PAC-Learning PROLOG clauses with or without errors

Rosario Gennaro*
Laboratory for Computer Science
Massachusetts Institute of Technology
`rosario@theory.lcs.mit.edu`

April 7, 1994

Abstract

Recently researchers have been interested in trying to expand the domain of learnability to subsets of first-order logic, in particular Prolog programs. This new research area has been named *Inductive Logic Programming* (ILP).

In a nutshell we can describe a generic ILP problem as following: given a set $\mathcal{E}$ of (positive and negative) examples of a target predicate, and some background knowledge $\mathcal{B}$ about the world (usually a logic program including facts and auxiliary predicates), the task is to find a logic program $\mathcal{H}$ (our hypothesis) such that all positive examples can be deduced from $\mathcal{B}$ and $\mathcal{H}$, while no negative example can.

In this paper we review some of the results achieved in this area and discuss the techniques used. Moreover we prove the following new results:

- Predicates described by non-recursive, local clauses of at most k literals are PAC-learnable under any distribution. This generalizes a previous result that was valid only for constrained clauses.
- Predicates that are described by k non-recursive local clauses are PAC-learnable under any distribution. This generalizes a previous result that was non constructive and valid only under some class of distributions.

Finally we introduce what we believe is the first theoretical framework for learning Prolog clauses in the presence of errors. To this purpose we introduce a new noise model, that we call the *fixed attribute noise* model, for learning propositional concepts over the Boolean domain. This new noise model can be of its own interest.

Keywords: Inductive Logic Programming, PAC-learning, noise models.

*Supported by NSF grant no. 9121466-CCR and by a graduate fellowship of Consiglio Nazionale delle Ricerche, Italy

1 Introduction

Machine learning theory has been very successful in the past years in the field of propositional logic. Despite these results (both positive and negative) propositional learning has the drawback of the limited expressiveness of the hypothesis language. Recently, researchers have been interested in trying to expand the domain of learnability to subsets of first-order logic. Prior work has established a strong negative result: Haussler in [9] shows that (even highly restricted) conjunctions in first-order logic cannot be learned in the Valiant (PAC) model [28]. These negative results prompted people to look at other more specific subsets of first-order logic. In particular a natural choice was to look at Prolog programs. This new research area has been named *Inductive Logic Programming* (ILP).

In a nutshell we can describe a generic ILP problem as following: given a set $\mathcal{E}$ of (positive and negative) examples of a target predicate, and some background knowledge $\mathcal{B}$ about the world (usually a logic program including facts and auxiliary predicates), the task is to find a logic program $\mathcal{H}$ (our hypothesis) such that all positive examples can be deduced from $\mathcal{B}$ and $\mathcal{H}$, while no negative example can.

The interest in this area has been paying off especially in terms of practical applications: systems that infer restricted class of Prolog programs have been successfully applied to real-world problems like structure-activity prediction for drug design [14], protein secondary structure prediction [20] or finite element mesh design [5].

Born as a branch of logic programming, ILP has been attracting more attention from the Computational Learning Theory community in an effort to provide this new area with solid theoretical grounds. In this paper we will try to describe and continue this effort.

1.1 Previous and new results

Probably the father of ILP can be considered Plotkin. His thesis [23] is not limited just to Horn clause logic (Prolog was not around yet). He gave an algorithm that returns the least general clause covering all the positive examples and no negative example. Muggleton and Feng in [19] show that, if restricted to the class of determinate logic programs [1], Plotkin's method works efficiently.

Numerous papers have appeared recently on the subject of PAC-learning logic programs. In particular the following results have been achieved:

1. in [7] the authors prove that predicates defined by non-recursive, constrained clauses of at most k literals are PAC-learnable under any distribution.

2. in [6] the authors prove PAC-learnability of predicates that are described by k determinate non-recursive clauses under a broad class of distributions.

3. Cohen [4] improves on (2) by proving that predicates defined by k non-recursive local clauses is PAC-learnable under the same broad class of distributions. Moreover he proves that further relaxations of the constraints on the hypothesis leads to hardness in learning.

[1]The definitions of constrained, determinate and local clauses is postponed to later in the paper. For now it's just interesting to notice that constrained clauses are a subclass of determinate clauses that are a subclass of local clauses.

Unfortunately results (2) and (3) are not constructive (we shall see why later). In this paper we will survey these results and improve on them by proving the following facts:

a. Predicates described by non-recursive, local clauses of at most k literals are PAC-learnable under any distribution.

b. Predicates that are described by k non-recursive local clauses are PAC-learnable under any distribution.

Both results are constructive and use the fact that we can express our output as a predicate belonging to a larger class. For applications in the real world it is necessary to analyze robustness of ILP systems. However there has been little work done on the effect of noise or errors in the training of ILP system, mostly empirical comparisons of the heuristics used by various implementations when dealing with noise (see for example [15]). In this paper we will present what we believe is the first theoretical treatment of the problem of the effects of noise and errors in ILP.

1.2 Outline of the paper

The paper continues as following. In the next section we will give a brief overview of logic programming. We will introduce some definitions and notations for the rest of the paper.

In Section 3 we will give an example of an ILP problem. To gain also some motivation we decided to describe a real world application. We will talk about prediction of protein structures.

Section 4 will describe GOLEM, the system implemented by Muggleton and Feng based on the ideas of Plotkin.

Section 5 will describe the PAC-learnability results.

Section 6 will contain a discussion on the effect of noise and errors on PAC-learning logic programs and we will propose a theoretical framework for ILP with imperfect data.

Section 7 will conclude the paper with some final remarks and open questions.

2 Preliminaries

In this section we will review the basics of logic programming. Our presentation is somewhat simplified according to the scope of this paper. For a complete description of the field the reader is referred to Lloyd's classic text [17].

Logic programs are written over an alphabet consisting of:

- *constants* that we will denote with lower case letters $b, c, \ldots$
- *predicates* which we will represent with letters like $p, q, r, \ldots$
- *variables* always represented with upper case letters $X, Y, Z, \ldots$

A *literal* A is an application of a predicate to variables and/or facts: $A = p(X_1, \ldots, X_a)$. a is the *arity* of the literal. A *fact* f is an application of a predicate to constants: $f = p(c_1, \ldots, c_a)$. Facts are also called *ground* literals

A *substitution* is a partial function mapping variables to other variables or constants. We will denote them with greek letters like θ or σ. If A is a literal we will denote with $A\theta$ the result of applying substitution θ to the variables appearing in A. A fact f is an *instance* of a literal A if $f = A\theta$ for some substitution θ.

A *definite clause* is a clause of the form: $A \leftarrow B_1 \wedge \ldots \wedge B_l$ where $A, B_1, \ldots, B_l$ are literals. A is called the *head* and $B_1, \ldots, B_l$ the *body* of the clause. Variables in the head of the clause are implicitly universally quantified, while variables in the body not appearing in the head are implicitly existentially quantified. In the following we will omit the word "definite".

