INFORMATION AND COMPUTATION 114, 88–114 (1994)

A Formal Model of Hierarchical Concept Learning

RONALD L. RIVEST*

Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, Massachusetts 02139

AND

ROBERT SLOAN†

Department of Electrical Engineering and Computer Science, University of Illinois at Chicago, Chicago, Illinois 60610

We show how to learn from examples (Valiant style) any concept representable as a boolean function or circuit, with the help of a teacher who breaks the concept into subconcepts and teaches one subconcept per lesson. Each subconcept corresponds to a gate in the boolean circuit. The learner learns each subconcept from examples which have been randomly drawn according to an arbitrary probability distribution, and labeled as positive or negative instances of the subconcept by the teacher. The learning procedure runs in time polynomial in the size of the circuit. The learner outputs not the unknown boolean circuit, but rather a program that, for any input, either produces the same answer as the unknown boolean circuit, or else says "I don't know." Thus the output of this learning procedure is *reliable*. Furthermore, with high probability the output program is nearly always *useful* in that it says "I don't know" very rarely. A key technique is to maintain a hierarchy of explicit "version spaces." Our main contribution is thus a learning procedure whose output is reliable and nearly always useful; this has not been previously accomplished within Valiant's model of learnability. © 1994 Academic Press, Inc.

1. INTRODUCTION

The field of inductive inference has been greatly broadened by Valiant's seminal paper [23] on "probably approximately correct" identification. He gave an excellent definition of what it means to learn—in a reasonable amount of time—a concept (for instance, a boolean function) from

* This paper was prepared with support from NSF Grant DCR-8607494, ARO Grant DAAL 03-86-K-0171, and the Siemens Corporation. E-mail address: rivest@theory.lcs.mit.edu.

† Supported by an NSF graduate fellowship, by the Siemens Corporation, and by ARO Grant DAAL 03-86-K-0171. E-mail address: sloan@eecs.uic.edu.

0890-5401/94 $6.00

Copyright © 1994 by Academic Press, Inc.
All rights of reproduction in any form reserved.

examples. Moreover, in that paper, and in a number of subsequent papers (e.g., [6, 10, 18]), algorithms were given showing how to efficiently learn various different concept classes.

Thus the good news is that we now have one crisp definition of concept learning, and a number of algorithms for efficiently learning various classes of concepts. The bad news is that Valiant presents strong evidence [23] that learning arbitrary polynomial-size circuits is computationally intractable, and Pitt and Valiant [18] show that learning certain particular interesting classes of boolean functions, such as boolean threshold formulas, is NP-complete, at least if we restrict the learner's hypothesis set. Recently there have been more results on concept classes that we cannot learn within the pac-learning model (given various cryptographic assumptions generally believed to be true) [12, 19].

The largest concept class that an optimist might reasonably hope to be able to learn efficiently in some model of concept learning is the set of all concepts that can be represented by polynomial-size circuits. For any larger class, just deciding whether a given instance is a positive or negative instance of a particular concept may be computationally intractable. The infeasibility of learning the class of concepts defined by polynomial-size circuits in the pac-learning model (given certain cryptographic assumptions) was shown in Valiant's original pac-learning paper [23] using cryptographic tools from Goldreich *et al.* [5].

In this paper we examine a model of learning with a more powerful teacher than the teacher of the pac-learning model, thus allowing us to learn this "largest reasonable" concept class.

1.1. *Hierarchical Learning*

The way we escape the infeasibility of learning arbitrary concepts is by first learning relevant subconcepts of the target concept, and then learning the target concept itself.

Learning by first learning relevant subconcepts has been a useful technique elsewhere in the field of learning:

- Cognitive psychologists believe that one way humans learn is by first organizing simple knowledge into "chunks," and then using these chunks as subconcepts in later learning [16].

- In the artificial intelligence community, the builders of the *Soar* computer learning system have built a system that saves useful "chunks" of knowledge acquired in the current learning task for use as subconcepts in furture learning tasks [14, 15]. Also, the SIERRA system learns how to do arithmetic in a manner broadly similar to what we suggest; it learns "one subprocedure per lesson" [24].

