Cryptographic Limitations on Learning One-Clause Logic Programs

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

Abstract

An active area of research in machine learning is learning logic programs from examples. This paper investigates formally the problem of learning a single Horn clause: we focus on generalizations of the language of constant-depth determinate clauses, which is used by several practical learning systems. We show first that determinate clauses of logarithmic depth are not learnable. Next we show that learning indeterminate clauses with at most k indeterminate variables is equivalent to learning DNF. Finally, we show that recursive constant-depth determinate clauses are not learnable. Our primary technical tool is the method of prediction-preserving reducibilities introduced by Pitt and Warmuth [1990]; as a consequence our results are independent of the representations used by the learning system.

Introduction

Recently, there has been an increasing amount of research in learning restricted logic programs, or *inductive logic programming (ILP)* [Cohen, 1992; Muggleton and Feng, 1992; Quinlan, 1990; Muggleton, 1992a]. One advantage of using logic programs (rather than alternative first-order logic formalisms [Cohen and Hirsh, 1992]) is that its semantics and computational complexity have been well-studied; this offers some hope that learning systems based on it can also be mathematically well-understood.

Some formal results have in fact been obtained; the strongest positive result in the pac-learning model [Valiant, 1984] shows that a single constant-depth determinate clause is pac-learnable, and that a non-recursive logic program containing k such clauses is learnable against any "simple" distribution [Džeroski *et al.*, 1992]. Some very recent work [Kietz, 1993] shows that a single clause is *not* pac-learnable if the constant-depth determinacy condition does not hold; specifically, it is shown that neither the language of indeterminate clauses of fixed depth nor the language of determinate clauses of arbitrary depth is pac-learnable. These negative results are of limited practical importance because they assume that the learner is required to output a *single* clause that covers all of the examples; however, most practical ILP learning systems do not impose this constraint. Such negative learnability results are sometimes called *representation-dependent*.[1]

This paper presents representation *independent* negative results for three languages of Horn clauses, all obtained by generalizing the language of constant depth determinate clauses. These negative results are obtained by showing that learning is as hard as breaking a (presumably) secure cryptographic system, and thus are not dependent on assumptions about the representation used by the learning system. Specifically, we will show that determinate clauses of log depth are not learnable, and that recursive clauses of constant depth are not learnable. We will also show that indeterminate clauses with k "free" variables are exactly as hard to learn as DNF.

Due to space constraints, detailed proofs will not be given; the interested reader is referred to a longer version of the paper [Cohen, 1993a]. We will focus instead on describing the basic intuitions behind the proofs.

Formal preliminaries

Learning models

Our basic notion of learnability is the usual one introduced by Valiant [1984]. Let X be a set, called the *domain*. Define a *concept* C over X to be a representation of some subset of X, and a *language* $\mathcal{L}$ to be a set of concepts. Associated with X and $\mathcal{L}$ are two *size complexity measures*; we use the notation X_n (respectively $\mathcal{L}_n$) to stand for the set of all elements of X (respectively $\mathcal{L}$) of size complexity no greater than n. An *example of* C is a pair (x, b) where $b = 1$ if $x \in C$ and $b = 0$ otherwise. If D is a probability distribution

[1] The prototypical example of a learning problem which is hard in a representation-dependent setting but not in a broader setting is learning k-term DNF. Pac-learning k-term DNF is NP-hard if the hypotheses of the learning system must be k-term DNF; however it is tractable if hypotheses can be expressed in the richer language of k-CNF.

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.