A *logic program* $\mathcal{P}$ is a collection of clauses (facts can be considered clauses with an empty body). The *extension* of a logic program is the set of facts that can be proven by the program: $E[\ \mathcal{P}\ ] = \{f : \mathcal{P} \vdash f\}$

Example 1 Suppose that our logic program consists of the following clauses

$$
\begin{aligned}
\mathcal{P} = \quad & mother(ann, bob) \\
& mother(ann, charlie) \\
& father(bob, julie) \\
& father(john, chris) \\
& siblings(X, Y) \leftarrow mother(V, X) \wedge mother(V, Y) \\
& siblings(X, Y) \leftarrow father(U, X) \wedge father(U, Y)
\end{aligned}
$$

so the extension of $\mathcal{P}$ contains the facts in $\mathcal{P}$ plus the extra fact $siblings(bob, charlie)$

2.1 The learning problem

The formal logical setting for an ILP problems is as follows. The learner is provided with some *background knowledge* $\mathcal{B}$, that is just a logic program containing facts and definitions of predicates. He or she is then given a set $\mathcal{E}^+$ of positive examples and a set $\mathcal{E}^-$ of negative examples, i.e. facts that are true or false, possibly involving predicates not defined in $\mathcal{B}$. The learner's task is to find another logic program $\mathcal{H}$ such that

$$\mathcal{B} \wedge \mathcal{H} \vdash \mathcal{E}^+$$

$$\mathcal{B} \wedge \mathcal{H} \wedge \mathcal{E}^- \not\vdash \square$$

where $\square$ is the empty clause that stands for contradiction.

Example 2 In the previous example we could set the database as a collection of facts like

$$
\begin{aligned}
\mathcal{B} = \quad & mother(ann, bob) \\
& mother(ann, charlie) \\
& father(bob, julie) \\
& father(john, chris) \\
& \ldots
\end{aligned}
$$

and then have $\mathcal{E}^+ = grandmother(ann, julie), \ldots$ and $\mathcal{E}^- = grandmother(ann, chris), \ldots$ and then try to infer that

$$
\begin{aligned}
\mathcal{H} = \quad & grandmother(X, Y) \leftarrow mother(X, Z) \wedge father(Z, Y) \\
& grandmother(X, Y) \leftarrow mother(X, U) \wedge mother(U, Y)
\end{aligned}
$$

This formalization does not guarantee a good performance of the hypothesis on unseen examples. For that, we should try to formalize the problems in terms of the PAC model as defined by Valiant in [28]. Let's just recall it for sake of completeness. Let X be a set called the *domain*. A *concept* c over X is just a subset of X. A *concept class* C is a family of concepts. An example of c is a pair (e, l) where $e \in X$ and $l = 1$ if $e \in c$, $l = 0$ otherwise. If D is a probability distribution over X, a *sample* S_D is a collection of examples drawn according to D. With $w_D(c)$ we define the "weight" of the subset $c \subseteq X$ according to D.

Definition 1 *A concept class C is PAC-learnable if there exists an algorithm L such that for any $c \in C$, any distribution D, any values $0 < \epsilon < 1$, $0 < \delta < 1$, when given as input a sample S_D,ϵ, δ outputs an hypothesis concept h such that with probability at least $1 - \delta$, $w_D(h \bigtriangledown c) < \epsilon$.*[2]

We usually require the algorithm L to be *efficient* i.e. to run in time polynomial in ϵ^{-1}, δ^{-1} and the size of an encoding of the concept class. In particular this means that the sample size (often referred as sample complexity) must be polynomial in the same quantities. We can relax the condition $h \in C$ and allow the algorithm to output $h \in C'$ where C' is a larger concept class.

This is the standard PAC-learning formalization. In the ILP case however we have to take in account the fact that the learner is not provided just with examples but with a database of background knowledge $\mathcal{B}$. Cohen in [4] addresses this issue and proposes the following approach: if C is a family of logic programs, each database $\mathcal{B}$ will define a particular concept class $C[\mathcal{B}] = \{< \mathcal{P}, \mathcal{B} > \text{ for } \mathcal{P} \in C\}$. Each pair $< \mathcal{P}, \mathcal{B} >$ represents the extension of the logic program $\mathcal{P} \wedge \mathcal{B}$. Then if DB is a collection of databases the $C[DB]$ is the set of all concept classes $C[\mathcal{B}]$ for $\mathcal{B} \in DB$

Definition 2 *The concept class family $C[DB]$ is (uniformly) PAC-learnable if there is an algorithm L that, given as input a database $\mathcal{B} \in DB$, PAC-learns the concept class $C[\mathcal{B}]$.*

2.2 Restricting the hypothesis space

The way the problem is stated, it looks hopelessly hard to find a general method to solve it. We will then try to put some restrictions on the family of logic programs and databases that we will take into consideration. Hopefully these restrictions will not be too severe to compromise the expressiveness of the language, but at the same time will allow us to achieve positive learnability results.

Bounded arity predicates: We will assume that there is a constant a that bounds the arity of all the predicates in the background knowledge of our problem.

Ground background knowledge: Some of the implemented ILP systems (like GOLEM) require the background knowledge to be composed only by facts. If this is not the case, before starting the learning process the system transforms the background knowledge $\mathcal{B}$ into a set of facts by building the h-easy ground model of $\mathcal{B}$ i.e. the set of facts that can

[2] $h \bigtriangledown c$ denotes the symmetric difference of h and c i.e. the set of elements that belong to one among h and c but not to the other

be proven from $\mathcal{B}$ with at most h binary resolutions starting from the constants symbols in the examples. h is usually a user-defined parameter

Efficient background knowledge: Other algorithms (notably all the PAC-learning algorithms we will see) simply query the background knowledge by calling a Prolog interpreter on it to prove a particular fact. In this case we require that all atomic queries can be answered in time polynomial in the arity of query predicate.

Single predicate program: We will assume that we are trying to infer a logic program consisting of definitions of a single predicate and that the examples are all positive or negative instantiations of this unknown predicate. We may make the assumption that the predicate is defined by k clauses. Or that each clause that defines the predicate has at most k body literals. The definition of this predicate may be non-recursive.

Constrained clauses: A clause is constrained when all the variables in the body of the clause appear also in the head of the clause. A constrained logic program is one composed just by constrained clauses.

Determinate clauses of constant depth: This restriction was first introduced by Muggleton and Feng in [19]. It's a generalization of the constrained condition. Consider the clause $A \leftarrow B_1 \wedge \ldots \wedge B_l$. We define the *input* variables of literal B_i those variables that also appear in the clause $A \leftarrow B_1 \wedge \ldots \wedge B_{i-1}$, *output* variables all the others. A literal B_i is *determinate* (with respect to the background knowledge $\mathcal{B}$) if for every possible substitution σ such that $A\sigma = f$ for some fact f and $\mathcal{B} \vdash B_1\sigma, \ldots, B_{i-1}\sigma$ then there is at most one possible substitution θ such that $\mathcal{B} \vdash B_i\sigma\theta$. I.e. given $\mathcal{B}$ and the binding of the input variables there is at most one possible binding for the output variables of a determinate literal. A clause is determinate if all of its literals are determinate. Informally a determinate logic program can be evaluated by a Prolog program without backtracking. Notice how the order of the literals in the clause is important.