• Within the framework of theoretical inductive inference, Angluin *et al.* [2] recently showed how to learn certain otherwise unlearnable recursive functions by first learning relevant subconcepts.

1.2. *Review of Valiant Model*

Before we discuss our results, we first give a brief review of Valiant's learnability model. For a more lengthy discussion of the model and recent results obtained using it, we refer the reader to the excellent survey article [11].

We will say that an algorithm learns from examples if it can, in a feasible (polynomial) amount of time, find (with high probability) a rule that is highly accurate. Now we must define what we mean by such terms as "find a rule," "with high probability," and "highly accurate."

Fix an instance space X. Formally a *concept* c for instance space X is some subset of X. If instance $x \in X$ is contained in concept c, then we say that x is a *positive instance* of c; otherwise we say that x is a *negative instance* of concept c. (We are slightly sloppy throughout and refer to concepts interchangeably both as subsets of the instance space and as $\{0, 1\}$-valued functions defined on the instance space. In particular, we will write $c(x) = 1$ if x is a positive instance, and $c(x) = 0$ if x is a negative instance.)

Let $\mathscr{C}$ be a set of concepts, or *concept class*, over X. Formally, $\mathscr{C} \subseteq 2^X$. If X is the set of inhabitants of the U.S., concepts would include both the rather simple concept *males*, and the doubtless more complicated concept, *people whose marginal federal income tax rate is 31%*. An example of a concept class would be TAX BRACKETS, which would include the concepts *people paying no income tax* and *people whose marginal tax rate is 31%*.

We call the concept $c_* \in \mathscr{C}$ that our algorithm is trying to learn the *target concept*. The target concept may be any concept whatsoever in the concept class. We think of it as being arbitrarily chosen by some outside teacher or supervisor.

The *length* of concept c, denoted $|c|$, is the number of bits it takes to write down c in some agreed-upon encoding scheme. For example, if our instance space is $\{0, 1\}^n$, possible representations for concepts include truth tables, boolean formulas, and boolean circuits. (Haussler [7] gives a good discussion of issues concerning coice of representation of concepts.) The length of an instance is defined similarly. Since we want to analyze the computational efficiency of learning algorithms, we must analyze their asymptotic resource consumption. Therefore we will parameterize our instance space, writing $X = \bigcup_{n=1}^{\infty} X_n$, where all instances in X_n have length bounded by some polynomial function of n. Any one concept $c \in \mathscr{C}$ is required to be completely contained in X_n for some n, and we define $\mathscr{C}_n$

to be $\{c \in \mathscr{C} \mid c \subseteq X_n\}$. In this paper we will assume that for $c \in \mathscr{C}_n$, $|c|$ is bounded by a polynomial in n, although this restriction is not necessary.

We assume that our learning algorithm has available to it a black box called EXAMPLES, and that each call to the black box returns a labeled example, $(x, c_*(x))$, where $x \in X$ is an instance, and c_* is the target concept. Furthermore, the EXAMPLES box generates the instances x according to some fixed but arbitrary probability distriution **P** on X. We make no assumptions whatsoever about the nature of **P**, and our learner is not told what **P** is.

First we define formally what it means for a concept to be an accurate approximation of target concept c_*, and then we define pac learning itself.

DEFINITION. Fix an instance space X and a probability distribution **P** on X. We say that concept c_1 is an *ε-approximation* of concept c_0 if and only if

$$\sum_{c_1(x) \neq c_0(x)} \mathbf{P}(x) \leqslant \varepsilon. \tag{1}$$

DEFINITION. Let $\mathscr{C}$ be a class of concepts on domain $X = \bigcup_{n=1}^{\infty} X_n$. Algorithm A *probably approximately correctly learns (pac learns)* $\mathscr{C}$ if and only if for every positive n, for every $c \in \mathscr{C}_n$, for every probability distribution on X, and for every positive ε and δ, Algorithm A, given only ε, δ, and access to EXAMPLES(c), meets the following two criteria.