Finally, a language $\mathcal{L}$ is *pac-learnable* iff there is an algorithm *LEARN* and a polynomial function $m(\frac{1}{\epsilon}, \frac{1}{\delta}, n_e, n_t)$ so that for every $n_t > 0$, every $n_e > 0$, every $C \in \mathcal{L}_{n_t}$, every $\epsilon : 0 < \epsilon < 1$, every $\delta : 0 < \delta < 1$, and every probability distribution function D, for any sample S^+, S^- of C from X_{n_e} drawn according to D containing at least $m(\frac{1}{\epsilon}, \frac{1}{\delta}, n_e, n_t)$ examples

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

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

With condition 3 above, the definition of pac-learnability makes a relatively strong restriction on the hypotheses the learner can generate; this can lead to some counterintuitive results.[2] If a learning algorithm exists that satisfies all the conditions above except condition 3, but does output a hypothesis that can be evaluated in polynomial time, we will say that $\mathcal{L}$ is *(polynomially) predictable*.[3]

These learning models have been well-studied, and are quite appropriate to modeling standard inductive learners. However, the typical ILP system is used in a somewhat more complex setting, as the user will typically provide both a set of examples and a *background theory* K: the task of the learner is then to find a logic program P such that P, together with K, is a good model of the data. To deal with this wrinkle, we will require some additional definitions. If $\mathcal{L}$ is some language of logic programs[4] and K is a logic program, then $\mathcal{L}[K]$ denotes the set of all pairs of the form (P, K) such that $P \in \mathcal{L}$: each such pair represents the set of all atoms e such that $P \wedge K \vdash e$. If $\mathcal{K}$ is a set of background theories, then the *family of languages* $\mathcal{L}[\mathcal{K}]$ represents the set of all languages $\mathcal{L}[K]$ where $K \in \mathcal{K}$. We will consider $\mathcal{L}[\mathcal{K}]$ to be predictable (pac-learnable) only when every $\mathcal{L}[K] \in \mathcal{L}[\mathcal{K}]$ is predictable (pac-learnable.) This requires one slight modification to the definitions of predictability pac-learnability: we must now assume a size complexity measure for background theories, and allow the sample and time complexities of the learner to also grow (polynomially) with the size complexity of the background theory provided by the user.

Reducibilities among prediction problems

Our main analytic tool in this paper is *prediction-preserving reducibility*, as described by Pitt and Warmuth [1990]. This is essentially a method of showing that one language is no harder to predict than another. If $\mathcal{L}_1$ is a language over domain X_1 and $\mathcal{L}_2$ is a language over domain X_2, then we say that *predicting $\mathcal{L}_1$ reduces to predicting $\mathcal{L}_2$*, written $\mathcal{L}_1 \trianglelefteq \mathcal{L}_2$, if there is a function $f_i : X_1 \rightarrow X_2$, henceforth called the *instance mapping*, and a function $f_c : \mathcal{L}_1 \rightarrow \mathcal{L}_2$, henceforth called the *concept mapping*, so that the following all hold:

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

Intuitively, $f_c(C_1)$ returns a concept $C_2 \in \mathcal{L}_2$ that will "emulate" C_1 (i.e., make the same decisions about concept membership) on examples that have been "preprocessed" with the function f_i. If predicting $\mathcal{L}_1$ reduces to predicting $\mathcal{L}_2$ and a learning algorithm for $\mathcal{L}_2$ exists, then one possible scheme for learning a concept $C_1 \in \mathcal{L}_1$ would be to preprocess all examples of C_1 with f_i, and then use these preprocessed examples to learn some H that is a good approximation of $C_2 = f_c(C_1)$. H can then 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 [1990] give a rigorous argument that this approach leads to a prediction algorithm for $\mathcal{L}_1$, leading to the following theorem.

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

Restrictions on Logic Programs

In this paper, logic programs will always be function-free and (unless otherwise indicated) nonrecursive. A background theory K will always be a set of ground unit clauses (aka relations, a set of atomic facts, or a model) with arity bounded by the constant a; the symbol a-$\mathcal{K}$ ($\mathcal{K}$ if a is an arbitrary constant) will denote the set of such background theories. The size complexity of a background theory K is its cardinality, and usually will be denoted by n_b. Examples will be represented by a single atom of arity n_e or less; thus we allow the

---

[2] In particular, it may be that a language is hard to learn even though an accurate hypothesis can be found, if it is hard to encode this hypothesis in the language $\mathcal{L}$.

[3] Such learning algorithms are also sometimes called *approximation algorithms*, as in the general case an approximation to the target concept may be produced.