Now we define the depth of a variable in a clause $A \leftarrow B_1 \wedge \ldots \wedge B_{i-1}$. Variables appearing in the head of the clause will have depth zero. Then let V be a variable and B_i be the first literal containing it. Then variable V has depth $d+1$ where d is the maximal depth of the input variables of B_i. The depth of a clause is the maximum depth of any variable in it. We will assume that the target predicate is defined by clauses whose depth is bounded by a constant.

Example 3 To illustrate the above definitions consider the definition of the predicate *grandmother* given above. The clauses are not constrained (the variables Z and U are free). They are not determinate either. In fact given a binding for X, the variable Z can be bound to any of the possible children of X (for example if $X = ann$ then Z could be bound to *bob* or *charlie*). Notice however that if we rewrite the definition in the following way:

$$\begin{aligned}\mathcal{H} = \quad & grandmother(X,Y) \leftarrow father(Z,Y) \wedge mother(X,Z) \\ & grandmother(X,Y) \leftarrow mother(U,Y) \wedge mother(X,U)\end{aligned}$$

then the clause becomes determinate (since fixed Y there is only one possible father/mother for him/her). Notice also that the arity is bounded by 2, and the depth of any variable is 1.

Local clauses: Consider the clause $A \leftarrow B_1 \wedge \ldots \wedge B_l$. Let X and Y be two free (i.e. not appearing in the head) variables. We say that X *touches* Y if they appear in the same literal. We say that X *influences* Y if X touches Y or touches a free variable Z that influences Y. Using this definition we can partition the literals of a clause in *locales*. Two free variables are in the same locale if they influence each other. The *locality* of a clause is the size of its biggest locale. Local clauses are clauses of constant locality.

Example 4 Consider the following clause:

$$\begin{array}{lll} sameagecousins(X,Y) \leftarrow & father(V,X) \wedge father(Z,Y) & (*) \\ & siblings(V,Z) \wedge & (*) \\ & age(X,A) \wedge age(Y,A) & (**) \end{array}$$

This clause has locality 3. Indeed there is one locale of size 3: the conjunction $father(V,X) \wedge father(Z,Y) \wedge siblings(V,Z)$, since V touches Z. And there is a locale of size 2: the conjunction $age(X,A) \wedge age(Y,A)$ since A does not touch any other free variable.

All of these restrictions are not very limiting, except for the ones dealing with the structure of the clause (constrained, determinate, local). Unfortunately as we will see in the following, Cohen [4], proves some negative results on the PAC-learnability of more expressive classes.

3 A case study: predicting protein structure

The family tree example of the previous section was adequate to exemplify the definitions, but not to give a real flavor of what kind of problems ILP can be applied to. In this section we will show how a very important biological question can be posed in terms of inductive logic programming. This section is intended to give a relevant example and some motivation. It may be skipped since it does not prejudicate the comprehension of the rest of the paper. The work synthesized in this section is reported in [20]. Interested readers are referred to that paper for more details.

A very active research area in molecular biology is the prediction of secondary structure of proteins from their primary structure. Just to make things understandable we can think of proteins as long sequences of aminoacids or *residues*. There are only a few aminoacids in nature, but they can assemble in many different ways to create different proteins. The list itself of residues is the primary structure of the protein. These long chains of residues can structure themselves in helices (so called α/α type) or in strands (so called β/β) or in alternate helices and strands (so called α/β). This is the secondary structure.

Some proteins have been already classified according to this scheme. The Brookhaven database contains all the information about proteins of which we know the secondary structure. This gives us a great opportunity of using this database as our set of examples.

Suppose we want to understand the rule that governs the formation of α-helices in a protein. Then we would have to infer a logical clause of the form $alpha(Prot, Pos)$ that is true when the residue in position Pos of protein $Prot$ is part of an α-helix.

The background knowledge given to the learner contains information about protein structure and chemical properties of the various residues. First of all contains the primary structure of the proteins: the fact $1structure(Prot, Pos, Res)$ is true when the residue in position Pos of protein $Prot$ is exactly Res. Then some information on the geometric structure of α-helices. And finally a database of facts about the chemical properties of aminoacids, for example the fact $aromatic(Res)$ is true when the residue Res is aromatic.

An example of clause that could be inferred by the program could be (we are in the realm of science fiction here, for a list of the real clauses inferred by GOLEM please refer to [20])

$$\begin{array}{rll} alpha(Prot, Pos) \leftarrow & 1structure(Prot, Pos, Res) & \wedge \\ & aromatic(Res) & \wedge \\ & Pos3 = Pos + 3 & \wedge \\ & 1structure(Prot, Pos3, Res1) & \wedge \\ & hydrophobic(Res1) & \end{array}$$

This clause expresses the theory that a residue is in an α-helix when it's aromatic and the residue at distance 3 from it is hydrophobic. As an aside notice that the clause above is determinate.

GOLEM has been used to perform this task with quite some success. It achieved a prediction accuracy on unseen examples of 81%. The best previously reported result for this type of prediction was 76% obtained using neural network. This approach has the advantage over neural networks of producing results that are more understandable.

As in any real-world application, the problem of noise has to be addressed. The Brookhaven database is not immune from errors, so some of our training examples may be misclassified. More seriously the background knowledge may be incomplete or inappropriate (for example we may not know some of the chemical properties for some aminoacid). GOLEM deals with this problem by allowing the inferred rule to cover some of the negative examples. We will come back to the issue of noise (intended either as misclassification of examples or as incompleteness of the background knowledge) later in the paper.

4 GOLEM: the logic-based approach

In [19] Muggleton and Feng describe GOLEM, a system that infers efficiently determinate constant-depth clauses. It uses Plotkin's results on Relative Least General Generalization (RLGG) of clauses. In this section we will present their approach.

4.1 Relative Least General Generalization

Informally we can consider Plotkin's notion of RLGG as the inverse of the concept of most general unifier of Robinson [24]. As unification (substitution of variables with terms) proves helpful in deduction, so generalization (substituting terms with variables) is useful for induction. We will discuss the aspects of Plotkin's work that are relevant to the ILP framework.