Learning Criterion. Algorithm A outputs some representation of a concept c' such that

$$\Pr[c' \text{ is an } \varepsilon\text{-approximation of } c] \geqslant 1 - \delta, \tag{2}$$

where the probability is taken over the output of EXAMPLES and any coin tosses A may make.

Efficiency Criterion. The running time of A is bounded by some polynomial function of $1/\varepsilon$, $1/\delta$, and n.

We say that a concept class $\mathscr{C}$ is *pac learnable* if there exists some algorithm that pac learns $\mathscr{C}$.

Discussion of the Definition. Intuitively, we are saying that the learner is supposed to do the following:

1. Ask Nature for a random set of examples of the target concept.
2. Run in polynomial time.
3. Output a formula that with high probability agrees with the target concept on a randomly chosen instance.

We think of Nature as providing examples to the learner according to the (unknown) probability that the examples occur in Nature. Though the learner does not know this probability distribution, he does know that his formula needs to closely approximate the target concept only with respect to this probability distribution.

Intuitively, there may be some extremely bizarre but low probability examples that occur in Nature, and it would be unreasonable to demand that the learner's output formula classify them correctly. Hence we require only approximate correctness. Moreover, with some very low but nonzero probability, the examples the learner receives from Nature might all be really bizarre. Therefore we cannot require the learner to *always* output an approximately correct formula; we only require that the learner do so with high probability.

1.3. *A Variation on the Valiant Model*

We introduce here a new definition of learning which is very similar to but more stringent than pac learning. In pac learning, the learner is required to output a description of some subset of the instance space (the learner's "best guess" at the positive instances) in some fixed representation, such as a boolean circuit or a formula in DNF. Our learner is instead supposed to give a (polynomial time) program taking instances as input, and having three possible outputs: 1, 0, and "I don't know."

DEFINITION. We call learning Algorithm A *reliable* if the program output by A says 1 only on positive instances, and says 0 only on negative instances of the target concept.

Of course, given that definition of reliable, it is very easy to design a reliable learning algorithm: Have the learning algorithm look at no examples, and output the program which just gives the useless answer "I don't know" on all instances. Informally we call a learning algorithm useful if the program it outputs says "I don't know" on at most a fraction ε of all instances, where ε is an input to the learning algorithm. Formally, we make the following definition, analogous to pac learning:

DEFINITION. *Let* $\mathscr{C}$ be a class of concepts on domain $X = \bigcup_{n=1}^{\infty} X_n$. We say Algorithm A *reliably and probably usefully learns* $\mathscr{C}$ if and only if for every positive n, for every $c \in \mathscr{C}_n$, for every probability distribution $\mathbf{P}$ on X, and for every positive ε and δ, Algorithm A, given only ε, δ and access to EXAMPLES(c), halts in time polynomial in n, $1/\varepsilon$, and $1/\delta$, and outputs a program Q that satisfies the following conditions.

1. For every $x \in X_n$, either $Q(x) =$ "I don't know" or $Q(x) = c_*(x)$. (A is reliable.)

2. With probability at least $1-\delta$,

$$\sum_{Q(x)=\text{I don't know}} \mathbf{P}(x) < \varepsilon.$$

(A is probably useful.)

The above definition is similar to the definition of pac learning, in that both definitions require the learner to find some concept that probably agrees with the target concept with high probability. Our new definition is stronger than pac learning in that we require, in addition, that the output of learner must *never* misclassify an instance. It must somehow "know enough" to say "I don't know," rather than to misclassify.

In this paper we present an algorithm that reliably and probably usefully learns the concept class of all functions that can be represented by polynomial-size circuits.

2. How to Learn: Sketch

The original definition of pac learning has many desirable features. Not least among them is that efficient algorithms for pac learning a number of interesting concept classes are now known. Of course, we do not always know a priori that the concept we want to learn is going to be in 3CNF or 7DNF or some other given class. We would like to have an algorithm that can pac learn regardless of what class the target concept is drawn from. More precisely, we would like to have an algorithm that could pac learn the class of all boolean functions that can be represented by polynomial-size boolean circuits (which includes any function that can be represented by a polynomial-size boolean formula).