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

head of a Horn clause to have a large arity, although literals in the body have constant arity.[5]

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

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

A determinate clause of depth bounded by a constant i over a background theory $K \in j\text{-}\mathcal{K}$ is called *ij-determinate*. The learning program GOLEM, which has been applied to a number of practical problems [Muggleton and Feng, 1992; Muggleton, 1992b], learns ij-determinate programs. Closely related restrictions also have been adopted by several other inductive logic programming systems, including FOIL [Quinlan, 1991], LINUS [Lavrač and Džeroski, 1992], and GRENDEL [Cohen, 1993c].

As an example, in the determinate clause

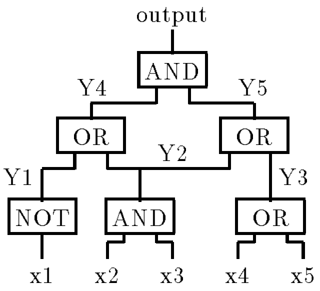
multiply(X,Y,Z) ←
  decrement(Y,W) ∧
  multiply(X,W,V) ∧
  plus(X,V,Z).

W has depth one and V has depth two; thus the clause is 23-determinate.

Learning log-depth determinate clauses

We will first consider generalizing the definition of ij-determinacy by relaxing the restriction that clauses have constant depth. The key result of this section is an observation about the expressive power of determinate clauses: *there is a background theory K such that every depth d boolean circuit can be emulated by a depth d determinate clause over K.* The background theory K contains these facts:

and(0,0,0)	*and(0,1,0)*	*and(1,0,0)*	*and(1,1,1)*
or(0,0,0)	*or(0,1,1)*	*or(1,0,0)*	*or(1,1,1)*
not(0,1)	*not(1,0)*	*true(1)*	



circuit(X1,X2,X3,X4,X5) ←
  not(X1,Y1) ∧
  and(X2,X3,Y2) ∧
  or(X4,X5,Y3) ∧
  or(Y1,Y2,Y4) ∧
  or(Y2,Y3,Y5) ∧
  and(Y4,Y5,Output) ∧
  true(Output)

Figure 1: Reducing a circuit to a clause

The emulation is the obvious one, illustrated by example in Figure 1. Notice an example from the circuit domain is a binary vector $b_1 \ldots b_n$ encoding an assignment to the n boolean inputs to the circuit, and hence must be preprocessed to the atom *circuit*$(b_1, \ldots, b_n)$.

The learnability of circuits has been well-studied. In particular, the language of log-depth circuits (also familiar as the complexity class NC^1) is known to be hard to learn, under cryptographic assumptions [Kearns and Valiant, 1989]. Thus we have the following theorem; here $\mathcal{C}_{ij\text{-DET}}$ denotes the language of logic programs containing a single non-recursive ij-determinate clause.

Theorem 2 *There is a small background theory $K \in 3\text{-}\mathcal{K}$ such that $NC^1 \trianglelefteq \mathcal{C}_{(\log n_e)3\text{-DET}}[K]$. Thus, for $j \geq 3$, the family of languages $\mathcal{C}_{(\log n_e)j\text{-DET}}[j\text{-}\mathcal{K}]$ is not polynomially predictable, and hence not pac-learnable, under cryptographic assumptions.*

It has recently been shown that NC^1 is not predictable even against the uniform distribution [Kharitonov, 1992], suggesting that distributional restrictions will not make log-depth determinate clauses predictable.

Learning indeterminate clauses

We will now consider relaxing the definition of ij-determinacy by allowing indeterminacy. Let the *free variables* of a Horn clause be those variables that appear in the body of the clause but not in the head; we will consider the learnability of the language $\mathcal{C}_{k\text{-FREE}}$, defined as all nonrecursive clauses with at most k free variables. Clauses in $\mathcal{C}_{k\text{-FREE}}$ are necessarily of depth at most k; also restricting the number of free variables is required to ensure that clauses can be evaluated in polynomial time.

