most researchers have used standard first-order logic as a representation

* E-mail: wcohen@research.att.com.

SSDI 0004-3702(94)00034-X

language; in particular, most have used restricted subsets of Prolog to represent concepts [7, 37, 39, 40, 46]. 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. A number of formal results have in fact been obtained; in particular, a number of previous researchers have derived learnability results in Valiant's [50] model of pac-learnability [16, 21, 23, 27]. This paper seeks to expand the theoretical foundations of ILP by further investigating the pac-learnability of logic programs. In particular, our goal is to investigate carefully the degree to which the representational restrictions imposed by certain practical systems are necessary; in other words, we wish to determine the representational "boundaries of learnability" for logic programs, rather than to analyze existing learning systems.

In this paper we will consider primarily logic programs consisting of a single function-free non-recursive clause. We focus on single clauses because the results that are obtainable on the learnability of multiple-clause programs are straightforward extensions of results for single clauses [10, 16]. We consider only non-recursive clauses here because analysis of recursive programs requires somewhat different formal machinery [8, 10]; we also note that recursion has not been important in several applications of ILP methods to real-world problems [17, 30, 35, 38]. A final restriction is that while background knowledge will be allowed in our analysis, we will allow only background theories of ground unit clauses (also known as a database or a model). This restriction has also been made by several practical learning systems [37, 40, 46].

In this paper, we will first define pac-learnability and review previous learnability results for logic programs: the most important of these (for the purpose of this paper) shows that a single determinate function-free clause of constant depth [1] is pac-learnable [16]. We then investigate a number of generalizations of this language, beginning with the language of determinate clauses of logarithmic depth (rather than constant depth). We show that this language is not pac-learnable, regardless of the language used to represent hypotheses.

We then investigate the effect of allowing indeterminacy in a clause, and obtain a series of results for indeterminate clauses. We show that clauses with even a single "free" variable are as hard to learn as DNF, and that a slightly more restricted language of clauses with "bounded indeterminacy" is not pac-learnable, but is predictable, using k-CNF to represent hypotheses. Both of these results are negative, as they demonstrate that apparently reasonable languages are surprisingly hard to learn. However, we next show that bounding the "locality" of a clause allows pac-learnability, even if an arbitrary amount of indeterminacy is allowed.

Our last result on indeterminate clauses concerns the relative expressive power of ij-determinate clauses and clauses with bounded locality: we show that for fixed i and j, every ij-determinate clause can be rewritten as a clause with locality no greater than j^{i+1}. Thus, the language of clauses of bounded locality is, in a very reasonable sense, a strict generalization of the language of ij-determinate clauses.

[1] These restrictions are precisely defined in Section 2.5.

To summarize these results, we show that although the obvious syntactic generalizations of ij-determinacy all fail to produce pac-learnable languages, generalizing to the language of clauses bounded locality does yield a pac-learnable language.

Finally, we state some additional results on the learnability of multiple-clause non-recursive programs, discuss related work, and conclude.

A number of the results in this paper have been previously presented in a preliminary form elsewhere [8, 9, 12]. The pac-learnability of recursive programs, another interesting formal issue, is considered in depth in two companion papers [13, 14].

2. Preliminaries

2.1. Logic programming

In this section, we will give an overview of logic programming. As we are considering very simple logic programs, our overview has been simplified accordingly; in particular, the definitions below only coincide with the usual ones for the case of non-recursive function-free single-clause Prolog programs. For a more complete description of logic programming the reader is referred to one of the standard texts (e.g., [34]).

Logic programs are written over an alphabet of *constant symbols*, which we will usually represent with letters like $t_1, t_2, \ldots$; an alphabet of *predicate symbols*, which we will usually represent with letters like p, q or r; and an alphabet of *variables*, which we will always represent with upper-case letters. A *literal* is written $p(X_1, \ldots, X_k)$ where p is a predicate symbol and $X_1, \ldots, X_k$ are variables. The number of arguments k in a literal is called the *arity* of the literal. A *fact* is written $p(t_1, \ldots, t_k)$ where p is a predicate symbol and $t_1, \ldots, t_k$ are constant symbols; again, the arity of a fact is the number of arguments k. A *substitution* is a partial function mapping variables to constant symbols or variables; we will represent substitutions with the Greek letters θ and σ and (when necessary) write them as sets $\theta = \{X_1 = t_1, X_2 = t_2, \ldots, X_n = t_n\}$ where t_i is the constant symbol onto which X_i is mapped. If θ is a substitution and A is a literal, we will use $A\theta$ to denote the result of replacing each variable X in A with the constant symbol to which X is mapped by θ; extending this notation slightly, if θ_1 and θ_2 are both substitutions, we will use $A\theta_1\theta_2$ to denote $(A\theta_1)\theta_2$.

A fact f is an *instance* of a literal A if there is some substitution θ such that $A\theta = F$. If θ_1 and θ_2 are substitutions such that $\theta_1 \subseteq \theta_2$, then we say that θ_1 is *more general than* θ_2. Notice that if θ_1 is more general than θ_2, then for any literal A, the set of instances of $A\theta_1$ is a superset of the set of instances of $A\theta_2$.

Finally, a *definite clause* is written $A \leftarrow B_1 \wedge \cdots \wedge B_l$ where A and $B_1, \ldots, B_l$ are literals. A is called the *head* of the clause, and the conjunction $B_1 \wedge \cdots \wedge B_l$ is called the *body* of the clause. If DB is a set of facts—which we will also call a *database*—then the *extension* of a clause $A \leftarrow B_1 \wedge \cdots \wedge B_l$ with respect to the database DB is the set of all facts f such that either

- $f \in$ DB, or
- there exists a substitution θ so that $A\theta = f$, and for every B_i from the body of the clause, $B_i\theta \in$ DB.

In the latter case, we will say that the substitution θ *proves* f to be in the extension of the clause. For brevity, we will let $ext(C, \mathrm{DB})$ denote the extension of C with respect to the database DB.

For technical reasons, it will be convenient to assume that every database DB contains an *equality predicate*—that is, a predicate symbol *equal* such that $equal(t_i, t_i) \in \mathrm{DB}$ for every constant t_i appearing in DB, and $equal(t_i, t_j) \notin \mathrm{DB}$ for any $t_i \neq t_j$. This assumption can be made without loss of generality, since such a predicate can be added to any database with only a polynomial increase in size.

Readers familiar with logic programming will notice that this definition of "extension" coincides with the usual fixpoint or minimal-model semantics of Prolog programs for the programs considered in this paper (i.e., single-clause function-free non-recursive Prolog programs over a ground background theory). Hence one might more succinctly define the extension of a clause C with respect to DB as $\{f : C \wedge \mathrm{DB} \vdash f\}$.

Again for those familiar with first-order logic, a clause $A \leftarrow B_1 \wedge \cdots \wedge B_l$ can also be thought of as a logical statement

$$\forall X_1, \ldots, X_n (\neg B_1 \vee \cdots \vee \neg B_l \vee A)$$

where $X_1, \ldots, X_n$ are the variables that appear in the clause. Then the extension of a clause with respect to DB is simply the set of facts e that follow from the logical statement above and the conjunction of the facts in DB.

Example. If DB is the set

$$\mathrm{DB} = \{\mathrm{mother(ann,bob)}, \mathrm{father(bob,julie)}, \mathrm{father(bob,chris)}\}$$

then the extension of the clause

$$\mathrm{grandmother}(X, Y) \leftarrow \mathrm{mother}(X, Z), \mathrm{father}(Z, Y)$$

with respect to DB is the set

$$\mathrm{DB} \cup \{\mathrm{grandmother(ann,julie)}, \mathrm{grandmother(ann,chris)}\}.$$

(Notice that we have adopted the convention that a function f is represented by a predicate $f(X, Y)$ where $f(X, Y)$ is true iff $Y = f(X)$.) The most general substitutions that prove the additional facts

$$f_1 = \mathrm{grandmother(ann,julie)},$$

$$f_2 = \mathrm{grandmother(ann,chris)}$$

in the extension are

$$\theta_1 = \{X = \mathrm{ann}, Y = \mathrm{julie}, Z = \mathrm{bob}\},$$

$$\theta_2 = \{X = \mathrm{ann}, Y = \mathrm{chris}, Z = \mathrm{bob}\}.$$

2.2. Models of learnability

Our goal is to determine by formal analysis which subsets of Prolog are efficiently learnable; we focus in this paper on the case of function-free non-recursive Prolog. Any formal analysis of learnability, of course, requires an explicit model of what it means for a language to be "efficiently learnable". In this section, we will describe our basic models of learnability; these are slight modifications of the models of pac-learnability, introduced by Valiant [50], and polynomial predictability, introduced by Pitt and Warmuth [44].

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* LANG to be a set of concepts. Associated with X and LANG are two *size complexity measures*. We will write the size complexity of some concept $C \in$ LANG or instance $e \in X$ 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 notation X_n (respectively LANG_n) to stand for the set of all elements of X (respectively LANG) 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; when there is a risk of confusion we will refer to the set represented by a concept C as the *extension of* C.

Example. For example, let X be the domain of binary vectors, interpreted as assignments to boolean variables, and let DNF be the language of boolean formulae 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. Thus for the instance $e = 00110$ we have $\|e\| = 5$, and for the concept $C = ((x_1 \wedge \overline{x_5}) \vee (\overline{x_1} \wedge x_5))$ we have $\|C\| = 4$.

An *example of* C is a pair (e, b) where $b = 1$ if $e \in C$ 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 our basic learning models.

Definition 1 (*Polynomially predictable*). A language LANG is *polynomially predictable* iff there is an algorithm PACPREDICT and a polynomial function $m(\frac{1}{\varepsilon}, \frac{1}{\delta}, n_e, n_t)$ so that for every $n_t > 0$, every $n_e > 0$, every $C \in \text{LANG}_{n_t}$, every $\varepsilon: 0 < \varepsilon < 1$, every $\delta: 0 < \delta < 1$, and every probability distribution function D, PACPREDICT has the following behavior:

(1) given a sample S^+, S^- of C from X_{n_e} drawn according to D and containing at least $m(\frac{1}{\varepsilon}, \frac{1}{\delta}, n_e, n_t)$ examples, PACPREDICT outputs a hypothesis H such that

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

where the probability is taken over the possible samples S^+ and S^- and (if PACPREDICT is a randomized algorithm) over any coin flips made by PACPREDICT;

(2) PACPREDICT runs in time polynomial in $\frac{1}{\varepsilon}$, $\frac{1}{\delta}$, n_e, n_t, and the number of examples; and

(3) H can be evaluated in polynomial time.

The algorithm PACPREDICT is called a *prediction algorithm* for LANG, and the function $m(\frac{1}{\varepsilon}, \frac{1}{\delta}, n_e, n_t)$ is called the *sample complexity* of PACPREDICT.

We will sometimes abbreviate "polynomial predictability" as "predictability".

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 usable in the very weak sense that it can be used to make predictions in polynomial time. Notice that this is a *worst-case* learning model, as the definition allows an adversarial choice of all the inputs of the learner.

The model of polynomial predictability has been well studied [44], and is a weaker version of Valiant's [50] criterion of *pac-learnability*:

Definition 2 (*Pac-learnable*). A language LANG is *pac-learnable* iff there is an algorithm PACLEARN so that

(1) PACLEARN satisfies all the requirements in the definition of polynomial predictability, and

(2) on inputs S^+ and S^-, PACLEARN always outputs a hypothesis $H \in$ LANG.

Thus if a language is pac-learnable it is predictable, but the converse need not be true.

Predictability also has an important property not shared by pac-learnability: if a language is *not* predictable, then no superset of that language is predictable. In other words, one cannot make a non-predictable language predictable by generalizing the language, only by adding additional restrictions. Showing a language is not predictable indicates that the language is, in some sense, too expressive to learn efficiently, and hence is a strong negative result.