According to Plotkin we say that a clause C is *more general* than a clause D (or $C \preceq D$) if there is a substitution θ such that $C\theta \subseteq D$. This relationship defines a lattice among the set of clauses: given two clauses D_1, D_2 we can define their *least general generalization* (lgg) as their greatest common lower bound in the relationship $\preceq$, i.e. $D = lgg(D_1, D_2)$ iff

- $D \preceq D_1$ and $D \preceq D_2$
- for all $D' \preceq D_1, D_2$ then $D' \preceq D$

Now suppose we have some background knowledge $\mathcal{B}$ consisting only of facts $\mathcal{B} = \bigwedge_{i=1}^{b} B_i$. And suppose we have two positive examples of the unknown predicate $E_1, E_2 \in \mathcal{E}^+$. The least general generalization of E_1 and E_2 relative to $\mathcal{B}$ is the least general clause C such that $\mathcal{B} \wedge C \vdash E_1 \wedge E_2$. We will write $C = RLGG(E_1, E_2)$. Now

$$\mathcal{B} \wedge C \vdash E_1$$

implies that

$$C \vdash \mathcal{B} \rightarrow E_1$$

or in other words

$$C \preceq (\bigwedge_{i=1}^{b} B_i) \rightarrow E_1$$

Similarly for E_2. Let $C_i = (\bigwedge_{i=1}^{b} B_i) \rightarrow E_i$ (for $i = 1, 2$) so by definition $C = lgg(C_1, C_2)$. So for the case of ground knowledge we reduced the problem of computing RLGG to the one of computing lgg. The requirement of ground background knowledge is necessary since Plotkin proves that in the general case the RLGG clause needs not to be finite.

Plotkin gives the following algorithm to compute lgg. The lgg of two terms $f(t_1, \ldots, t_n)$, $g(s_1, \ldots, s_m)$ with $f \neq g$ is a new variable X that will represent this pair of terms from now on. The lgg of two terms $f(t_1, \ldots, t_n)$, $f(s_1, \ldots, s_n)$ is the term $f(lgg(t_1, s_1), \ldots, lgg(t_n, s_n))$. Same for two literals $p(t_1, \ldots, t_n)$, $p(s_1, \ldots, s_n)$. If the two literals have different sign or predicate symbol then the lgg is undefined. The lgg of two clauses C_1, C_2 is the clause $C = \{\ l = lgg(l_1, l_2) \text{ for } l_1 \in C_1\ ,\ l_2 \in C_2\ \}$. [3]

Example 5 Let's go back to the family tree. Suppose our background knowledge is:

$$\begin{aligned} \mathcal{B} = \quad & mother(ann, bob) \\ & mother(ann, charlie) \\ & mother(nancy, jesse) \\ & father(bob, julie) \\ & father(jesse, lucas) \end{aligned}$$

and that we have the positive examples

$$E_1 = grandmother(ann, julie)$$

$$E_2 = grandmother(nancy, lucas)$$

applying the above method we would get that $RLGG(E_1, E_2)$ is :

$$\begin{array}{llll} grandmother(X, Y) \leftarrow & mother(ann, V_1) & \wedge & \\ & mother(X, V_2) & \wedge & (*) \\ & mother(X, V_3) & \wedge & \\ & father(V_2, Y) & \wedge & (*) \end{array}$$

The literals marked with a star are the correct ones, the others are logically redundant.

[3] Intuitively this is because a new variable is introduced when we find two different terms in the same place in two compatible literals (i.e literals with the same predicate and same sign). Incompatible literals cannot be generalized. The recursive definition makes sure we find the *least* general generalization

The example above shows us that RLGGs usually contain more literals that we really need to. In particular if b is the size of the background knowledge we have that $RLGG(E_1, E_2)$ will have $O(b^2)$ literals and it appears evident that if we construct the RLGG of m examples the size of the resulting clause will be exponential in m. Plotkin was aware of that and he suggested the use at each step of theorem proving to eliminate the redundant literals in the clause. This approach is of course semi-decidable (or in any case highly inefficient). Moreover even after reduction the inferred clause may still have a large number of literals.

Muggleton and Feng in [19] show that if we assume that the hidden predicate is defined by a determinate clause of depth i over a background knowledge of arity a, then the size of the RLGG of m examples does not depend on m (it is however $O(b^{i^a})$). Notice moreover that RLGGs use only positive examples and may at the end cover some of the negative examples as well.

4.2 The implementation

To solve this problem GOLEM follows a "greedy" covering strategy. First of all it checks the background knowledge $\mathcal{B}$ and if $\mathcal{B}$ is not ground reduce it to its h-easy model. Then given the sets $\mathcal{E}^+$ and $\mathcal{E}^-$ of positive and negative examples, it samples random pairs from $\mathcal{E}^+$, constructs the RLGGs of those random pairs and chooses the one that covers the maximum number of positive examples but no negative example. It adds this clause (possibly reducing it somewhat) to the hypothesis and then starts again, iterating this process until no positive examples are left.

GOLEM has a very rudimentary noise-handling mechanism. It works by fixing a fraction of the negative examples that the hypothesis is allowed to cover. Other ILP systems like LINUS and FOIL have more sophisticated noise-handling mechanisms (see [15])

5 PAC-learnability results

As we said in the introduction, recently there has been some attempt to provide ILP with solid theoretical grounds. In particular some work has been done trying to formalize ILP in terms of the Valiant PAC model [28]. In this section we will describe these results and our improvements to them. Section 2.2 contains all the definitions about the subclasses we are going to consider.

In the following we will assume k to be a constant, b the size of the background knowledge and m the number of examples drawn. We will assume also that we are trying to infer a single predicate $q(X_1, X_2, \ldots, X_n)$ defined by non-recursive function-free local clauses over a ground or efficient background knowledge $\mathcal{B}$ of constant arity a. Let $p_1, \ldots, p_l$ be the predicates defined in $\mathcal{B}$.

5.1 Reducing the problem to a propositional one

We can think of q to be defined as following:

$$q(X_1, \ldots, X_n) \leftarrow \Phi(X_1, \ldots, X_n, V_1, \ldots, V_s)$$

(where V_j are the free variables) and Φ is a DNF first order formula built from literals using $p_1, \ldots, p_l$. Each term of the DNF is the body of one of the clauses defining q. Notice that to be able to output a logic program we have to learn Φ in a DNF form, otherwise we will not be able to reconstruct the clauses defining q.

Using a technique due to Cohen [4] we show how to reduce the problem of learning such a predicate to the one of learning a propositional concept over $\{0,1\}^{poly(n)}$.

Since there are n variables in the head of the clause and each literal in the body can introduce at most a new variables, a locale of size i can contain at most $n + ai$ distinct variables. So the number of different locales is at most $(l(n+ai)^a)^i$. We build a list F of variables x_j such that each variable corresponds to every possible locale. We transform each example $q(c_1, \ldots, c_n)$ in a propositional one by querying the background knowledge to see if each locale is true or not once we fix $X_j = c_j$. At this point we have to learn a propositional DNF formula. To go back to Φ we just substitute to each variable in the DNF its correspondent locale.

David McAllester pointed out to us that this technique allows the learner to reconstruct only definitions that do *not* contain constants as arguments but only variables.