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

[5]It should be noted that the parameters n_e, n_b, and n_t (the complexity of the target concept) all measure different aspects of the complexity of a learning problem; as the requirement on the learner is that it is polynomial in all of these parameters, the reader can simply view them as different names for a single complexity parameter $n = \max(n_e, n_b, n_t)$.

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

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

Equivalent clause:
$\text{dnf}(X_1,X_2,X_3,X_4) \leftarrow$
$\text{true}_1(X_1,Y) \wedge \text{false}_1(X_3,Y) \wedge \text{true}_1(X_4,Y) \wedge$
$\text{false}_2(X_2,Y) \wedge \text{false}_2(X_3,Y) \wedge$
$\text{true}_3(X_1,Y) \wedge \text{false}_3(X_4,Y)$.

Figure 2: Reducing a DNF formula to a clause

this language: *for every r, there is a background theory K_r such that every DNF formula with r or fewer terms can be emulated by a clause in $\mathcal{C}_{1\text{-FREE}}[K_r]$.* The emulation is a bit more indirect than the emulation for circuits, but the intuition is simple: it is based on the observation that a clause $p(X) \leftarrow q(X, Y)$ classifies an example $p(a)$ as true exactly when $K \vdash q(a, b_1) \vee \ldots \vee K \vdash q(a, b_r)$, where $b_1, \ldots, b_r$ are the possible bindings of the (indeterminate) variable Y; thus indeterminate variables allow some "disjunctive" concepts to be expressed by a single clause.

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

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

The way in which a formula is emulated is illustrated in Figure 2. The free variable in the emulating clause is Y, and the r possible bindings of Y correspond to the r terms of the emulated DNF formula. Assume without loss of generality that the DNF has exactly r terms[6] and let the i-th term of the DNF be $T_i = \bigwedge_{j=1}^{s_i} l_{ij}$; this will be emulated by a conjunction of literals $C_i = \bigwedge_{j=1}^{s_i} Lit_{ij}$ designed so that C_i will succeed exactly when T_i succeeds and Y is bound to i, or when Y is bound to some constant other than i. This can be accomplished by defining

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

Now, assume that X_i is bound to the value of the i-th boolean variable v_i that is used in the DNF formula; then the conjunction $\bigwedge_{i=1}^{r} C_i$ will succeed when Y is bound to 1 and T_1 succeeds, or when Y is bound to 2 and T_2 succeeds, or ...or Y is bound to r and T_r succeeds. Hence, if we assume that each example $b_1 \ldots b_n$ is preprocessed to the atom $dnf(b_1, \ldots, b_n)$, the clause $dnf(X_1, \ldots, X_n) \leftarrow \bigwedge_{i=1}^{r} C_i$ (in which Y can be bound to any of the r values) will correctly emulate the original formula.

[6] If necessary, null terms $v_1\overline{v_1}$ can be added to make the DNF r terms long.

As a consequence, we have the following theorem:

Theorem 3 *Let DNF_n denote the language of DNF expressions of complexity n or less; for all n there is a polynomial sized background theory K_n such that $DNF_n \trianglelefteq \mathcal{C}_{1\text{-FREE}}[K_n]$. Thus, for all constant k, the family of languages $\mathcal{C}_{k\text{-FREE}}[\mathcal{K}]$ is predictable only if DNF_n is predictable.*

An important question is whether there are languages in $\mathcal{C}_{k\text{-FREE}}$ that are *harder* to learn than DNF. The answer to this questions is no: *for every k and every background theory $K \in a\text{-}\mathcal{K}$, every clause in $\mathcal{C}_{k\text{-FREE}}[K]$ can be emulated by a DNF formula.* Let $Cl = A \leftarrow B_{c_1} \wedge \ldots \wedge B_{c_l}$ be a clause in $\mathcal{C}_{k\text{-FREE}}[K]$. As we assume clauses are nonrecursive, Cl covers an example e iff

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

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

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

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

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

We can thus strengthen Theorem 3 as follows:

Theorem 4 *The family of languages $\mathcal{C}_{k\text{-FREE}}$ is predictable if and only if DNF is predictable.*