On the other hand, in ILP contexts, it is often considered desirable to output hypotheses that are logic programs; hence a polynomial prediction algorithm, which may output hypotheses in an arbitrary format, may be much less desirable than a pac-learning algorithm. Thus ideally one would like all positive results to be given in the pac-learning model, and all negative results to be given in the polynomial prediction model. In this paper we will (whenever possible) give positive results in the pac-learning model, and use predictability primarily in negative results.

2.3. Background knowledge: extending the standard models

So far, our formalization is standard. However, in a typical ILP system, the user provides both a set of examples and a "background theory" defining a set of predicates that may be useful in constructing a hypothesis: the task of the learner is then to find a

logic program P such that P, together with the background theory, is a good model of the data.

To account for the background knowledge, it is necessary to extend the model of learnability. One way of doing this is to allow examples to be clauses that are entailed by the target concept [20,27,45]. However, in this paper, we will follow Haussler [23] and Džeroski et al. [16] in using a closely related formalism which more directly models the typical use of background knowledge in ILP systems.

If LANG is some set of definite clauses and DB is a database, then LANG[DB] denotes the set of pairs of the form (C, DB) such that $C \in$ LANG. Each such pair represents the extension of C with respect to DB, as defined in Section 2.1—i.e., the set of all facts e such that $C \wedge \text{DB} \vdash e$. If $\mathcal{DB}$ is some set of databases, then LANG[$\mathcal{DB}$] denotes the set of all languages LANG[DB] where DB $\in \mathcal{DB}$. Such a set of languages will be called a *language family*. The set of definite clauses LANG will be called a *clause language*.

In this paper, we will consider primarily the learnability of language families, using learning algorithms that accept a database as input in addition to the usual set of training examples. The following definitions extend the notions of pac-learnability and polynomial predictability to this new setting.

Definition 3. A language family LANG[$\mathcal{DB}$] is *polynomially predictable* iff for every DB $\in \mathcal{DB}$ there is a prediction algorithm PACPREDICT$_{\text{DB}}$ for LANG[DB].

A language family LANG[$\mathcal{DB}$] is *uniformly polynomially predictable* iff it is polynomially predictable and there is an algorithm PACPREDICT(DB, S^+, S^-), which runs in time polynomial in all of its inputs, such that PACPREDICT, with its first argument fixed to be DB, is a prediction algorithm for LANG[DB].

The (*uniform*) *pac-learnability* of a language family is defined analogously.

Intuitively, a language family is predictable if it can be predicted regardless of the database DB, and a language family is uniformly predictable if there is a single prediction algorithm that works for all databases.

Notice that PACPREDICT(DB, S^+, S^-) must run in time polynomial in *all* of its inputs, including the size of the database DB. Thus uniform predictability (and pac-learnability) requires the prediction (or learning) algorithm to scale well with the size of the background database DB.

Finally, let us define a-$\mathcal{DB}$ to be the set of databases containing only facts of arity a or less. Most of the results in this paper will be in one of the following forms:

- For any fixed constant a, LANG[a-$\mathcal{DB}$] is uniformly pac-learnable.
  Such a result means that even if one allows an adversary choice of the database DB, clauses in LANG[DB] are pac-learnable, that there is a known learning algorithm that works for any database DB, and furthermore that the algorithm requires time only polynomial in the size of the database DB. (However, it may require time exponential in the maximum arity a of facts in the database.) This is a strong positive result about the learnability of clauses in LANG.
- For every $a \geqslant a_0$ (where a_0 is some small fixed constant, say $a_0 = 3$) LANG[a-$\mathcal{DB}$] is not predictable.

Such a result means that for at least some databases DB $\in$ a-$\mathcal{DB}$, clauses in LANG[DB] are not predictable (and hence not pac-learnable regardless of the representation used for hypotheses). This is a negative result about the learnability of clauses in LANG.

The notions of uniform pac-learnability and predictability extend the standard models to the ILP setting, where the database is an additional input to the learner. The standard models are worst case over all distributions and all target concepts; we have simply made the learning model worst case also over all possible choices of a database. At first glance, it may seem odd to allow an adversarial choice of the database. This is reasonable, however, because if the database DB is such that the target concept cannot be expressed using predicates defined in DB, or is only expressible by an extremely large concept, then the learning system is not required to find an accurate hypothesis quickly (since time and sample complexity may grow with the size of the target concept $C \in$ LANG[DB]). Thus the model is actually worst case over all databases DB that are "appropriate" in the sense that a concise representation of the target concept can be found using the predicates defined in DB.

We will typically use n_b to denote the size of a database DB. The parameters n_e, n_t and n_b 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 $n = n_e + n_b + n_t$.

Example. As an example of an ILP learning problem, to learn the predicate *maternal_grandmother*, the user might provide the database

DB = { father(charlie,william), mother(charlie,susan),
father(susan,dan), mother(susan,ruth),
father(william,maurice), mother(william,caroline),
father(rachel,maurice), mother(rachel,caroline),
father(elizabeth,warren), mother(elizabeth,rachel) }

and the examples

S^+ = { maternal_grandmother(charlie,ruth),
maternal_grandmother(elizabeth,caroline) },

S^- = { maternal_grandmother(charlie,dan),
maternal_grandmother(william,caroline),
maternal_grandmother(ruth,dan),
maternal_grandmother(maurice,susan) }.

In this problem, the user's database DB is in 2-$\mathcal{DB}$, the size of the database is $n_b = 10$, and the size of the examples is $n_e = 2$. An ILP learning system for the clause language 1-DEPTHDETERM (see below for definition) might produce the hypothesis

H = maternal_grandmother(X, Y) $\leftarrow$ mother(X, Z) $\wedge$ mother(Z, Y).

If the learning system were a pac-learning system with a known sample complexity, then (if S^+ and S^- were sufficiently large, and drawn from a fixed distribution) one could make some guarantees about the error rate ε of the learner. Note, however, that the user provides only the inputs S^+, S^-, and the database DB.

2.4. Sample complexity of learning logic programs

In typical ILP problems the examples will all have the same predicate symbol p and arity n_e; thus, in effect, the predicate and arity of the head of the target clause are given. One important fact to note is the following.

Theorem 4. *Let* DATALOG$^{p/n_e}$ *be the language of all function-free non-recursive clauses that have a head with predicate symbol p and arity* n_e. *Then for any fixed constant a and any* DB $\in$ *a-DB, the Vapnik–Chervonenkis dimension* [4] *of* DATALOG$^{p/n_e}_{n_t}$[DB] *is polynomial in* n_e, n_t *and* n_b (*where* n_b *is the size of* DB).

Proof. We will establish an upper bound on the number of semantically different clauses. A DATALOG clause of size n_t can contain at most $n_e + an_t$ distinct variables, as at most n_e variables can appear in the head, and at most an_t variables can appear in the body; thus there are at most $(n_e + an_t)^{n_e}$ possible clause heads. Since there are at most n_b predicates that appear in the database, and each literal consists of one such predicate symbol and a or fewer variables, there are at most $n_b(n_e + an_t)^a$ literals that can appear in the body of a clause that succeeds with the database DB. Putting these two bounds together, the total number of semantically different clauses is

$$(n_e + an_t)^{n_e} \cdot (n_b(n_e + an_t)^a)^{n_t}.$$

The VC dimension is bounded by the logarithm of this quantity

$$n_e \log_2 (n_e + an_t) + n_t \log_2 (n_b(n_e + an_t)^a)$$

which is polynomial in n_b, n_e, and n_t. □

Blumer et al. [4] show that if a concept class has polynomial VC dimension, then for a certain polynomial sample size, *any* consistent hypothesis H of minimal or near-minimal size will with high confidence have low error. More specifically, any algorithm A that outputs a consistent hypothesis that is within a polynomial of the size of the smallest consistent hypothesis will satisfy all the requirements of pac-learning—except, perhaps, the requirement that the learner runs in polynomial time.

Thus, the following simple procedure will satisfy all of the requirements of uniform pac-learnability for DATALOG, except the requirement that the learning program be polynomial time: enumerate all non-vacuous DATALOG clauses in increasing order of size, and return the first clause that is consistent with the sample. Since this paper considers only languages that are restrictions of DATALOG, this means that if computational complexity is ignored, all of the languages considered in this paper are pac-learnable. The central question we address, then, is when polynomial-time learning is possible.

2.5. Constant-depth determinacy and previous results

Muggleton and Feng [37] have introduced several useful restrictions on definite clauses, which we will now describe. If $A \leftarrow B_1 \wedge \cdots \wedge B_r$ is an (ordered) definite clause, then the *input variables* of the literal B_i are those variables appearing in B_i that also appear in the clause $A \leftarrow B_1 \wedge \cdots \wedge B_{i-1}$; all other variables appearing in B_i are called *output variables*. A literal B_i is *determinate* (with respect to DB) if for every possible substitution σ that unifies A with some fact e such that $B_1\sigma \in \text{DB}$, $B_2\sigma \in \text{DB}, \ldots,$ and $B_{i-1}\sigma \in \text{DB}$ there is at most one substitution θ so that $B_i\sigma\theta \in \text{DB}$. Less formally, a literal is determinate if its output variables have only one possible binding, given DB and the binding of the input variables.

A clause is determinate if all of its literals are determinate. Informally, determinate clauses are those that can be evaluated without backtracking by a Prolog interpreter.

Next, define the *depth* of a variable appearing in a clause $A \leftarrow B_1 \wedge \cdots \wedge B_r$ 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.

Example. The clause

$$\text{maternal\_grandmother}(C, G) \leftarrow \text{mother}(C, M) \wedge \text{mother}(M, G)$$

is determinate (assuming *mother* is functional). The maximum depth of a variable is one, for the variable M, and hence the clause has depth one. Assuming that the predicates *enclosed_paper* and *length* are determinate, the clause

$$\begin{aligned}\text{unwelcome\_mail}(E) \leftarrow\ & \text{envelope}(E) \wedge \\ & \text{enclosed\_paper}(E, P) \wedge \text{must\_review}(P) \wedge \\ & \text{length}(P, L) \wedge \text{gt50}(L)\end{aligned}$$

is determinate and of depth two. The variable P from this clause has depth one, and the variable L has depth two.

An interesting class of logic programs is the following.

Definition 5 (*ij-determinate*). A determinate clause of depth bounded by a constant i over a database $\text{DB} \in j\text{-}\mathcal{DB}$ is called *ij-determinate.*

The learning program GOLEM, which has been applied to a number of practical problems [17, 30, 38], learns ij-determinate programs. Closely related restrictions also have been adopted by several other inductive logic programming systems, including FOIL [47] and LINUS [32].

The learnability of non-recursive ij-determinate clauses has also been formally studied [16]. For notation, let i-DEPTHDETERM be the language of determinate clauses of depth i or less; the language family of ij-determinate clauses is thus denoted i-DEPTHDETERM[$j\text{-}\mathcal{DB}$]. One important result is the following.

Theorem 6 (Džeroski, Muggleton and Russell [16]). *For any fixed i and j, the language family i-DEPTHDETERM[j-$\mathcal{DB}$] is uniformly pac-learnable.*

Other previous work has established that a single clause is *not* pac-learnable if the ij-determinacy condition does not hold; specifically, it has been shown that neither the language of indeterminate clauses of fixed depth nor the language of determinate clauses of arbitrary depth is pac-learnable [27]. 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).

Unfortunately, these negative results are of limited practical importance, because they only show learning to be hard when 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, not a single clause, and the results do *not* show that learning using this more expressive representation is intractable. Such negative learnability results are sometimes called *representation-dependent.*[2] One of the goals of this paper is to develop *representation-independent* learning results that complement the positive result of Theorem 6. These results will be developed shortly; first, however, we will describe the analytic tool used to obtain the results.