Since learning DNF is an open problem, we will consider two subclasses: predicates defined by local clauses with a most k literals and predicates defined by at most k local clauses.

5.2 Local Clauses with at most k literals

Now suppose that the predicate we are trying to learn is defined by local clauses with at most k body literals. This is equivalent to say that Φ is a k-DNF first order formula. In particular there will be at most k locales for each term of the DNF. So the transformed propositional problem will be learning a k-DNF formula. After we do this using the standard algorithm, when go back to the first-order case we may end up with clauses containing more of k literals. So we are not learning this class by itself but by a larger class.

Theorem 3 *The class of predicates defined by non-recursive clauses of constant locality i containing at most k literals is PAC-learnable under any distribution by the larger class of predicates defined by clauses of constant locality i.*

This improves on [7] where a similar result was obtained for constrained clauses. They learn the class of predicates defined by constrained clauses containing at most k literals by itself. We improve on that by allowing a larger output class.

5.3 Predicates defined by k local clauses

The situation becomes more complicated when we approach the problem of learning predicates defined by k local clauses. Now the corresponding propositional problem becomes the one of learning a k-term DNF formula. Unfortunately for $k \geq 2$, k-term DNF are not PAC-learnable by k-term DNF (under the assumption **NP**$\neq$**RP** [21]). As we said above it does not help to be able to learn k-term DNF by k-CNF since we will not then be able to reconstruct and output a logic program as an answer. Li and Vitanyi [16] prove that under the class of *simple* distributions (it includes all the computable ones) we can learn k-term

DNF by k-term DNF. Cohen [4] uses this result to prove that the class of predicates defined by k local clauses is learnable under simple distribution. The result in [16] however is not constructive, in fact it requires sampling according to a *universal* distribution that is not computable (or in some restricted case exponential time computable).

We follow a different approach. We use the algorithm by Blum and Singh [3] to learn k-term DNF by the larger class of general DNF formulae. They show that k-term DNF over $\{0,1\}^n$ are learnable by DNF formulae containing $O(n^{k-1})$ terms under *any distribution.* If we apply this result to our problem we will end up with a logic program for q containing more than k clauses, so again we need the assumption of being able to output a logic program from a larger class than the original one. So we can state the following:

Theorem 4 *The class of predicates defined by k non-recursive clauses of constant locality i is PAC-learnable under any distribution by the larger class of predicates defined by clauses of constant locality i.*

This result improves on [4] in two ways: it is constructive and works under any distribution.

Of course by the result of Cohen, both theorems hold for determinate clauses of constant depth as well. However in that case it could be more efficient to use directly the transformation in [6] instead of rewriting a determinate clause as a local one.

5.4 Dealing with recursion

In [6] the authors point out that the general technique we have been using for non-recursive clauses works as well in the recursive case provided we allow the learner to ask questions about the target predicate (and we bound the arity of the target predicate by a as well). In the language of Angluin [1] we would say that the results hold if the learner is allowed *membership* and *disjointness* queries about the target predicate. In the logic programming community disjointness queries are known as *existential* queries. We will use this term since in this context is more intuitive.

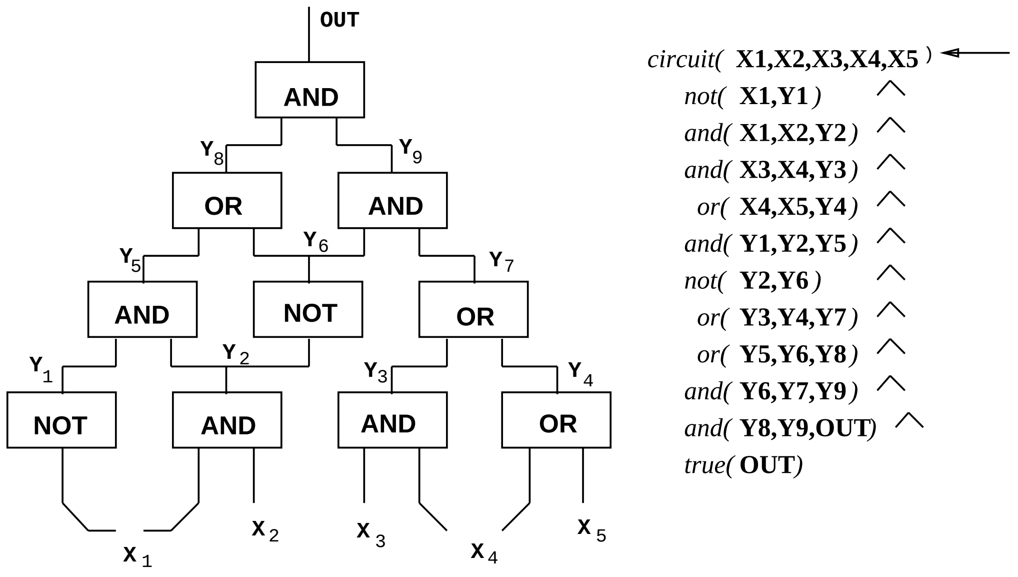
A membership query is a question of the kind "Is $q(c_1, \ldots, c_a)$ true or false?" An existential query is a question like "Which instantiations of the unknown variables make $q(c_1, \ldots, c_i, X_{i+1}, \ldots, X_a)$ true?" (actually the query can be more general than this with any combination of constant and variables as argument of the predicate).

The line of reasoning is as follows: when building the list of features F we will use the target predicate as any of the background predicates. However when we query the background knowledge to know about the truth status of $q(d_1, \ldots, d_a)$ we may not find this value among the examples provided to us so we have to ask a membership query. For the local clauses case we need existential queries as well since the variables are not determinate.

This observation applies of course to our stronger results in Section 5.4 as well. So Theorems 3 and 4 hold for the recursive case as well if we allow membership and existential queries (and assume $n \leq a$).

5.5 Limits on PAC-learning logic programs

An entire section of Cohen's paper [4] is devoted to prove negative results for restricted classes of logic programs. He proves that relaxing in various ways the assumptions we made



```
circuit( X1,X2,X3,X4,X5 ) ←
   not( X1,Y1)        ∧
   and( X1,X2,Y2)     ∧
   and( X3,X4,Y3)     ∧
    or( X4,X5,Y4)     ∧
   and( Y1,Y2,Y5)     ∧
   not( Y2,Y6)        ∧
    or( Y3,Y4,Y7)     ∧
    or( Y5,Y6,Y8)     ∧
   and( Y6,Y7,Y9)     ∧
   and( Y8,Y9,OUT)    ∧
   true( OUT)
```

Figure 1: How to reduce a circuit to a determinate clause

above, results in unlearnabilty. These results are obtained using the framework of prediction preserving reducibility devised by Pitt and Warmuth in [22]. Typically the reduction is conducted over a particular background knowledge $\mathcal{B}$.