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

Learning recursive clauses

Finally, we will consider learning a single *recursive* ij-determinate clause. A realistic analysis of recursive clauses requires a slight extension of our formalism: rather than representing an example as a single atom of arity n_e, we will (in this section of the paper only) represent an example as a ground *goal atom*, e, plus a set of up to n_e ground *description atoms* $D = \{d_1, \ldots, d_{n_e}\}$ of arity bounded by a. A program P now classifies an example (e, D) as true iff $P \wedge K \wedge D \vdash e$. This also allows structured objects like lists to be used in examples; for example, an example of the predicate *member(X,Ys)* might be represented as the goal atom e =*member(b,list_ab)* together with the description

$$D = \{ \text{head(list\_ab,a), tail(list\_ab,list\_b),} \\ \text{head(list\_b,b), tail(list\_b,nil)} \}$$

This formalism follows the actual use of learning systems like FOIL. Finally, in order to talk sensibly about one-clause recursive programs, we will also assume that the non-recursive "base case" of the target program is part of D or K.

Again, the key result of this section is an observation about expressive power: *there is a background theory K_n such that every log-space deterministic (DLOG) Turing machine M can be emulated, on inputs of size n or less, by a single recursive ij-determinate clause.* Since DLOG Turing machines are cryptographically hard to predict, this will lead to a negative predictability result for recursive clauses.

Before describing the emulation, we will begin with some basic facts about DLOG machines. First, we can assume, without loss of generality, that the tape alphabet is $\{0, 1\}$. Then a DLOG machine M accepting only inputs of size n can be encoded by a series of *transitions* of the following form:

if $x_i = b$ and CONFIG=c_j **then** let CONFIG:=c'_j

where x_i denotes the value of the i-th square of the input tape, $b \in \{0, 1\}$, and c_j and c'_j are constants from a polynomial-sized alphabet $CON = \{c_1, \ldots, c_{p(n)}\}$ of constant symbols denoting *internal configurations* of M.[7] One can also assume without loss of generality that there is a unique starting configuration c_0, a unique accepting configuration c_{acc}, and a unique "failing" configuration c_{fail}, and that there is exactly one transition of the form given above for every combination of $i : 1 \leq i \leq n$, $b \in \{0, 1\}$, and $c_j \in CON - \{c_{acc}, c_{fail}\}$. On input $X = x_1 \ldots x_n$ the machine M starts with CONFIG=c_0, then executes transitions until it reaches CONFIG=c_{acc} or CONFIG=c_{fail}, at which point X is accepted or rejected (respectively).

To emulate M, we will preprocess an example $S = b_1 \ldots b_n$ into the goal atom e and the description D defined below:

$$e \equiv tm(new_S, c_0)$$

$$D \equiv \{bit_i(new_S, b_i)\}_{i=1}^{n} \cup \{tm(new_S, c_{acc})\}$$

Now we will define the following predicates for the background theory K_n. First, for every possible $b \in \{0, 1\}$ and $j : 1 \leq j \leq p(n)$, the predicate $stat_{b,j}(B,C,Y)$ will be defined so that given bindings for variables B and C, $stat_{b,j}(B,C,Y)$ will fail if $C = c_{fail}$; otherwise it will succeed, binding Y to *active* if $B = b$ and $C = c_j$ and binding Y to *inactive* otherwise. Second, for $j : 1 \leq j \leq p(n)$, the predicate $next_j(Y,C)$ will be true if $Y = active$ and $C = c_j$, or if $Y = inactive$ and $C = c_{acc}$. It is easy to show that these definitions require only polynomially many facts to be in K_n.

$$TRANS_{ibj} \equiv \mathrm{bit}_i(\mathrm{S},\mathrm{B}_{ibj}) \wedge \mathrm{stat}_{b,j}(\mathrm{C},\mathrm{B}_{ibj},\mathrm{Y}_{ibj}) \wedge \\ \mathrm{next}_{j'}(\mathrm{Y}_{ibj},\mathrm{C1}_{ibj}) \wedge \mathrm{tm}(\mathrm{S},\mathrm{C1}_{ibj})$$