2.6. *Reducibility among prediction problems*

Pitt and Warmuth [44] have introduced a notion of reducibility between prediction problems, analogous to the notion of reducibility for decision problems that is commonly used to prove a problem NP-hard. *Prediction-preserving reducibility* is essentially a method of showing that one language is no harder to predict than another.

Definition 7 (*Prediction-preserving reducibility*). Let LANG_1 be a language over domain X_1 and LANG_2 be a language over domain X_2. We say that *predicting* LANG_1 *reduces to predicting* LANG_2, denoted $\text{LANG}_1 \trianglelefteq \text{LANG}_2$, if there is a function $f_i : X_1 \to X_2$, henceforth called the *instance mapping*, and a function $f_c : \text{LANG}_1 \to \text{LANG}_2$, 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;
(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.

Intuitively, $f_c(C_1)$ returns a concept $C_2 \in \text{LANG}_2$ that will "emulate" C_1—i.e., make the same decisions about concept membership—on examples that have been "prepro-

[2] The prototypical example of a learning problem that is hard in a representation-dependent setting but not in a broader setting is learning k-term DNF. Assuming that RP $\neq$ NP, pac-learning k-term DNF is intractable if the hypotheses of the learning system must be k-term DNF, but tractable if hypotheses can be expressed in the richer language of k-CNF [42].

cessed" with the function f_i. If predicting LANG_1 reduces to predicting LANG_2 and a learning algorithm for LANG_2 exists, then one possible scheme for learning concepts from LANG_1 would be the following. First, convert any examples of the unknown concept C_1 from the domain X_1 to examples over the domain X_2 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 LANG_2 should produce some hypothesis H that is a good approximation of $f_c(C_1)$. Of course, it may not be possible to map H back into the original language LANG_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 [43] give a more rigorous argument that this approach leads to a prediction algorithm for LANG_1, leading to the following theorem.

Theorem 8 (Pitt and Warmuth). *Assume that* $\text{LANG}_1 \trianglelefteq \text{LANG}_2$. *Then the following hold:*

- *If* LANG_2 *is polynomially predictable, then* LANG_1 *is polynomially predictable.*
- *If* LANG_1 *is not polynomially predictable, then* LANG_2 *is not polynomially predictable.*

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 prediction is as hard as breaking cryptographic schemes that are widely assumed to be secure. The first case of the theorem gives a means of obtaining a prediction algorithm for LANG_1, given a prediction algorithm for LANG_2.

If f_c is one-to-one and f_c^{-1} is computable, then the reduction is said to be "invertible"; in this case it can be shown that if LANG_2 is pac-learnable then LANG_1 is also pac-learnable. For example, the proof of Theorem 6 is based on an invertible prediction-preserving reduction between ij-determinate clauses and monotone monomials.

3. Log-depth clauses are hard to learn

In the next two sections, we will investigate the learnability of definite clauses in the models described above: pac-learnability and polynomial predictability. Our starting point will be Theorem 6—in particular, we will consider generalizing the result of Theorem 6 by generalizing the definition of ij-determinacy in various ways, and seeing if the corresponding languages are learnable.

We will first consider relaxing the restriction that clauses have constant depth. Muggleton and Feng [37] argue that many practically useful programs are limited in depth; however, in the list of clauses they provide as examples to support their argument, it is frequently the case that the more complex clauses have greater depth. It might be plausibly argued that it is more reasonable to assume that clause depth d is some slowly growing function of problem size (as measured by either clause size n_t, database size n_b, or example size n_e).

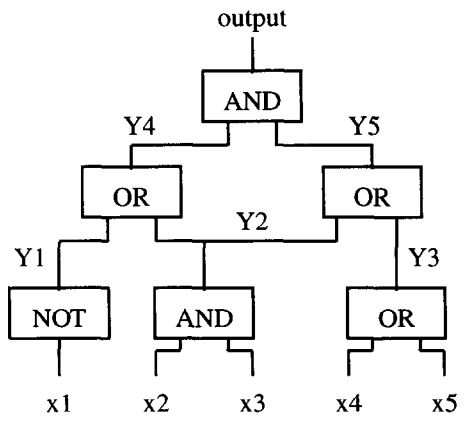


Fig. 1. Constructing a determinate clause equivalent to a circuit.

The key result of this section is that increasing the depth bound to be even logarithmic in the size of examples makes determinate clauses hard to predict.

Theorem 9. *For any constant $a \geqslant 3$, the language family*

$$(\log n_e)\text{-DEPTHDETERM}[a\text{-}\mathcal{DB}]$$

is not polynomially predictable, under cryptographic assumptions.[3]

Proof. The proof is based on a prediction-preserving reduction from boolean circuits of depth d to determinate clauses of depth d. Let d-CIRCUIT 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.[4] We will show that there exists a database $\text{DB}_{\text{CIR}} \in 3\text{-}\mathcal{DB}$, containing only eleven atomic facts, such that

$$d\text{-CIRCUIT} \trianglelefteq d\text{-DEPTHDETERM}[\text{DB}_{\text{CIR}}].$$

The expressive power of depth-bounded boolean circuits, as well as their learnability, has been well studied [5]; in particular it is known that log-depth circuits are hard to predict under cryptographic assumptions [25, Theorem 4]. Thus the theorem follows immediately from this reduction and Theorem 8.

The construction used in the reduction is illustrated in Fig. 1. An example for the circuit language is a binary vector $b_1 \ldots b_n$, which is converted by the instance mapping

[3] More precisely, the 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 [25]. These problems are widely conjectured to be intractable.

[4] Specifically, the extension of a circuit C is 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.

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 database DB_{CIR} contains definitions of the boolean functions *and*, *or*, and *not*, as well as a definition of the unary predicate *true*, which succeeds whenever its argument is a "1":

$$DB_{CIR} = \{ and(0,0,0),\ and(0,1,0),\ or(0,0,0),\ or(0,1,1),\ not(0,1),\ and(1,0,0),\ and(1,1,1),\ or(1,0,1),\ or(1,1,1),\ not(1,0),\ true(1) \}.$$

Finally, the concept mapping f_c is as indicated in the figure. To be precise, for each gate G_i in the circuit there is a single literal L_i with a single output variable Y_i, defined as

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

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

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

Notice that the construction preserves depth. □

The algorithm presented by Muggleton and Feng for learning a single ij-determinate clause is doubly exponential in the depth of the clause. The result above shows that no learning algorithm for determinate clauses exists that improves this bound much, e.g., that is even singly exponential in depth. The result holds even for learning systems that use an alternative representation for their hypotheses (e.g., systems that approximate one clause with several).

Recent work has shown that log-depth circuits are hard to predict even if examples are drawn from a uniform distribution [26].[5] Thus even making fairly strong assumptions about the distribution of examples will not make log-depth determinate clauses predictable.

4. Hard-to-learn indeterminate 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 relaxing

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

the second key aspect of the ij-determinacy restriction: the condition that clauses be determinate. While for many problems determinacy is an appropriate restriction, there are real-world problems for which some of the background knowledge cannot be accessed using only determinate literals [11]; thus for practical reasons, it would be useful to be able to relax this restriction.

In this section, we will consider several plausible ways to relax the determinacy restriction, and show that these relaxations lead to languages that are hard to pac-learn and (in most cases) also hard to predict. In particular, we first consider bounding the depth of a clause, but not restricting it in any other way, and show that this leads to a language that is hard to predict. We then consider bounding the number of "free" variables in an indeterminate clause, and show that this language is exactly as hard to predict as DNF. We then consider an alternative set of restrictions in which the degree of indeterminacy is also bounded, and show that that language is predictable, but not pac-learnable.

4.1. Constant-depth indeterminate clauses

The most obvious way of relaxing ij-determinacy would be to consider constant-depth clauses that are not determinate. Unfortunately, this leads to a language family that is hard to learn. Letting k-DEPTH denote the language of all clauses of depth k or less, we have the following result:

Theorem 10. *For $a \geqslant 3$ and $k \geqslant 1$, the language family k-DEPTH[a-$\mathcal{DB}$] is not predictable, unless* NP $\subseteq$ P/Poly.

Proof. Schapire [47] shows that if a language LANG is polynomially predictable, then every $C \in$ LANG can be emulated by a polynomial-sized circuit. To prove the theorem, therefore, it is sufficient to construct a polynomial sized database DB $\in 3$-$\mathcal{DB}$ such that for some $C \in 1$-DEPTH[DB] testing membership in C is NP-hard.

Let DB contain the following predicates:

- The predicate *boolean(X)* is true if $X = 0$ or $X = 1$.
- For $k = 1, \ldots, n$, 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 $X = \neg V$,
  - $M \neq k$ and $M \neq -k$ (and X and V have any values).
- Finally, 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 a 3-sat formula $\phi = \wedge_{i=1}^{n}(l_{i_1} \vee l_{i_2} \vee l_{i_3})$ over the n variables $x_1, \ldots, x_n$. We will encode this formula as the following 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 consider the clause C_{SAT} below:

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

The first two sets of literals introduce two sets of depth-1 indeterminate variables: the X_k's correspond to possible values for the variables $x_1, \ldots, x_n$ over which ϕ is defined, and the V_{i_j}'s correspond to values that can be assigned to the literals l_{i_j} that appear in ϕ. The third set of literals ensures that if $l_{i_j} = x_k$ then $V_{i_j} = X_k$, and that if $l_{i_j} = \overline{x_k}$ then V_{i_j} and X_k are complements; this conjunction thus ensures that the V_{i_j}'s have values consistent with some assignment to the x_i's. Finally, the last conjunction of literals ensures that the values given to the V_{i_j}'s are such that every clause in ϕ succeeds: *i.e.*, that ϕ is satisfied.

Thus we conclude that ϕ is satisfiable iff the clause C_{SAT} succeeds on the instance e_ϕ, and hence that determining whether C_{SAT} succeeds must be NP-hard.

Finally, notice that the *boolean* predicate requires two facts to define; the *sat* predicate requires seven facts to define; and (since each $link_k$ predicate requires at most $2n \cdot 2 \cdot 2 = 8n$ facts to define) the link predicates together require only $8n^2$ facts to define. Hence $\|DB\|$ is bounded by a polynomial in n. It is also clear that C_{SAT} is of size polynomial in n. This completes the proof. □

Therefore, in the remainder of this section, we will consider the learnability of languages strictly more restrictive than k-DEPTH. We will first consider the learnability of clauses with a bounded number of "free variables".

4.2. *Clauses with k free variables*

Let the *free variables* of a clause be those variables that appear in the body of the clause but not in the head. One reasonable restriction to impose is to consider clauses with only a small number of free variables. This restriction is analogous to that imposed by Haussler [23].

We will consider now the learnability of the language k-FREE, defined to be all non-recursive clauses containing at most k free variables. Notice that clauses in k-FREE are necessarily of depth at most k; also restricting the number of free variables ensures that clauses can be evaluated in polynomial time. While at first glance this language seems to be quite simple, notice that a clause $p(X) \leftarrow q(X, Y)$ classifies an example $p(a)$ as true

Database:

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:

$\text{dnf}(X_1, X_2, X_3, X_4) \leftarrow$

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

$\text{false}_2(X_2, Y) \wedge \text{false}_2(X_3, Y) \wedge$

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

Fig. 2. Constructing an indeterminate clause equivalent to a DNF formula.

exactly when $q(a, b_1) \in \text{DB} \vee \cdots \vee q(a, b_r) \in \text{DB}$, 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.

As it turns out, we can exploit the expressive power of indeterminate free variables to encode any boolean expression in disjunctive normal form[6] using a single k-free clause (over a suitable database). This leads to the following theorem.