Definition. Let f be a boolean function defined on $\{0,1\}^n$. We say that boolean circuit C computes f if $f(x)=1$ just in case $C(x)=1$. The *circuit size complexity* of f is defined to be the size of the smallest circuit computing f. For a boolean function g with domain $\{0,1\}^*$, the *circuit size complexity* of g is a function $\mathcal{G}: \mathbf{N} \to \mathbf{N}$ such that $\mathcal{G}(n)=$ circuit size complexity of g_n (where g_n is g restricted to the domain $\{0,1\}^n$).

Definition. A boolean function is a member of the class PC if its circuit size complexity is bounded above by some polynomial.

Unfortunately, the goal of pac learning the class PC is unlikely to be attainable. As Valiant explains [23], assuming that one-way functions exist (an assumption which we feel is likely to be correct), the class PC is not pac learnable.

Thus we are driven to look for some way of learning arbitrary boolean circuits. Our solution is to learn in a hierarchical manner. First we pac learn some important subconcepts of the target concept, and then we pac learn the final concept as a function of these subconcepts.

To be more precise, our method is as follows. We learn our first subconcept knowing that it must be some simple boolean function of the instance attributes. We learn each following subconcept knowing that it must be some simple boolean function of the instance attributes *and previously learned subconcepts.* Ultimately we learn the original target concept as some simple boolean function of the instance attributes and all of the previously learned subconcepts.

Consider, for instance, the concept of one's *dependents*, as defined by the IRS:[1]

$$\begin{aligned} \textit{dependent} = (&\text{SupportFromMe} > 1/2) \wedge \neg \text{FiledJointReturn} \\ &\wedge [(\text{Income} < 1900 \wedge (\text{MyChild} \vee \text{MyParent})) \\ &\vee (\text{MyChild} \wedge (\text{Age} < 19 \vee \text{IsInCollege}))]. \end{aligned} \tag{3}$$

One can readily imagine such a complicated definition being too hard to learn from examples. On the other hand, if we first teach some simple subconcepts such as "MyChild $\vee$ MyParent," and "Age $< 19 \vee$ IsInCollege," and next teach some harder subconcepts as functions of those, and then finally the *dependent* concept as a function of all previously learned subconcepts, then the learning task becomes easier.

Moreover, because we break the target concept into very simple subconcepts, we can develop a learning protocol that has one very nice feature absent from ordinary pac learning—our learner *knows* when it is confused. (Formally, we achieve reliable and probably useful learning.) Continuing with the above example, we probably do not need to force our learner/taxpayer to learn the concept *dependent* perfectly. It is acceptable if the learner is unable to correctly classify certain unusual, very low probability instances such as, say, the case of "your underage great-great-great-granddaughter when all intervening generations are deceased." The probability of such an instance occurring is extremely low. Nevertheless, it would be desirable, if one ever did encounter such an "ever so great" grandchild, to be able to say, "I don't know if she is an instance of a dependent," rather than to misclassify her.

Our learner can, if desired, do precisely that—output a short fast progam taking instances as its input and having the three outputs, 1 (*dependent*), 0 (not a *dependent*), and "I don't know." This program is guaranteed to

[1] This is, in fact, a great oversimplification of the IRS definition.

Input variables: FiledJointReturn, SupportFromMe>1/2, MyChild, MyParent, Age<19, IsInCollege, Income<1900.
Output: *dependent* = y_7.

$$
\begin{aligned}
y_1 &= (\text{SupportFromMe} > 1/2) \wedge \neg\text{FiledJointReturn} \\
y_2 &= \text{MyChild} \vee \text{MyParent} \\
y_3 &= \text{Age} < 19 \vee \text{IsInCollege} \\
y_4 &= \text{Income} < 1900 \wedge y_2 \\
y_5 &= \text{MyChild} \wedge y_3 \\
y_6 &= y_4 \vee y_5 \\
y_7 &= y_1 \wedge y_6
\end{aligned}
$$

FIG. 1. A straight line program for *dependent*.

be correct whenever it gives a 1 or 0 classification, and moreover, with probability at least $1-\delta$ it says "I don't know" for at most a fraction ε of all instances. In short, it meets our definition of reliable and probably useful learning.