Log-depth clauses: The first class that he considers is the one of determinate clauses with depth bounded by $\log n$. The proof of unpredictability comes from a reduction of Boolean circuits of depth i to determinate clauses of depth i. Using a result by Kearns and Valiant[12] (they prove log-depth circuits are hard to predict under suitable cryptographic assumptions), we know that log-depth clauses are hard to predict as well. Actually this result can be strenghtned using a recent result by Kharitonov [13], stating that log-depth circuits are hard to learn even under the uniform distribution. The reduction uses a database of only eleven facts (the definitions of the boolean predicates *and, or, not, true*) and it is shown in Figure 1.

Constant-depth indeterminate clauses: This class is not predictable because the corresponding language includes an **NP**-complete problem. Schapire [26] proved that this is a sufficient condition for unpredictability. For example if $\mathcal{B}$ defines a predicate $node(G, X)$ to be true if X is a node of G and similarly $edge(G, X, Y)$ if there is an edge from X to Y, then the predicate

$$clique_k(G) \leftarrow (\bigwedge_{i=1}^{k} node(G, X_i)) \wedge (\bigwedge_{i \neq j} edge(G, X_i, Y_j))$$

is true if and only if the graph G has a clique of size k. Notice that the depth of the above clause is only 1.

Clauses with k free variables: Learning a predicate defined by a single clause with k free variables is reduced to the problem of learning Boolean DNF formulae. The reduction is based on the fact that there can be at most $l(n+k)^a$ literals in a clause with k free variables and at most $(ab)^k$ substitutions for the free variables. So let v_{ij} be a variable with $i = 1, \ldots, (n+k)^a$ and $j = 1, \ldots, (ab)^k$ representing every possible literal with every possible substitution. So a clause $C \leftarrow B_1 \ldots B_s$ is mapped to the DNF formula

$$\phi_C = \bigvee_{j=1}^{(ab)^k} \bigwedge_{i=1}^{s} v_{ij}$$

The status of the problem of PAC-learning DNF formulae is open.

6 The problem of imperfect data

Considering the potential practical applications of this problem, it is surprising how little work has been done on the effects of data imperfections over the learning process. An empirical study of the performance of various noise-handling heuristics is presented in [15].

In a typical ILP problem, the most common forms of data imperfections are the following:

1. errors in the training examples, caused by an erroneous classifications of facts about the target predicate.

2. errors in the background knowledge, caused by erroneous classification of facts about the auxiliary predicates introduced in it.

3. incomplete background knowledge, i.e missing facts about the auxiliary predicates.

If we go back to the protein structure example in Section 3, an example of (1) would be that the Brookhaven database would tell us that in protein 155C the residue in position 25 forms an α-helix while in reality that residue is on a β-strand. An example of (2) would be that we may have classified a particular aminoacid as aromatic while instead it is not the case. Finally an example of (3) would be that when asked, we can't find in our database if a particular aminoacid is aromatic or not.

Implemented machine learning systems have a general criterion for dealing with errors. It usually involves methods to avoid the output hypothesis from *over-fitting* the data, i.e. to prevent concept descriptions that are too specific. In GOLEM the system allows the output hypothesis to cover some of the negative examples. In [15] the authors describe the noise-handling mechanisms of two other ILP systems: LINUS and FOIL. LINUS transforms the ILP problem into a attribute-value problem that is learned as a decision tree. Overfitting avoidance is then obtained by various tree pruning heuristics. FOIL instead uses an *encoding length restriction* i.e. does not allow output clauses whose encoding is longer of the encoding of the training examples. All mechanisms are just heuristics with little theoretical justification besides empirical results. Moreover recently in the literature there have been various criticisms on the general validity of overfitting avoidance heuristics (see for example [25])

In this section we will expand our quest for solid theoretical grounds for ILP to the problem of learning in the presence of imperfect data. We saw in the previous section that

PAC-learning restricted classes of logic program can be transformed into learning over the propositional domain. We will show how this transforms the first two kind of errors in two well studied noise models for the "standard" PAC-learning literature. For the third kind of errors we will introduce a new noise model for the propositional domain that could be of its own interest.

6.1 Classification errors

Among the most common errors any learning system has to deal with is the one related to misclassifications of the training examples. A very reasonable assumption to make is to assume that with probability $\eta < \frac{1}{2}$ every example independently can have a wrong classification. We would say then that we are dealing with *random classification noise* and η is called the *error rate*. If we look back at the transformations outlined in Section 5 we can see how random classification noise in the ILP problem induces the same kind of noise in the corresponding propositional problem.

Recently Kearns in [10] introduced a very powerful technique to deal with random classification noise. Using so-called *statistical queries* he proves that a large class of propositional learning algorithms works in the presence of random classification noise with error rate $\eta < \frac{1}{2}$. This class contains the algorithm to learn conjunctions, the one to learn k-DNF and the Blum-Singh one to learn k-term DNF by general DNF. Those are the algorithms that we used in Section 5 to prove PAC-learnability of the corresponding classes of restricted logic programs. So as a corollary we get that Theorems 3 and 4 hold also in the presence of random classification noise with error rate $\eta < \frac{1}{2}$

Corollary 5 *The class of predicates defined by non-recursive clauses of constant locality i with at most k literals and the class of predicates defined by k non-recursive clauses of constant locality i are PAC-learnable under any distribution, in the presence of random classification noise with error rate $\eta < \frac{1}{2}$, by the larger class of predicates defined by clauses of constant locality i.*

6.2 Errors in the background knowledge

Now we turn to a different kind of data errors. We assume that some of the facts in our background knowledge are stated incorrectly. Going back to the transformations of Section 5 we can see that this problem induces a related *attribute noise* in the corresponding propositional problem.

The definition of attribute noise is as follows: suppose we are learning Boolean concepts over $\{0,1\}^n$, and let $e =< (c_1, \ldots, c_n), l_e >$ be a correctly labeled example. Then the learner will be presented with $e' =< (c'_1, \ldots, c'_n), l_e >$ where for some of the indices i $c'_i = \bar{c}_i$. The most studied version of this model is random attribute noise, where we assume that $c'_i = \bar{c}_i$ with probability $\nu < \frac{1}{2}$ independently for each single feature and example.

For this model we can use the results of Goldman and Sloan [8] that show how to learn conjunctions and Shackelford and Volper [27] that show how to learn k-DNF with random attribute noise. Nothing is known on learning k-term DNF with random attribute noise.

However we feel that modeling errors in the background knowledge with random attribute noise in the corresponding propositional problem is inappropriate. In fact this model could be acceptable if we think of our background knowledge to be errorless and then something happening during the query process that randomly flips the classification of our background query (like an imperfect communication channel between the learner and the database). Even if this is an eventuality to consider, what we are talking here is something different.