Theorem 11. *For any constants $a \geqslant 2$ and $k \geqslant 1$, if the language family k-*FREE*$[a\text{-}\mathcal{DB}]$ is predictable, then* DNF *is predictable.*

Proof. As in Theorem 9, the statement of the theorem follows directly from a single reduction:

Lemma 12. *Let r-*TERMDNF *denote the language of r-term DNF formulae. There is a database* DB_r *of size polynomial in r such that r-*TERMDNF $\trianglelefteq$ 1-FREE$[\text{DB}_r]$.

(To see that the theorem follows from the lemma, notice that a DNF formula of complexity n can have at most n terms, and that any DNF formula with fewer than n terms can be padded to exactly n terms by adding terms of the form $v_1\overline{v_1}$; thus any DNF of size n_t or less could be predicted using 1-free clauses over the database DB_{n_t}.)

Proof. The construction on which this reduction is based is illustrated in Fig. 2. Let DB_r contain sufficient atomic facts to define the binary predicates $true_1, false_1, \ldots, true_r, false_r$ that behave as follows:

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

[6] Recall that boolean formulae of the form $\bigvee_i \bigwedge_j l_{ij}$ are said to be in *disjunctive normal form*. We denote this language as DNF, with the size measure for examples being the number of variables (i.e., the length of a bit vector encoding an assignment) and the size measure for a formula being the number of literals it contains.

Since $2r - 1$ facts are required to define each of these predicates, the total size of DB_r is $\mathrm{O}(r^2)$.

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

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

to the clause

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

where Lit_{ij} is defined as

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

Clearly $f_i(e)$ and $f_c(\phi)$ are of the size as e and ϕ respectively; since DB_r is also of polynomial size, this reduction is polynomial.

Next, notice that in $f_c(\phi)$ there is only one variable Y not appearing in the head, and (if the clause $f_c(\phi)$ is to succeed) it can be bound to only the r values $1, \ldots, r$. Thus 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. 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 $\bigwedge_{j=1}^{s_i} Lit_{ij}$ will fail. Thus concept membership is preserved by the mapping. □

It also should be noted that every clause in 1-FREE[DB_r] can be translated into an r-term DNF expression; thus Lemma 12, together with existing hardness results for pac-learning k-term DNF [24], leads to the following result.

Observation 13. *For $a \geqslant 2$ the language family* 1-FREE[a-$\mathcal{DB}$] *is not pac-learnable.*

It is straightforward to obtain a number of other similar representation-*dependent* hardness results for pac-learning clauses in k-FREE, somewhat along the lines of Theorem 1 of Kietz [27]. However, if one accepts the conjecture that learning DNF is hard, then these are of limited interest, given Theorem 11; hence we will not develop such results here. We turn instead to another question: whether there are languages in k-FREE that are *harder* to learn than DNF. The answer to this question is no:

Theorem 14. *If DNF is predictable then for all constants a and k, the language family k-*FREE[a-$\mathcal{DB}$] *is uniformly predictable.*

Proof. It suffices to show that for all constants a and k and every background theory $\mathrm{DB} \in a$-$\mathcal{DB}$

k-FREE[DB] $\trianglelefteq$ DNF

since if this reduction holds, one could use the hypothesized prediction algorithm for DNF to predict k-FREE[DB]. Below we will give such a reduction for an arbitrary database DB.

Let C be a clause in k-FREE[DB]. The predicate symbol and arity of the head of C can be determined from any of the positive examples, and because we assume that DB contains an equality predicate, one can also assume that all of the variables in the head of C are distinct.[7] Thus the head of C can be determined from the examples.

Notice also that each clause has at most $n_e + k$ variables, and hence there are only $(n_e + k)^a$ a-tuples of variables that could serve as arguments to a literal. Let the background database DB be of size n_b. Since the database DB contains at most n_b predicate symbols, there are at most $n_b \cdot (n_e + k)^a$ possible literals $B_1, \ldots, B_{n_b \cdot (n_e+k)^a}$ that can appear in the body of a k-FREE clause.

Now, let $C = A \leftarrow B_{c_1} \wedge \cdots \wedge B_{c_l}$ be a clause in k-FREE[DB]. Recall that C covers an example e iff there exists some substitution θ such that

$$B_{c_1}\sigma\theta_e \in \text{DB} \wedge \cdots \wedge B_{c_l}\sigma\theta_e \in \text{DB} \tag{1}$$

where θ_e is the most general substitution such that $A\theta_e = e$. However, since the background theory DB is of size n_b and all predicates are of arity a or less, there are at most an_b constants in DB, and hence only $(an_b)^k$ possible substitutions $\sigma_1, \ldots, \sigma_{(an_b)^k}$ to the k free variables.

Thus, let us introduce the boolean variables v_{ij} where i ranges from one to $n_b \cdot (n_e+k)^a$ and represents a literal, and j ranges from one to $(an_b)^k$ and represents a substitution. Notice that the size of this set of variables is polynomial in n_e and n_e. We will define the instance mapping $f_i(e)$ of an example e to return an assignment η_e to these variables as follows: v_{ij} will be true in η_e if and only if $B_i\sigma_j\theta_e \in$ DB, where θ_e is as defined above. Finally, let the concept mapping $f_c(C)$ map a clause $C = A \leftarrow B_{c_1} \wedge \cdots \wedge B_{c_l}$ to the DNF formula

$$f_c(C) \equiv \bigvee_{j=1}^{(an_b)^k} \bigwedge_{i=1}^{l} v_{c_i j}.$$

Since both $(an_b)^k$ and l are polynomial (in n_e, n_b, and n_t) the formula $f_c(C)$ is of polynomial size. It also can be verified that $f_c(C)$ is true exactly when Eq. (1) is true, and hence this mapping preserves concept membership. This completes the reduction and the proof. □

The predictability of DNF has been an open problem in computational learning theory for several years. Thus, while this result does not actually settle the question of whether

[7] More precisely, for every target clause C in which the variables in the head are are not distinct, there is an equivalent clause C' in which the variables in the head of C' are distinct, and the necessary equality constraints are represented by conditions in the body of C'. It is easy to see that C' need be only polynomially larger than C'.

indeterminate clauses are predictable, it does show that answering the question will require a substantial theoretical advance.

4.3. Clauses with bounded indeterminacy

If one believes that DNF is hard to predict, then the result above is negative; however, it does suggest some possible restrictions that might lead to learnable languages. The first restriction suggested by this result is based on the observation that the "degree of indeterminacy" of a clause is closely related to the number of terms in the DNF formula that is needed to emulate it, and that k-term DNF is predictable for any fixed k. Hence, it may be that bounding the number of possible substitutions associated with a clause will lead to a predictable language. Such a result would be useful: intuitively, this would show that predictability (if not learnability) decreases gradually as indeterminacy is introduced to a language.

In this section, we will investigate such a restriction. It turns out that this intuition is correct: in particular, the result of Theorem 6 can be extended to a certain language of clauses with bounded indeterminacy. This gives us a positive learnability result in the weaker model of predictability. We will first present a fairly general version of this result, and then consider some concrete instantiations of the general result.

4.3.1. Bounding the indeterminacy of a clause

We will want to talk about clauses that are almost, but not quite, deterministic; hence the following definition.

Definition 15 (*Effectively k-indeterminate*). A language LANG[DB] is called *effectively k-indeterminate* (with respect to X) iff there is a polytime computable procedure $SUBST(e, \mathrm{DB})$ that, given any $e \in X$, computes a set of substitutions $\{\theta_1, \ldots, \theta_l\}$ having the following properties:

- the number of substitutions l is bounded above by k,
- for every $C \in \mathrm{LANG}[\mathrm{DB}]$, if e is in the extension of C, then every most general substitution θ' that proves e to be in the extension of C is included in the set of θ_i's generated by $SUBST$.

Note that since duplications are allowed among the θ_i (i.e., it might be that $\theta_i = \theta_j$ for some $i \neq j$) we can assume without loss of generality that $l = k$.

Informally, a language is k-indeterminate if given an instance e, one can produce a small set of candidate substitutions that suffice for all the theorem proving that might be necessary. As one example of such a language, ij-determinate clauses are effectively 1-indeterminate: here, $SUBST$ can be implemented by using a Prolog style theorem prover to generate the single substitution that proves e to be in the extension of C. Some additional examples are given in Section 4.3.2.

The following property will also be important:

Definition 16 (*Polynomial literal support*). A language family LANG[$\mathcal{DB}$] has *polynomial literal support* iff for every X_{n_v} and every $\mathrm{DB} \in \mathcal{DB}$ there is a set of literals LIT

and a *partial order* $\prec$ on *LIT* such that

- the cardinality of *LIT* is polynomial in n_e and $\|\mathrm{DB}\|$;
- LANG[DB] is exactly those clauses $A \leftarrow B_1 \wedge \cdots \wedge B_r$, where A is fixed, all the B_i are members of *LIT*, and the body of the clause satisfies the following restriction: if $B_i \prec B_j$ and B_j is in the body of the clause, then B_i also is in the body of the clause and appears to the left of B_j.

One example of a language with polynomial literal support is the language of ij-determinate clauses: in this case, the polynomial bound on the number of literals in a clause can be obtained by a simple counting argument [37], and the ordering function is the relationship

$$B_i \prec B_j \quad \text{iff} \quad \text{the input variables of } B_j \text{ are bound by } B_i.$$

This definition in fact generalizes a key property that, together with determinism, makes ij-determinate clauses pac-learnable. The language of k-free clauses also has polynomial literal support; in this case, the ordering function might be a constant function, or might be used to ensure that clauses are "linked" in such a way as to reduce indeterminacy. The example of k-free clauses shows that polynomial literal support is not sufficient to ensure learnability.

The principle result of this section is the theorem below, which shows that imposing these two restrictions yields a predictable language of clauses. Unfortunately, it is difficult to extend this predictability result to a pac-learning result. This issue is discussed further in Section 4.3.3.

Theorem 17. *Let* k-INDETERMPLS *be any clause language with polynomial literal support that is also effectively* k*-indeterminate. Then for any fixed* a*, the language family* k-INDETERMPLS$[a\text{-}\mathcal{DB}]$ *is uniformly predictable.*

Proof. The proof is analogous to the proof of Theorem 14, except that we will reduce learning a clause in k-INDETERMPLS to learning a k-term DNF expression: i.e., we will show that for any $\mathrm{DB} \in a\text{-}\mathcal{DB}$

$$k\text{-INDETERMPLS}[a\text{-}\mathcal{DB}] \trianglelefteq k\text{-TERMDNF}.$$

The theorem follows immediately from this, since k-term DNF is predictable using k-CNF as a hypothesis space [42].

Since the language k-INDETERMPLS has polynomial literal support, there is some set of literals $B_1, \ldots, B_n$ such that each clause C in the language can be written $C = A \leftarrow B_{c_1} \wedge \cdots \wedge B_{c_l}$. As before we will introduce a set of variables $v_{c_i j}$ where c_i ranges from 1 to n and encodes a literal B_{c_i}, and j ranges from 1 to k and encodes a substitution.

The instance mapping will map an example e to an assignment η_e over these kn variables as follows. First, the procedure $SUBST(e, \mathrm{DB})$ guaranteed by the definition of effective k-indeterminacy is used to generate a set of k substitutions $\theta_1, \ldots, \theta_k$. The ordering of these substitutions can be arbitrary (we will see why shortly). An assignment η_e is then constructed in which $v_{c_i j}$ is true if and only if $B_i\theta_j \in \mathrm{DB}$ for θ_j.