Given K_n and D, and assuming that S is bound to new_S and C is bound to some configuration c, this conjunction will fail if $c = c_{fail}$; otherwise, it will succeed trivially if $x_i \neq b$ or $c \neq c_j$[8]; finally, if $x_i = b$ and $c = c_j$, $TRANS_{ibj}$ will succeed only if the atom $tm(new_S,c_{j'})$ is provable.[9] From this it is clear that the one-clause logic program

$$\mathrm{tm}(\mathrm{S},\mathrm{C}) \leftarrow \bigwedge_{\substack{i \in \{1,\ldots,n\},\ b \in \{0,1\} \\ j \in \{1,\ldots,p(n)\}}} TRANS_{ibj}$$

will correctly emulate the machine M. Thus, letting $\mathcal{R}_{ij\text{-DET}}$ denote the language of one-clause ij-determinate recursive logic programs, and letting DLOG_n represent the class of problems on inputs of size n computable in deterministic space $\log n$, we have the following theorem.

Theorem 5 *For all n there is a background theory K_n such that $DLOG_n \trianglelefteq \mathcal{R}_{33\text{-DET}}[K_n]$. Thus the family of languages $\mathcal{R}_{ij\text{-DET}}[\mathcal{K}]$ is not polynomially predictable, and hence not pac-learnable, under cryptographic assumptions.*

Although this result is negative, it should be noted that (unlike the case in the previous theorems) the preprocessing step used here distorts the distribution of examples: thus this result does not preclude the possibility of distribution-specific learning algorithms for recursive clauses.

Concluding remarks

This paper presented three negative results on learning one-clause logic programs. These negative results are stronger than earlier results [Kietz, 1993] in a number of ways; most importantly, they are not based on restricting the learner to produce hypotheses in some

[7] An internal configuration encodes the contents of M's worktape, along with all other properties of its internal state relevant to the computation. Since M's worktape is of length $\log n$ it can have only n different contents; thus there are only a polynomial number $p(n)$ of internal configurations for M.

[8] In this case Y_{ibj} will be bound to *inactive* and $C1_{ibj}$ will be bound to c_{acc}—the recursive call to *tm/2* succeeds because $tm(new_S,c_{acc}) \in D$.

[9] In this case, Y_{ibj} will be bound to *active* and $C1_{ibj}$ will be bound to $c_{j'}$.

designated language. Instead, they are obtained by showing that learning is as hard as breaking a secure cryptographic system.

In particular, we have shown that several extensions to the language of constant-depth determinate clauses lead to hard learning problems. First, allowing depth to grow as the log of the problem size makes learning a single determinate clause as hard as learning a log-depth circuit, which is hard under cryptographic assumptions; this shows that all learning algorithms for determinate clauses will require time worse than exponential in the depth of the target concept. Second, indeterminate clauses with k "free" variables (a restriction of the language of constant-depth indeterminate clauses) are exactly as hard to learn as DNF; the learnability of DNF is a long-standing open problem in computational learning theory.

Finally, adding recursion to the language of ij-determinate clauses makes them as hard to learn as a log-space Turing machine; again, this learning problem is cryptographically hard. There are however, learnable classes of one-clause linearly recursive clauses; a discussion of this is given in a companion paper [Cohen, 1993b].

Acknowledgements

Thanks to Mike Kearns and Rob Schapire for several helpful discussions, and to Susan Cohen for proofreading the paper.

References

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

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

(Cohen, 1993a) William W. Cohen. Learnability of restricted logic programs. In *Proceedings of the Third International Workshop on Inductive Logic Programming*, Bled, Slovenia, 1993.

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

(Cohen, 1993c) William W. Cohen. Rapid prototyping of ILP systems using explicit bias. In preparation, 1993.

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

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

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

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

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

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

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

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

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

(Pitt and Warmuth, 1990) Leonard Pitt and Manfred Warmuth. Prediction-preserving reducibility. *Journal of Computer and System Sciences*, 41:430–467, 1990.

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

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

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