For example some of the background predicates could be more sensitive to errors than others. If we take the protein example we could imagine that the lab test for aminoacid aromaticity has a higher confidence rate than the one for aminoacid hydrophobia. This would in turn imply that different attributes have different noise rates ν_i in the corresponding propositional problem (remember that each attribute corresponds to a particular background predicate applied to some variables). But Goldman and Sloan prove in [8] that for this kind of attribute noise, even the simple task of learning conjunctions become impossible unless $\nu_i < 2\epsilon$ for all i where ϵ is the admissible error of the output hypothesis. So we can tolerate only very small amount of noise.

Moreover if some facts are stated wrong in the background knowledge, when queried they will *consistently* return wrong answers. The errors in the attributes are not really randomly distributed, but there is some determinism involved in the process. If we think of an adversary choosing the places where to flip the attribute bits, then we fall in the model considered by Kearns and Li [11]. And again they prove that only a $\frac{\epsilon}{1+\epsilon}$ fraction of errors can be tolerated.

Errors in the background knowledge appear then as a very hard problem to deal with.

6.3 Incomplete background knowledge

In this case we are assuming that some important facts are not present in the background knowledge $\mathcal{B}$. If we follow the logic programming approach and consider $\mathcal{B}$ as a Prolog program to query for some facts, then everything that is not provable from $\mathcal{B}$ is assumed to be false. So if some facts are missing we just assume that they do not hold. This assumption will reduce this case to the previous one of errors in $\mathcal{B}$. But since we failed in finding a satisfactory solution to this last problem we will try to consider another approach.

Given a batch of positive and negative examples of the target predicate we saw how to transform these in corresponding examples of a propositional concept. In particular to do so we have to query the background knowledge over a bunch of questions. We will think of $\mathcal{B}$ as of an oracle that will answer those questions by three possible answers: "true", "false", "don't know". We will assume that we will ask all the questions at once to $\mathcal{B}$ and so we may think of an adversary deciding on when to answer "don't know" to a question, with a complete full knowledge of the questions being asked. The only condition that we make on $\mathcal{B}$ is that every feature of the corresponding propositional problem must have at most a fraction $\rho < 1$ of unanswered questions. We will call ρ the *ignorance* of the background knowledge $\mathcal{B}$. Before proceeding let us notice that this a pretty realistic model of the incomplete background knowledge: it is deterministic and the only condition imposed is that for each predicate the facts that we know and those we don't know are in some fixed proportion.

Under these assumptions we reduce the problem of learning logic programs with incomplete background knowledge to the one of learning a boolean concept over $\{0,1\}^n$ in the presence of the following form of noise. Let $e_i =< (c_i^1, \ldots, c_i^n), l_i >$ for $i = 1, \ldots, m$ the examples drawn according to the target distribution, then the learner receives $\bar{e}_i =< (\bar{c}_i^1, \ldots, \bar{c}_i^n), l_i >$ where $\bar{c}_i^j = c_i^j$ or $\bar{c}_i^j =?$ The only condition is that for all $j = 1, \ldots, n$ the set $\{i : \bar{c}_i^j =?\}$ has cardinality less than ρm for $\rho < 1$. We will call this model the *fixed attribute noise with hidden rate* ρ.

Conjunctions: The standard algorithm for learning conjunctions can be modified to work under this kind of noise. Start with the hypothesis h containing a conjunction of all literals. For each positive example erases from the hypothesis the literals that make h false on that example. After m examples output the current hypothesis. Now let z be a literal that appears in h but not in the target conjunction c. Then z causes h to err on those positive examples in which $z = 0$. Let w_z be the weight of this set under the target probability distribution. If $w_z \leq \frac{\epsilon}{2n}$ then we don't care since the error due to those literals will never sum up to more than ϵ. What is then the probability of a *bad* literal (i.e. such that $w_z > \frac{\epsilon}{2n}$) to appear in h? For a particular bad literal z the probability that it survives m examples is less than $(1 - \frac{\epsilon}{2n})^{m-d_z}$ where d_z is the number of examples in which the bit corresponding to z was deleted. Since $d_z < \rho m$ for any z we have that the probability that some bad literal is still in h after m examples is

$$2n(1 - \frac{\epsilon}{2n})^{m(1-\rho)} \leq 2ne^{-m(1-\rho)\frac{\epsilon}{2n}} < \delta$$

if

$$m > \frac{2n}{\epsilon(1-\rho)} \log \frac{2n}{\delta}$$

So we can PAC-learn conjunctions with just an increase of a factor of $(1-\rho)^{-1}$ in the sample complexity.

k-DNF: Remember that learning k-DNF is performed by learning a disjunction over $\{0,1\}^{n^k}$ where each variable represents a possible conjunction over $x_1, \ldots, x_n$. Learning disjunctions is the dual of the algorithm above (just consider negative examples instead of positive). To have an hidden rate $\rho < 1$ in the disjunction learning problem we need that the original hidden rate to be less than $\frac{1}{k}$. In fact a single bit set to ? can set to ? the entire conjunction. It is not hard to come up with an example in which if the original rate is $> \frac{1}{k}$ then in the disjunction problem we are going to have $\rho = 1$ i.e. for some bit we see *no* information at all.

k-term DNF: A reasoning similar to the one above applies to the algorithm of Blum and Singh [3] to learn k-term DNF by general DNF formulae. It can be shown that restricting the hidden rate to be $\rho < \frac{1}{k}$ we can PAC-learn with only a constant factor increase in the sample complexity ($(1 - k\rho)^{-1}$ to be precise)

Notice that in the algorithms above, we need to use the original 2-buttons model of Valiant [28] i.e. we need to be able to ask for a positive or a negative example. Consider the algorithm for learning conjunctions. There we use only positive examples. Since the hidden bits are placed maliciously by an adversary, he can concentrate them on the positive examples, so while the hidden rate is still $\rho < 1$, since we discard negative examples the practical hidden rate for us could easily be equal to 1. Similarly we could define a *weak* version of the fixed attribute noise model in which the rate of hidden bits is ρ separately for both positive and negative examples.

So we can state the following:

Theorem 6 *Under the fixed attribute noise 2-button model (or the weak 1-button version of it) with hidden rate ρ we can learn the following Boolean concept classes*

- *conjunctions, for all $\rho < 1$*
- *k-DNF if $\rho < \frac{1}{k}$*
- *k-term DNF if $\rho < \frac{1}{k}$*

The increase in sample complexity is linear in $(1-\rho)^{-1}$.

If we remember that:

- learning a predicate defined by a single local clause corresponds to the propositional problem of learning a conjunction.
- learning a predicate defined by local clauses with at most k literals corresponds to the propositional problem of learning a k-DNF formula.
- learning a predicate defined by k local clauses corresponds to the propositional problem of learning a k-term DNF formula.

we get the following as a corollary:

Corollary 7 *The following classes of logic programs are PAC-learnable with incomplete background knowledge of ignorance ρ:*

- *predicates defined by a single local clause for any $\rho < 1$*
- *predicates defined by local clauses with at most k literals for any $\rho < \frac{1}{k}$*
- *predicates defined by k local clauses for any $\rho < \frac{1}{k}$*

The easy way out: It may still happen that after translating all the m examples in the propositional form, some features can have a higher hidden rate than the allowed ρ. The easy way out of this problem is to allow interaction between the program and the user. The program would stop, query the user about some facts it could not find in the database (probably ask the questions that will reduce the number of hidden bits around) and when all features have an acceptable hidden rate apply the algorithms described above. The interesting part of this approach would be that the program finds out by itself what are the relevant missing information to perform the learning task.