Finally, define the concept mapping f_c to map the clause $C = A \leftarrow B_{c_1} \wedge \cdots \wedge B_{c_l}$ to the k-term DNF formula

$$f_c(C) \equiv \bigvee_{j=1}^{k} (v_{c_1,j} \wedge \cdots \wedge v_{c_l,j}).$$

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 also that the ordering of the θ_i is irrelevant, and can even be different for different examples. □

4.3.2. Languages satisfying the restrictions

Although the result above is stated quite generally, it is nonetheless rather difficult to devise natural syntactic restrictions that enforce these two key restrictions: that clauses have polynomial support, and that they be effectively k-indeterminate. One possible language is suggested by the proof of Theorem 14, which shows that any language k-FREE[DB] is effectively $\|\mathrm{DB}\|^k$-indeterminate. Thus the language family of k-free clauses over databases of constant size l is predictable. Thus letting $\mathcal{DB}_l$ denote the set of databases of size less than or equal to l:

Observation 18. *For fixed k and l the language family k-FREE[$\mathcal{DB}_l$] is uniformly predictable.*

Note however that the time complexity of the most natural prediction algorithm (where one predicts k-term DNF using k-CNF) is $O({n_e}^{lk})$, which seems rather high for a practical algorithm. Also, restricting the size of the background database is a rather severe restriction.

Another possibly more useful way of defining a language meeting the restrictions above is as follows.

- First, specify a tuple of n_e variables. The head of every clause in the language will have as its arguments the tuple T.
  For instance, in learning family relationships like *grandfather* or *nephew*, one might fix the arguments to be the two variables X and Y, in that order.
- Next, specify some small set of output literals $S = \{L_1, \ldots, L_c\}$ and an ordering function $\prec_S$ such that each output literal can have at most d possible bindings, when used in a clause in a manner consistent with $\prec_S$.
  Continuing the example given above (in which X and Y are the arguments to the head of the clause), one might specify the following set of $c = 6$ output literals

$$\begin{aligned} S = \{ \; & L_1 = parent(X, A), L_2 = parent(Y, B), \\ & L_3 = parent(A, C), L_4 = parent(B, D), \\ & L_5 = spouse(X, E), L_6 = spouse(Y, F) \} \end{aligned}$$

  together with the following ordering $\prec_S$:

$L_1 \prec_S L_3$,

$L_2 \prec_S L_4$.

Given this ordering constraint, literals L_1, L_2, L_3 and L_4 can have at most two bindings (assuming that a person has at most two parents) and literals L_5 and L_6 can have at most one binding (assuming each person has only one spouse). Thus, for this set S, we have $d = 2$.

- Finally, define the language S-OUTPUT to be the set of clauses that have heads with the argument list T and bodies that contain output literals selected from the set S, and used in an order consistent with $\prec_S$.

It is easy to show that a clause in S-OUTPUT is effectively $(|S| \cdot d)$-indeterminate: the procedure for generating substitutions is simply to backtrack to generate all possible substitutions for the free variables in the literal set S. Also, for databases with a fixed arity a the language S-OUTPUT has polynomial literal support, since it has at most $a|S|$ free variables. Hence:

Observation 19. *For every constants a, c, and d, and every literal set S, S-OUTPUT[a-$\mathcal{DB}$] is uniformly predictable, provided that $|S| < c$, and every literal in S can have at most d bindings.*

The time complexity of the k-CNF-based prediction algorithm is $O(n_e{}^{cd})$.

It should be noted that there is no a priori way to choose the literal set S and ordering function $\prec_S$. Thus in practice, specifying a language S-OUTPUT requires additional user input. For example, in the family relationship learning problem given above, the user had to specify (in addition to the examples and the background database DB)

- the pair of variables X, Y that must appear in the head of the hypothesis clause;
- the set of indeterminate literals $S = \{parent(A, X), \ldots\}$ that can appear in a hypothesis clause;
- the ordering function $\prec_S$.

In this respect the clause language S-OUTPUT differs from i-DEPTHDETERM and k-FREE, which require little user input to specify.

This result also can be generalized somewhat. One generalization is based on the fact that ij-determinate clauses also have polynomial literal support. It is thus possible to combine the language of ij-determinate clauses with the language S-OUTPUT to obtain a new predictable language, of clauses of the form

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

where $A \leftarrow B_1 \wedge \cdots \wedge B_r$ is ij-determinate and $A \leftarrow D_1 \wedge \cdots \wedge D_s$ is S-OUTPUT. This result provides one way of introducing a small amount of non-determinism into the language of ij-determinate clauses without making prediction intractable.

4.3.3. Further discussion

It should be emphasized that although this is a positive result, there are a number of reasons why the result is rather weak. First, we have not shown the language to be

pac-learnable, only to be predictable; thus there is no way of obtaining a clause that accurately approximates the target clause. This is a disadvantage if the ultimate goal is to integrate the result of learning with a reasoning system based on logic programs. Furthermore, the result appears to be difficult to extend to pac-learnability, for two reasons: first, because k-term DNF is hard to pac-learn, and second, because the concept mapping used to reduce clause learning to k-term DNF cannot be easily reversed. The latter fact means that even if the prediction algorithm used for k-term DNF yields hypotheses that can be easily converted to logic programs (see, for example, [3], which describes an algorithm that learns k-term DNF with general DNF) or even for classes of distributions under which k-term DNF is directly learnable (see, for example, [33]) it may still be impossible to pac-learn clauses from an effectively k-indeterminate language with polynomial literal support.

A second problem is that all known algorithms for predicting k-term DNF require time exponential in k. This suggests that only a small amount of indeterminism can be tolerated without imposing additional restrictions. For these reasons, we will consider in the next section a different restriction on indeterminate clauses.

5. Learnable indeterminate clauses

5.1. Highly local clauses are learnable

We will now consider an alternative restriction on indeterminate clauses, the aim being to find a language of indeterminate clauses that is not only predictable, but also pac-learnable.

The construction in Lemma 12 requires a free variable that appears in every literal; a natural question to ask is if limiting the number of occurrences of each free variable makes indeterminate clauses easier to learn. This restriction, unfortunately, does not help in general;[8] however a closely related restriction does make learning easier. The basic idea behind the restriction is to limit the length of a "chain" of "linked" variables; we develop this notion more formally below.

Definition 20 (*Locale*). Let V_1 and V_2 be two free variables appearing in a clause $A \leftarrow B_1 \wedge \cdots \wedge B_r$. 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_3 that influences V_2. The *locale* of a variable V is the set of literals $\{B_{i_1}, \ldots, B_{i_l}\}$ that contain either V, or some variable influenced by V.

Thus *influences* and *touches* are both symmetric and reflexive relations, and *influences* is the transitive closure of *touches*. Informally, variable V_1 influences variable V_2 if the choice of a binding for V_1 can affect the possible choices of bindings for V_2 (when testing to see if a ground fact e is in the extension of C). The locality of a clause is the

[8] The problem is that if a database contains an equality predicate, then variables can be "copied" an arbitrary number of times.

size of the largest set of literals influenced by a free variable. The following examples illustrate locality.

Example. In the following clauses, the free variables are highlighted, and the locale of each free variable is underlined.

$$\text{father}(F, S) \leftarrow \text{son}(S, F) \wedge \underline{\text{husband}(F, \mathbf{W})}.$$

$$\text{no\_payment\_due}(S) \leftarrow \underline{\text{enlist}(S, \mathbf{PC}) \wedge \text{peace\_corps}(\mathbf{PC})}.$$

$$\text{draftable}(S) \leftarrow \underline{\text{citizen}(S, \mathbf{C}) \wedge \text{united\_states}(\mathbf{C}) \wedge \text{age}(S, \mathbf{A}) \wedge (\mathbf{A} \geqslant 18) \wedge (\mathbf{A} \leqslant 26)}.$$

Notice that the influence relation applies only to free variables; thus in the third clause above, the variable S is *not* influenced by C, and hence $age(S, A)$ is not in the locale of C.

Finally, let the *locality* of a clause be the cardinality of the largest locale of any free variable in that clause, and let k-LOCAL denote the language of clauses with locality k or less. The principle result of this section is the following.

Theorem 21. *For any fixed k and a, the language family k-LOCAL$[a\text{-}\mathcal{DB}]$ is uniformly pac-learnable.*

Proof. Let S^+, S^- be a sample labeled by the k-local clause $A \leftarrow B_1 \wedge \cdots \wedge B_l$. As in the proof of Theorem 17, one can assume that predicate symbol and arity of A are known, and that the arguments to A are n_e distinct variables. As every new literal in the body can introduce at most a new variables, any size k locale can contain at most $n_e + ak$ distinct variables. Also note that there are at most n_b distinct predicates in the database DB. Since each literal in a locality has one predicate symbol and at most a arguments, each of which is one of the $n_e + ak$ variables, there are only $n_b(n_e + ak)^a$ different literals that could appear in a locality, and hence at most $p = (n_b(n_e + ak)^a)^k$ different[9] localities of length k. Let us denote these localities as $LOC_1, \ldots, LOC_p$. Note that for constant a and k, the number of distinct localities p is polynomial in n_e and n_b.

Now, notice that every clause C of locality k can be written in the form

$$A \leftarrow LOC_{i_1}, \ldots, LOC_{i_r}$$

where each LOC_{i_j} is one of the p possible locales, and no free variable appears in more than one of the LOC_{i_j}. Since no free variables are shared between locales, the different locales do not interact, and hence $e \in ext(C, \text{DB})$ exactly when $e \in ext(A \leftarrow LOC_{i_1}, \text{DB}), \ldots, e \in ext(A \leftarrow LOC_{i_r}, \text{DB})$. In other words, C can be decomposed into a conjunction of components of the form $A \leftarrow LOC_{i_j}$. One can thus use Valiant's [50] technique for monomials to learn C.

[9] Up to renaming of variables.

In a bit more detail, the following algorithm will pac-learn k-local clauses. The learner initially hypothesizes the most specific k-local clause, namely

$$A \leftarrow LOC_1, \ldots, LOC_p.$$

The learner then examines each positive example e in turn, and deletes from its hypothesis all LOC_i such that $e \notin ext(A \leftarrow LOC_i, \mathrm{DB})$. (Note that e is in this extension exactly when $\exists\sigma$: $\mathrm{DB} \vdash LOC_i\theta_e\sigma$ where θ_e is the most general substitution such that $A\theta_e = e$. To see that this condition can be checked in polynomial time, recall that σ can contain at most ak free variables, and DB can contain at most an_b constants; hence at most $(an_b)^{ak}$ substitutions σ need be checked, which is polynomial.) Following the argument used for Valiant's procedure, this algorithm will pac-learn the target concept. □

Again, this result can be extended somewhat; for example, there is pac-learning algorithm for the language of clauses of the form

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

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

5.2. *The expressive power of local clauses*

Theorem 21 is a positive result; it shows that k-local clauses can be efficiently learned in a reasonable formal model. The importance of this result, however, depends a great deal on the usefulness of k-local clauses as a representation language; we note that k-local clauses, unlike ij-determinate clauses, do not seem to correspond very well to the sorts of clauses typically used in logic programs for list manipulation and other programming tasks. In this section, we will attempt to evaluate the usefulness of locality as a bias.

5.2.1. *Experimental results*

One way to evaluate a bias is empirically, by applying a learning system that uses that bias to benchmark problems. Some preliminary experiments of this sort are reported elsewhere [11]. In these experiments, several different versions of the experimental ILP system Grendel were constructed, each of which learned programs made up of clauses from a different clause language. Among the clause languages considered were ij-determinate and k-local clauses. These different versions of Grendel were then compared on a set of eight benchmark problems taken from the literature. These experiments confirmed that ij-determinacy is useful on many problems, notably in learning simple recursive programs like *append* and *list*. However, on two of the eight benchmarks, significantly better results were obtained by discarding the determinacy restriction and imposing instead a locality restriction. Thus, the results suggest that it is sometimes important to relax the determinacy restriction, and indicate that locality is, at least in some cases, a useful way of doing so.

5.2.2. Locality generalizes ij-determinacy

A second way to evaluate the usefulness of locality is to formally analyze the expressive power of k-local clauses. The easiest way to do this is by comparing k-LOCAL to other languages. For instance, any clause with locality k clearly must have depth $k+1$ or less; thus k-LOCAL is also a restriction of the language of clauses of constant depth. However, the language k-LOCAL is incomparable to the language of clauses with a bounded number of free variables. To see this, note that the construction used in Lemma 12 is a length-n clause with a single free variable that has locality n, while similarly the clause

$$p(X) \leftarrow q_1(X, Y_1) \wedge \cdots \wedge q_n(X, Y_n)$$

has locality one, but n free variables. To summarize, k-LOCAL $\subseteq (k+1)$-DEPTH, but for all k', k-LOCAL $\not\subseteq k'$-FREE and k-FREE $\not\subseteq k'$-LOCAL.

A more interesting question is the relationship of k-local clauses to ij-determinate clauses. Clearly, since k-local clauses can include indeterminate literals, some k-local clauses are not ij-determinate. It is also the case that determinate clauses with bounded depth can have unbounded locality. As an example, consider the clause

$$p(X) \leftarrow \text{successor}(X, Y) \wedge q_1(Y) \wedge \cdots \wedge q_n(Y).$$

However, there is a surprising relationship between the two languages: it turns out that every ij-determinate clause can be *rewritten* as a clause with bounded locality, where the bound on the locality is a function only of i and j. Thus, in a very reasonable sense, the language of clauses of constant locality is a strict generalization of the language of determinate clauses of constant depth.

More precisely, the following relationship holds between these languages.

Theorem 22. *For every* DB $\in$ *a-$\mathcal{DB}$, every d, and every clause $C \in$ d-*DEPTHDETERM[DB], *there is clause $C' \in k$-*LOCAL[DB] *such that C' is equivalent to C and $\|C'\| \leqslant k\|C\|$, where $k = a^{d+1}$.*

Proof. Let $C = A \leftarrow B_1 \wedge \cdots \wedge B_r$ be a clause. We will say that literal B_i *directly supports* literal B_j iff some output variable of B_i is an input variable of B_j, and that literal B_i *indirectly supports* B_j iff B_i directly supports B_j, or if B_i directly supports some B_k that indirectly supports B_j. (Thus "indirectly supports" is the transitive closure of "directly supports".)

Now, for each B_i in the body of C, let LOC_i be the conjunction

$$LOC_i = B_{j_1} \wedge \cdots \wedge B_{j_{k_i}} \wedge B_i$$

where the B_j are all of the literals of C that support B_i, either directly or indirectly, appearing in the same order that they appeared in C. Next, let us introduce for $i = 1, \ldots, r$ a substitution

$$\sigma_i = \{Y = Y_i : Y \text{ is a variable occurring in } LOC_i \text{ but not in } A\}.$$

The determinate clause C: Below is a function-free version of a depth-2 determinate clause learned on an actual problem [30].

more_active(DrugA,DrugB) ←
structure(DrugA,X,Y,Z)∧ % B_1
not_equal_to_h(X)∧ % B_2
polarity(Y,P)∧ % B_3
equal_to_2(P)∧ % B_4
structure(DrugB,T,U,V)∧ % B_5
equal_to_h(V). % B_6

The support relationships: B_1 is not directly supported by any literals; B_2 is directly supported by B_1; B_3 is directly supported by B_1; B_4 is directly supported by B_3, and indirectly supported by B_1; B_5 is not directly supported by any literals; and B_6 is directly supported by B_5.

The first phase of the construction:

LOC_1 = structure(DrugA,X,Y,Z)
LOC_2 = structure(DrugA,X,Y,Z)∧not_equal_to_h(X)
LOC_3 = structure(DrugA,X,Y,Z)∧polarity(Y,P)
LOC_4 = structure(DrugA,X,Y,Z)∧polarity(Y,P)∧equal_to_2(P)
LOC_5 = structure(DrugB,T,U,V)
LOC_6 = structure(DrugB,T,U,V)∧equal_to_h(V)

The constructed clause C': After renaming the variables in these conjunctions so that all free variables appear in only a single conjunction and collecting them into a single clause, we obtain the following clause C'.

more_active(DrugA,DrugB) ←
structure(DrugA,X_1,Y_1,Z_1) ∧
structure(DrugA,X_2,Y_2,Z_2)∧not_equal_to_h(X_2) ∧
structure(DrugA,X_3,Y_3,Z_3)∧polarity(Y_3,P_3) ∧
structure(DrugA,X_4,Y_4,Z_4)∧polarity(Y_4,P_4)∧equal_to_2(P_4) ∧
structure(DrugB,T_5,U_5,V_5) ∧
structure(DrugB,T_6,U_6,V_6)∧equal_to_h(V_6).

Fig. 3. Constructing a local clause equivalent to a determinate clause.

We can then define $LOC'_i = LOC_i\sigma_i$; the effect of this last step is that $LOC'_1, \ldots, LOC'_r$ are copies of $LOC_1, \ldots, LOC_r$ in which variables have been renamed so that the free variables of LOC'_{i_1} are different from the free variables of LOC'_{i_2}. Finally, let C' be the clause

$$A \leftarrow LOC'_1 \wedge \cdots \wedge LOC'_r.$$

An example of this construction is given in Fig. 3. We suggest that the reader refer to the example at this point.

We claim that C' is k-local, for $k = a^{d+1}$, that C' is at most k times the size of C, and furthermore that if C is determinate, then C' has the same extension as C. In the remainder of the proof, we will establish these claims.

To establish the first two claims (that C' is k-local and at most k times the size of C for $k = a^{d+1}$) it is sufficient to show that the number of literals in every LOC'_i (or equivalently, every LOC_i) is bounded by k. To establish this, let us define $N(d)$ to be the maximum number of literals in any LOC_i corresponding to a B_i with *input* variables at depth d or less. Clearly for any DB $\in a\text{-}\mathcal{DB}$ and $C \in d$-DEPTHDETERM[DB], the function $N(d)$ is an upper bound on k.

The function $N(d)$ is bounded by the following lemma.

Lemma 23. *For any* DB $\in a\text{-}\mathcal{DB}$, $N(d) \leqslant \sum_{i=0}^{d} a^i (\leqslant a^{d+1})$.

Proof. By induction on d. For $d = 0$, no literals will support B_i, and hence each locality LOC_i will contain only the literal B_i, and $N(0) = 1$.

Now assume that the lemma holds for $d - 1$ and consider a literal B_i with inputs at depth d. Notice that LOC_i can be no larger than the conjunction

$$\left(\bigwedge_{j:B_j \text{ directly supports } B_i} LOC_j \right) \wedge B_i.$$

Also, any literal B_j that directly supports B_i must be at depth $d - 1$ or less, and since there are no more than a input variables of B_i, there are at most a different B_j that directly support B_i. Putting this together, and using the inductive hypothesis that $N(d-1) \leqslant \sum_{i=1}^{d-1} a^i$, we see that

$$N(d) \leqslant aN(d-1) + 1 \leqslant a\left(\sum_{i=0}^{d-1} a^i\right) + 1 = \sum_{i=0}^{d} a^i.$$

By induction, the lemma holds. □

Now we consider the second claim that for any determinate C, the C' constructed above has the same extension. The first direction of this equivalence actually holds for any clause C:

Lemma 24. *If a fact f is in the extension of C with respect to* DB, *then f is in the extension of C' with respect to* DB.

Proof. We wish to show that if $f \in ext(C, \text{DB})$, then $f \in ext(C', \text{DB})$. Since duplicating literals in the body of a clause does not change its extension, it is sufficient to show that if $f \in ext(\hat{C}, \text{DB})$, then $f \in ext(C', \text{DB})$, where

$$\hat{C} = (A \leftarrow LOC_1 \wedge \cdots \wedge LOC_r).$$

Consider the substitutions σ_i introduced in the construction of C'. Since each free variable in LOC_i is given a distinct name in LOC'_i, σ_i is a one-to-one mapping, and since the free variables in the LOC'_i are distinct, the substitution $\sigma = \bigcup_{i=1}^{r} \sigma_i^{-1}$ is well defined. As an example, for the clause C' from Fig. 3, we would have

$$\sigma = \{X_1 = X, X_2 = X, X_3 = X, X_4 = X,$$
$$Y_1 = Y, Y_2 = X, Y_3 = Y, Y_4 = Y,$$
$$Z_1 = Z, Z_2 = Z, Z_3 = Z, Z_4 = Z,$$
$$P_3 = P, P_4 = P, T_5 = T, T_6 = T,$$
$$U_5 = U, U_6 = U, V_5 = V, V_6 = V\}.$$

It is easy to see that applying this substitution to C' will simply "undo" the effect of renaming the variables—i.e. that $C'\sigma = \hat{C}$.

Now, assume $f \in ext(\hat{C}, \mathrm{DB})$; then there is by definition some substitution θ so that all literals in the body of the clause $\hat{C}\theta$ are in DB. Clearly for the substitution $\theta' = \sigma \circ \theta$ all literals in the body of the clause $C'\theta'$ are in DB, and hence $f \in ext(C', \mathrm{DB})$. □

We must finally establish the converse of Lemma 24. This direction of the equivalence requires that C be determinate.

Lemma 25. *If a fact f is in the extension of C' with respect to* DB, *and C is determinate, then f is in the extension of C with respect to* DB.

Proof. If $f \in ext(C', \mathrm{DB})$, then there must be some θ' that proves this. Let us define a variable Y_i in C' to be a "copy" of $Y \in C$ if Y_i is a renaming of Y (i.e., if $Y_i\sigma_i^{-1} = Y$ for the σ_i defined above). Certainly if C is determinate then C' is determinate, and it is easy to show that for a determinate C', θ' must map every copy of Y to the same constant t_Y. (This is most easily proved by picking two copies Y_i and Y_j of Y and then using induction over the depth of Y to show that they must be bound to the same constant. For variables of depth $d = 0$, the statement is vacuously true, since there are no copies of depth-0 variables. For variables of depth $d > 0$, consider the literals B_i and B_j that contain Y_i and Y_j as output variables, and apply the inductive hypothesis to show that their input variables must have the same bindings; together with the determinism of C, this shows that Y_i and Y_j must be bound to the same constant.)

Hence, let us define the substitution

$$\theta = \{Y = t_Y : \text{copies of } Y \text{ in } C' \text{ are bound to } t_Y \text{ by } \theta'\}.$$

Clearly, for all i: $1 \leqslant i \leqslant r$, $LOC_i\theta = LOC_i\theta'$; hence if θ' proves that $f \in ext(C', \mathrm{DB})$ then θ proves that $f \in ext(C, \mathrm{DB})$. □

Proof of Theorem 22 (*continued*). We have now established that C' is k-local, of bounded size, and is equivalent to C. This concludes the proof of the theorem. □

We note that the proof technique used in Theorem 22 is similar to that used by Džeroski, Muggleton and Russell [16] to show that ij-determinate clauses are learnable; in particular, Džeroski, Muggleton and Russell showed that a ij-determinate clause can be rewritten as a conjunction of boolean propositions, each of which corresponds closely to the conjunctions $LOC'_1, \ldots, LOC'_r$ introduced in the proof above.

Finally, although we have shown that ij-determinate clauses have locality bounded by a constant, it should be observed that the constant is fairly large: for example,

for $i = j = 3$, the bound on locality would be $k = 3^4 = 81$. Hence the algorithm of Theorem 21 need not be the best algorithm for learning ij-determinate clauses.