7 Final Remarks

In this paper we have discussed a new emerging area of research: *Inductive Logic Programming.* This area is concerned with the inductive inference of Prolog programs from examples of their behavior and a general background knowledge.

We have surveyed the following results: Muggleton and Feng's approach to learn Prolog clauses by relative least general generalization [19]; Cohen's non-constructive results on PAC-learning local Prolog clauses under broad classes of distributions [4].

We have improved some of these results by giving algorithms that PAC-learn local Prolog clauses under any distribution. Moreover we presented the first (up to our knowledge) theoretical framework for Inductive Logic Programming in the presence of imperfect data.

This investigation leaves many questions open for further research. Let us address some of them:

Open Problem 1 We have discussed only the inference of a single predicate. It would be very interesting to address the issue of inferring large logic programs composed by definitions of multiple predicates.

Open Problem 2 In our treatment of data imperfections we did not mention the possibility of having *inappropriate* background knowledge. By that we mean a background knowledge that introduces facts and predicates that are not relevant to the learning problem at hand. This appears to be a very serious problem. Maybe some of the work done in *Machine Predicate Invention* (see [18]) could be useful in this case.

Open Problem 3 In spite of the progress done in this paper, the question about learning subclasses of Prolog clauses is far from being settled. There are still various interesting classes of Prolog clauses that are not known to be PAC-learnable or to be hard to PAC-learn. For many of them the transformation to the propositional domain yields a DNF learning problem. Maybe some recent results about PAC-learning DNF (see [2]) could be useful in this setting.

Open Problem 4 Finally lot has to be done regarding the new noise model we introduced. For example

- We have just analyzed the learning algorithms (conjunctions, k-DNF, k-term DNF) that arise in the transformation of restricted ILP problems to the propositional domain. What happens to other learning algorithms under the fixed attribute noise model?
- We introduced three variations of the fixed attribute noise model: 2-button, 1-button, and weak model. Are they equivalent? Or are they separable?
- The tolerance of higher hidden bits rates can improve if we assume a particular distribution over the examples. Consider the case of k-DNF, we said that ρ must be less than $\frac{1}{k}$. But if we assume that the examples are distributed uniformly, maybe with high probability we can tolerate a higher hidden rate, since a hidden bit will hide the entire conjunction if and only if the other bits are all 1's. So the question is: in the fixed attribute noise model are distribution-dependent improvements possible?

Acknowledgments

This paper is a revised version of my Area Exam paper. I would like to thank the chair of the committee David McAllester for suggesting such an interesting topic and the other two members of the committee, Ron Rivest and Mike Sipser, for their comments and help.

I would like to acknowledge the help of the following people: Mona Singh explained her paper [3] to me, making my reading of it easier and faster; Donna Slonim suggested the "easy way out" in Section 6.3; Lisa Tucker-Kellogg explained to me a lot of stuff about proteins.

References

[1] D. Angluin, *Queries and Concept Learning*, Machine Learning 2, 1988.

[2] A. Blum, M. Furst, J. Jackson, M. Kearns, Y. Mansour, S. Rudich, *Weakly learning DNF*, to appear in STOC 1994

[3] A. Blum, M. Singh, *Learning functions of k terms*, COLT 1990

[4] W.W. Cohen, *PAC-Learning Non-Recursive Logic Programs*, preprint, 1993.

[5] B. Dolsak, S. Muggleton, *The application of Inductive Logic Programming to finite element design*, in S. Muggleton ed., *Inductive Logic Programming*, Academic Press 1992.

[6] S. Dzeroski, S. Muggleton, S. Russell, *PAC-learnability of Determinate Logic Programs*, COLT 1992.

[7] S. Dzeroski, S. Muggleton, S. Russell, *Learnability of Constrained Logic Programs*, ECML 1993.

[8] S.A. Goldman, R.H. Sloan, *Can PAC-learning Algorithms Tolerate Random Attribute Noise?*, Tech. Rep. Wash. Univ. WUCS-92-25, 1992

[9] D. Haussler, *Learning conjunctive concepts in structural domains*, Machine Learning, 5 (2), 1990.

[10] M. Kearns, *Efficient Noise-Tolerant Learning from Statistical Queries*, STOC 1993.

[11] M. Kearns, M. Li, *Learning in the Presence of Malicious Errors*, STOC 1988

[12] M. Kearns, L. Valiant, *Cryptographic limitations on learning Boolean formulae and finite automata*, STOC 1989.

[13] M. Kharitonov, *Cryptographic lower bounds on the learnability of Boolean functions on the uniform distributions*, COLT 1992

[14] R. King, S. Muggleton, R. Lewis, M. Sternberg, *Drug design by machine learning*, Proceedings of the National Academy of Science, 89 (23), 1992.

[15] N. Lavrac, S. Dzeroski, *Inductive Learning of Relations from Noisy Examples*, in S. Muggleton ed., *Inductive Logic Programming*, Academic Press 1992.

[16] M. Li, P. Vitanyi, *Learning simple concepts under simple distributions*, SIAM J. of Comp. 20 (5), 1991.

[17] J.W. Lloyd, *Foundations of Logic Programming*, Springer-Verlag, 1987.

[18] S. Muggleton, W. Buntine, *Machine Invention of First-Order Predicate by Inverting Resolution*, in S. Muggleton ed., *Inductive Logic Programming*, Academic Press 1992.

[19] S. Muggleton, C. Feng, *Efficient induction of logic programs*, in S. Muggleton ed., *Inductive Logic Programming*, Academic Press 1992.

[20] S. Muggleton, R. King, M. Sternberg, *Protein secondary structure prediction using logic-based machine learning*, Protein Engineering, 5 (7), 1992.

[21] L. Pitt, L. Valiant, *Computational limitations on learning from examples*, JACM 35 (4), 1988

[22] L. Pitt, M. Warmuth, *Prediction Preserving Reducilibity*, JCSS 41 (3), 1990

[23] G.D. Plotkin, *Automatic Methods of Inductive Inference*, Ph.D. thesis, Edinburgh University, 1971.

[24] J.A. Robinson, *A machine-oriented logic based on the resolution principle*, JACM 12 (1), 1965.

[25] C. Schaffer, *Overfitting Avoidance as Bias*, Machine Learning 10, 1993.

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

[27] G. Shackelford, D. Volper, *Learning k-DNF with Noise in the Attribute*, COLT 1988

[28] L. Valiant, *A Theory of the Learnable*, Comm of the ACM, 27 (11), 1984