6. Extensions to multiple-clause programs

So far, all of our results have been for programs containing a single clause. We will now consider extending the results presented above to programs that contain more than one clause. This is an important topic, because many practical systems learn programs containing multiple clauses; further understanding of the limitations of such systems would clearly be useful.

Still considering non-recursive programs over a fixed database, an immediate result is that even with severe restrictions, learning an arbitrary logic program is cryptographically hard. (We will use, in this proof, the usual semantics for logic programs [34].)

Theorem 26. *For $a \geqslant 1$, the language of non-recursive multiple-clause programs containing clauses from the following language families are not polynomially predictable, under cryptographic assumptions:*

- 0-DEPTHDETERM$[a\text{-}\mathcal{DB}]$,
- 0-FREE$[a\text{-}\mathcal{DB}]$,
- 0-LOCAL$[a\text{-}\mathcal{DB}]$.

This is true even if the background database is restricted to contain only a single fact.

Proof. The proof is a straightforward adaptation of the proof of Theorem 9; we will again reduce predicting a boolean circuit to an ILP learning problem (in this case, learning a multiple-clause program). We will assume that the circuit contains only AND and OR gates.[10] The instance mapping is as in Theorem 9. The concept mapping maps a circuit to a program P as follows.

- For every AND gate G_i the program P will contain a clause

$$p_i(X_1, \ldots, X_n) \leftarrow L_{i1} \wedge L_{i2}$$

where L_{i1} and L_{i2} are defined as follows:

$$L_{ij} \equiv \begin{cases} true(X_k), & \text{if the } j\text{th input to gate } G_i \text{ is the variable } x_k, \\ p_k(X_1, \ldots, X_n), & \\ & \text{if the } j\text{th input to gate } G_i \text{ is the output of gate } G_k. \end{cases}$$

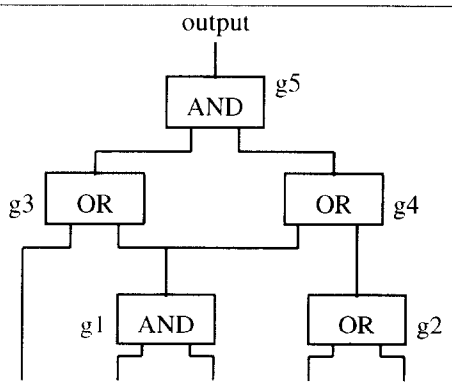
- For every OR gate G_i the program P will contain two clauses

$$p_i(X_1, \ldots, X_n) \leftarrow L_{i1},$$

$$p_i(X_1, \ldots, X_n) \leftarrow L_{i2},$$

where L_{i1} and L_{i2} are defined analogously.

[10] This is possible because of another reduction; by repeatedly applying DeMorgan's laws to a circuit we can force all NOT's to be negations of input variables, and by constructing an instance mapping that introduces n new variables equivalent to $\overline{x_1}, \ldots, \overline{x_n}$ we can also eliminate these NOT gates.



$circuit(X1,X2,X3,X4,X5) \leftarrow p_5(X1,X2,X3,X4,X5).$
$p_5(X1,X2,X3,X4,X5) \leftarrow$
$\quad p_3(X1,X2,X3,X4,X5) \wedge p_4(X1,X2,X3,X4,X5).$
$p_4(X1,X2,X3,X4,X5) \leftarrow p_1(X1,X2,X3,X4,X5).$
$p_4(X1,X2,X3,X4,X5) \leftarrow p_2(X1,X2,X3,X4,X5).$
$p_3(X1,X2,X3,X4,X5) \leftarrow true(X1).$
$p_3(X1,X2,X3,X4,X5) \leftarrow p_1(X1,X2,X3,X4,X5).$
$p_2(X1,X2,X3,X4,X5) \leftarrow true(X4).$
$p_2(X1,X2,X3,X4,X5) \leftarrow true(X5).$
$p_1(X1,X2,X3,X4,X5) \leftarrow true(X2) \wedge true(X3).$

Fig. 4. Constructing a logic program equivalent to a circuit.

- Finally, P contains a single clause

$$circuit(X_1,\ldots,X_n) \leftarrow p_n(X_1,\ldots,X_n)$$

where p_n is the gate whose output is the output of the circuit.

An example of this construction is shown in Fig. 4.

It is easy to verify that this construction reduces learning circuits to learning multiple-clause programs of any of the types named in the theorem over the database DB = $\{true(1)\}$. □

Since arbitrary logic programs are hard to learn, we will henceforth restrict ourselves to cases in which the heads of all the clauses in the program have the same predicate symbol and arity; this restriction has been made by a number of practical learning systems [6, 37, 40, 46]. We will call such programs *multiple-clause predicate definitions*. For this case, the semantics of our representation remains simple: a multiple-clause predicate definition is simply a set of clauses $P = \{C_1,\ldots,C_k\}$, and the extension of P with respect to a database DB is the union of the extensions of the C_i (again with respect to DB). Again, this coincides with the usual semantics for non-recursive programs.

Many of the preceding results extend immediately to multiple-clause predicate definitions; for completeness, we will state these extensions below.

Observation 27. *For $a \geqslant 3$, the language of multiple-clause predicate definitions containing clauses from* ($\log n_e$)-DEPTHDETERM[a-$\mathcal{DB}$] *are not polynomially predictable, under cryptographic assumptions.*

This follows directly from the non-predictability of single clauses.

Observation 28. *For any $a \geqslant 2$ and any $k \geqslant 1$, the language of multiple-clause predicate definitions containing clauses from* k-FREE[a-$\mathcal{DB}$] *is uniformly predictable if and only if* DNF *is predictable.*

This follows directly from Theorems 11 and 14, and from the fact that the disjunction of a set of DNF formula is still in DNF.

Observation 29. *Let* k-INDETERMPLS *be any clause language with polynomial literal support containing only effectively k-indeterminate clauses. Then for any fixed a, k and l, the language of multiple-clause predicate definitions containing at most l clauses from* k-INDETERMPLS[a-$\mathcal{DB}$] *is uniformly predictable.*

This follows directly from Theorem 17 and the fact that the union of l distinct k-term DNF formulas, where l is constant, is a $(k \cdot l)$-term DNF formula.

It remains to consider extensions of Theorems 6 and 21, which show the pac-learnability of ij-determinate clauses and k-local clauses respectively. Again, these extensions are straightforward, based on previous results. The proof of Theorem 6 is based on an invertible reduction to boolean monomials: any ij-determinate clause can be learned by constructing an appropriate set of boolean features, learning a monotone monomial over those features, and then converting this monomial back to a ij-determinate clause. While Theorem 21 was proved directly, it could have also been proved via an invertible reduction to monomials.[11] Since in both cases a single clause reduces to a monotone monomial, and it is known that in a distribution-independent setting, monotone DNF is as hard to learn as general DNF, one can easily obtain this result:

Observation 30. *For any fixed i and j, the language of multiple-clause predicate definitions containing clauses from* i-DEPTHDETERM[j-$\mathcal{DB}$] *is pac-learnable iff* DNF *is pac-learnable.*

For any fixed k and a, the language of multiple-clause predicate definitions containing clauses from k-LOCAL[a-$\mathcal{DB}$] *is pac-learnable iff* DNF *is pac-learnable.*

Some positive results are also obtainable. It is known that for any constant l, monotone l-term DNF is learnable against simple distributions [33].[12] Džeroski, Muggleton and Russell have observed that the learning algorithm for monotone l-term DNF against a simple distribution can be used to learn an l-clause ij-determinate predicate definition

[11] Specifically, one could construct p variables $v_1, \ldots, v_p$, and map an example e to an assignment η_e in which each v_i is true iff $e \in ext(A \leftarrow LOC_i, \mathrm{DB})$.

[12] Simple distributions are a broad class of probability distributions that include all computable distributions.

by first, constructing the appropriate set of boolean features, then learning an l-term DNF over those features, and finally converting this formula back to a ij-determinate predicate definition. Thus:

Theorem 31 (Džeroski, Muggleton and Russell [16]). *For any fixed i and j, the language of multiple-clause predicate definitions containing at most l clauses from*

$$i\text{-DEPTHDETERM}[j\text{-}\mathcal{DB}]$$

is uniformly pac-learnable against simple distributions.

The same proof technique can be applied to predicate definitions containing k-LOCAL clauses; thus we have the following corollary of Theorems 21 and 31.

Observation 32. *For any fixed k and a, the language of multiple-clause predicate definitions containing at most l clauses from k-LOCAL$[a\text{-}\mathcal{DB}]$ is uniformly pac-learnable against simple distributions.*

One problem with applying these results in practice is that the proofs of learnability for simple distributions are not completely constructive: in particular, the learning algorithm must sample against a certain "universal distribution", which is not computable [33]. Implemented systems have thus used heuristic methods to learn multiple clauses.

7. Concluding remarks

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 [44]. This model encourages analyzing the learnability of a language by characterizing its expressive power.

In this paper we have characterized several extensions of the language of determinate clauses of constant depth [16, 37]. These results will now be summarized.

First, via a reduction from log-depth circuits, we showed that a single log-depth determinate clause is not pac-learnable.

Next, we relaxed the condition of determinacy, and obtained a number of results. Since indeterminate clauses of constant depth can be shown to be hard to predict, we considered several restrictions of this language. We showed that a clause with k free variables is as hard to learn as DNF. We also showed that restricting the degree of indeterminacy of a clause leads to predictability (but not pac-learnability) for any clause language with "polynomial literal support".

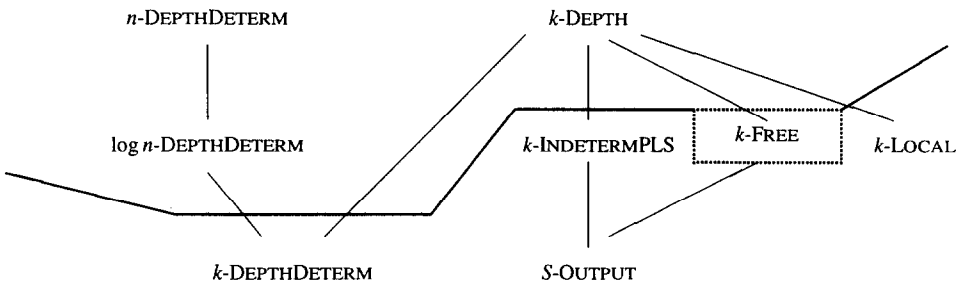


Fig. 5. Summary of results for single clauses. Above the heavy line are languages that are hard to predict, and below the heavy line are languages that are predictable. All predictable languages *except* k-INDETERMPLS are also pac-learnable. Predicting the boxed language k-FREE is equivalent to predicting DNF, an open problem.

Finally, we showed that restricting the locality of a clause to a constant k leads to pac-learnability. This result is especially interesting because k-local clauses can be shown to be a strict generalization of ij-determinate clauses in the following sense: for fixed i and j, every ij-determinate clause can be rewritten as a clause with locality no greater than j^{i+1}.

These results are summarized in Fig. 5, which shows the languages considered in this paper, partially ordered by their expressive power.[13] Some results are from previous work [16,27]; note however that the previous results for arbitrary-depth determinate and constant-depth indeterminate clauses make representational assumptions that we have relaxed. The analysis associated with the remaining languages is original. The languages below the heavy line are pac-learnable or (in the case of k-INDETERMPLS) polynomially predictable. The languages above the heavy line are hard to predict, under cryptographic assumptions, as are all supersets of these languages. The language k-free, shown boxed in a dotted line, is predictable iff DNF is predictable; this is an open question in computational learning theory.

In obtaining these results, several previous results from the literature have been extended. Haussler [23] raises the question of the learnability of existential conjunctive concepts with k variables in a representation-independent (i.e., predictability) setting. It is easy to show that every existential conjunctive concept can be expressed by a single indeterminate clause; thus an immediate result of Theorem 17 is that these concepts are in general as hard to predict as DNF.

More recently, Kietz [27] has shown that arbitrary-depth determinate clauses are hard to pac-learn, and that constant-depth indeterminate clauses are also hard to pac-learn. These results have both been strengthened in a number of ways in this paper: in particular, we have presented representation-*independent* hardness results for determinate

[13] Strictly speaking, k-INDETERMPLS is a set of languages, not a single language, and languages in the set need not be of depth k. However, every k-INDETERMPLS language that we have considered is of bounded depth.

clauses of only log depth, rather than arbitrary depth, and also for indeterminate constant-depth clauses.

We have also further investigated the learnability of various subclasses of indeterminate clauses. One result of this in-depth investigation has been isolation of an interesting subclass of indeterminate clauses (the class of k-local clauses) that is pac-learnable, and that is a strict generalization of the class of ij-determinate clauses.

Kietz and Džeroski [28] have also investigated the complexity of a closely related task called the *ILP problem*. The "ILP problem" for $(\vdash, \mathcal{DB}, \text{LANG}_E, \text{LANG}_H)$ is to find, given a background theory DB $\in \mathcal{DB}$ and a set of examples from LANG_E, a hypothesis in LANG_H that is "consistent" with the examples with respect to the provability relationship $\vdash$. One corollary of Theorem 21 is that the ILP problem for k-local clauses is tractable. The connection between our negative results and the ILP problem is somewhat more complex. As formalized by Kietz and Džeroski, it is possible to solve the ILP problem in polynomial time using an algorithm that generates a hypothesis that grows quickly with the number of examples—for example, if the hypothesis language LANG_H is sufficiently expressive, one might hypothesize a lookup table containing all the positive examples. Thus it is possible in principle that the ILP problem for a language might be solvable, even if the language is hard to predict. However, by the results of Blumer et al. [4] and Theorem 4, if a language LANG is not polynomially predictable and algorithm A solves the ILP problem for LANG, then either (i) A does not run in polynomial time, or (ii) the size of the hypotheses returned by A grows nearly linearly in the number of examples.[14] Thus the negative predictability results of this paper imply that the corresponding ILP problems cannot be tractably solved in any way that yields a concise hypothesis.

A number of further questions suggest themselves. The learnability of recursive logic programs is a challenging problem; some results in this area appear in [10, 13, 14, 19]. The learnability of multiple-clause predicate definitions is largely an open issue; although analysis is difficult, continued progress on the learnability of fairly general classes of DNF is encouraging [18, 22, 31]. The learnability of restricted classes of general logic programs, or general logic programs in more restricted settings (perhaps analogous to the settings considered by Angluin, Frazier and Pitt [2]) is also an open area. Finally, much work remains to be done in relating the learnable languages of first-order clauses to each other, as well as to other learnable first-order languages (e.g., [15, 41]).

Acknowledgments

The author would like to thank Haym Hirsh and Rob Schapire for comments on the presentation; Mike Kearns, Jörg-Uwe Kietz, and Rob Schapire for a number of helpful discussions; and the reviewers, for their many helpful comments on the presentation and technical content.

[14] More precisely, the hypothesis size for a sample of m examples is not always less than m^α for any $\alpha < 1$.

References

[1] A.V. Aho, J.E. Hopcroft and J.D. Ullman, *The Design and Analysis of Computer Algorithms* (Addison-Wesley, Reading, MA, 1974).

[2] D. Angluin, M. Frazier and L. Pitt, Learning conjunctions of horn clauses, *Mach. Learn.* **9** (2/3) (1992).

[3] A. Blum and M. Singh, Learning functions of k terms, in: *Proceedings Third Annual Workshop on Computational Learning Theory*, Rochester, NY (Morgan Kaufmann, Los Altos, CA, 1990).

[4] A. Blumer, A. Ehrenfeucht, D. Haussler and M. Warmuth, Classifying learnable concepts with the Vapnik–Chervonenkis dimension, *J. ACM* **36** (1989) 929–965.

[5] R. Boppana and M. Sipser, The complexity of finite functions, in: J. van Leeuwen, ed., *Handbook of Theoretical Computer Science* (North-Holland, Amsterdam, 1990) 758–804.

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

[7] W.W. Cohen, Compiling knowledge into an explicit bias, in: *Proceedings of the Ninth International Conference on Machine Learning*, Aberdeen (1992).

[8] W.W. Cohen, Cryptographic limitations on learning one-clause logic programs, in: *Proceedings Tenth National Conference on Artificial Intelligence*, Washington, DC (1993).

[9] W.W. Cohen, Learnability of restricted logic programs, in: *Proceedings Third International Workshop on Inductive Logic Programming*, Bled (1993).

[10] W.W. Cohen, A pac-learning algorithm for a restricted class of recursive logic programs, in: *Proceedings Tenth National Conference on Artificial Intelligence*, Washington, DC (1993).

[11] W.W. Cohen, Rapid prototyping of ILP systems using explicit bias, in: *Proceedings 1993 IJCAI Workshop on Inductive Logic Programming*, Chambery (1993).

[12] W.W. Cohen, Pac-learning nondeterminate clauses, in: *Proceedings Eleventh National Conference on Artificial Intelligence*, Seattle, WA (1994).

[13] W.W. Cohen, Pac-learning recursive logic programs: efficient algorithms, *J. Artif. Intell. Res.* **2**, 500–539.

[14] W.W. Cohen, Pac-learning recursive logic programs: negative results, *J. Artif. Intell. Res.* **2**, 541–573.

[15] W.W. Cohen and H. Hirsh, Learnability of description logics, in: *Proceedings Fourth Annual Workshop on Computational Learning Theory*, Pittsburgh, PA (ACM, New York, 1992).

[16] S. Džeroski, S. Muggleton and S. Russell, Pac-learnability of determinate logic programs, in: *Proceedings 1992 Workshop on Computational Learning Theory*, Pittsburgh, PA (1992).

[17] C. Feng, Inducing temporal fault diagnosic rules from a qualitative model, in: *Inductive Logic Programming* (Academic Press, New York, 1992).

[18] M. Flammini, A. Marchetti-Spaccamela and L. Kučera, Learning DNF formulae under classes of probability distributions, in: *Proceedings Fourth Annual Workshop on Computational Learning Theory*, Pittsburgh, PA (ACM, New York, 1992).

[19] M. Frazier and C.D. Page, Learnability of recursive, non-determinate theories: Some basic results and techniques, in: *Proceedings Third International Workshop on Inductive Logic Programming*, Bled (1993).

[20] M. Frazier and L. Pitt, Learning from entailment: An application to propositional horn sentences, in: *Proceedings Tenth International Conference on Machine Learning*, Amherst, MA (Morgan Kaufmann, Los Altos, CA, 1993).

[21] A. Frisch and C.D. Page, Learning constrained atoms, in: *Proceedings Eighth International Workshop on Machine Learning*, Ithaca, NY (Morgan Kaufmann, Los Altos, CA, 1991).

[22] T. Hancock, Learning $k\mu$ decision trees on the uniform distribution, in: *Proceedings Sixth Annual ACM Conference on Computational Learning Theory*, Santa Cruz, CA (ACM, New York, 1993).

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

[24] M. Kearns, M. Li, L. Pitt and L. Valiant, On the learnability of boolean formulae, in: *Proceedings 19th ACM Symposium on Theory of Computing*, New York (1987).

[25] M. Kearns and L. Valiant, Cryptographic limitations on learning Boolean formulae and finite automata, in: *Proceedings 21th ACM Symposium on Theory of Computing*, New York (1989).

[26] M. Kharitonov, Cryptographic lower bounds on the learnability of boolean functions on the uniform distribution, in: *Proceedings Fourth Annual Workshop on Computational Learning Theory*, Pittsburgh, PA (ACM, New York, 1992).
[27] J.-U. Kietz, Some computational lower bounds for the computational complexity of inductive logic programming, in: *Proceedings 1993 European Conference on Machine Learning*, Vienna (1993).
[28] J.-U. Kietz and Džeroski, Inductive logic programming and learnability, *SIGART Bull.* **5** (1994) 22–31.
[29] J.-U. Kietz and K. Morik, Constructive induction of background knowledge, in: *Proceedings Workshop on Evaluating and Changing Representation in Machine Learning (at IJCAI-91)*, Sydney (1991).
[30] 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, *Proc. Nat. Acad. Sci.* **89** (1992).
[31] E. Kushilevitz and D. Roth, On learning visual concepts and DNF formulae, in: *Proceedings Sixth Annual ACM Conference on Computational Learning Theory*, Santa Cruz, CA (ACM, New York, 1993).
[32] N. Lavrač and S. Džeroski, Background knowledge and declarative bias in inductive concept learning, in: K.P. Jantke, ed., *Analogical and Inductive Inference: International Workshop AII'92*, Lecture Notes in Artificial Intelligence **642** (Springer, Berlin, 1992).
[33] M. Li and P. Vitanyi, Learning simple concepts under simple distributions, *SIAM J. Comput.* **20** (5) (1991).
[34] J.W. Lloyd, *Foundations of Logic Programming* (Springer, Berlin, 2nd ed., 1987).
[35] B.D.S. Muggleton, The application of inductive logic programming to finite-element mesh design, in: *Inductive Logic Programming* (Academic Press, New York, 1992).
[36] S. Muggleton, Inductive logic programming, in: *Inductive Logic Programming* (Academic Press, New York, 1992).
[37] S. Muggleton and C. Feng, Efficient induction of logic programs, in: *Inductive Logic Programming* (Academic Press, New York, 1992).
[38] S. Muggleton, R.D. King and M.J.E. Sternberg, Protein secondary structure prediction using logic-based machine learning, *Protein Engrg.* **5** (1992) 647–657.
[39] S.H. Muggleton, ed., *Inductive Logic Programming* (Academic Press, New York, 1992).
[40] M. Pazzani and D. Kibler, The utility of knowledge in inductive learning, *Mach. Learn.* **9** (1) (1992).
[41] L. Pitt and M. Frazier, Classic learning, in: *Proceedings Seventh Annual ACM Conference on Computational Learning Theory*, New Brunswick, NJ (ACM, New York, 1994).
[42] L. Pitt and L. Valiant, Computational limitations on learning from examples, *J. ACM* **35** (1988) 965–984.
[43] L. Pitt and M.K. Warmuth, Reductions among prediction problems: On the difficulty of predicting automata, in: *Proceedings 3rd Annual IEEE Conference on Structure in Complexity Theory*, Washington, DC (IEEE Computer Society Press, Silver Spring, MD, 1988).
[44] L. Pitt and M. Warmuth, Prediction-preserving reducibility, *J. Comput. Syst. Sci.* **41** (1990) 430–467.
[45] G.D. Plotkin, A note on inductive generalization, *Mach. Intell.* **5** (1969) 153–163.
[46] J.R. Quinlan, Learning logical definitions from relations, *Mach. Learn.* **5** (3) (1990).
[47] J.R. Quinlan, Determinate literals in inductive logic programming, in: *Proceedings Eighth International Workshop on Machine Learning*, Ithaca, NY (Morgan Kaufmann, Los Altos, CA, 1991).
[48] R.E. Schapire, The strength of weak learnability, *Mach. Learn.* **5** (2) (1990).
[49] E. Shapiro, *Algorithmic Program Debugging* (MIT Press, Cambridge, MA, 1982).
[50] L.G. Valiant, A theory of the learnable, *Commun. ACM* **27** (11) (1984).
[51] M. Vilain, P. Koton and M. Chase, On analytical and similarity-based classification, in: *Proceedings Eighth National Conference on Artificial Intelligence*, Boston, MA (MIT Press, Cambridge, MA, 1990).