2.1. *Notation*

Before showing how to break our target concept, c_*, into pieces, we must first specify the problem more precisely. We assume that c_* is represented as a straight-line program of size s. Let the inputs to c_* be $x_1, \ldots, x_n$ and call the output y_s. The ith line of the program for c_*, for $1 \leqslant i \leqslant s$ is of the form

$$y_i = z_{i,1} \circ z_{i,2}, \tag{4}$$

where $\circ$ is one of the two boolean operators $\vee$ and $\wedge$, and every $z_{i,k}$ is either an input literal x_j or its complement $\bar{x}_j$ for some j, $1 \leqslant j \leqslant n$, or else y_j or $\bar{y}_j$ for some previously computed y_j (i.e., $j < i$).[2]

We will use EB(L) (standing for "Easy Boolean") to denote the set of all boolean formula that are the conjuction or disjunction of two variables, either or both of which may be negated, chosen from the set L. We will often write simply EB when the set L is clear from context.

Figure 1 shows a straight line program for the *dependent* concept defined in Eq. (3) above. The expression for every y_i comes from the class EB(V_i), where V_i is defined to be the set of variables $\{x_1, \ldots, x_n, y_1, \ldots, y_{i-1}\}$.

2.2. *An Easy but Trivial Way to Learn*

As a first attempt to develop a protocol for learning an arbitrary target concept c_* piece by piece, we might try the following: Have the teacher

[2] Note that straight line programs are equivalent to topologically sorted circuits, with lines being equivalent to the gates of the circuit.

supply not only examples, but also the pieces—the values y_i of all the relevant subconcepts. In particular, let the y_i be rearranged in some arbitrary order, $y_{j_1}, ..., y_{j_s}$. Now, each time the learner requests an example, he gets more than just a labeled example drawn according to **P**. The learner receives $x_1 \cdots x_n \# y_{j_1}, ..., y_{j_s}$ (and its label). Given all this help, the learning algorithm can easily determine which of the other variables a given variable depends on, and so can easily learn c_*.

This solution is not very satisfying, however, since it requires that the learner receive a large amount of "extra help" with each example. In essence, every time our learner is given an example while learning *dependent*, he would have to be told whether it is a child in college, whether it is a relative, and so on. Instead, our approach is to first teach the learner about y_1 for a while, assume the learner has learned y_1, then move on to y_2, never to return to y_1, and so on, y_i by y_i.

2.3. *High Level View of Our Solution*

The learning proceeds as follows: As in regular pac learning, there is one fixed probability distribution, **P**, on examples throughout; the teacher is *not* allowed to help the student by altering the probability distribution.

There are s rounds. The teacher moves from round i to round $i+1$ when the learner requests that he do so. In round i, the learner learns an approximation to y_i.

When our learner requests an example during round i, the teacher gives the learner a pair, $(x_1 \cdots x_n, l)$, where $x_1 \cdots x_n$ is drawn according to **P**, and l specifies whether $x_1 \cdots x_n$ is a positive or negative instance of y_i. In other words, in round i, l gives the truth value of $y_i(x_1 \cdots x_n)$ (rather than the truth value of $c_*(x_1 \cdots x_n)$).

During each round i, the learner tries to (ε', δ')-learn y_i where $\varepsilon' = \varepsilon / p_1(n, s)$, $\delta' = \delta p_2(n, s)$, and p_1 and p_2 are polynomials to be specified below. This learning task at first glance appears to be extremely simple, because y_i must be a simple conjunction or disjunction of two variables from the set V_i (and perhaps their negations). The catch is that while the learner gets the true values for $x_1 \cdots x_n$ he only gets his computed values for $y_1, ..., y_{i-1}$.

For instance, it might be that the true formula for y_3 is $y_1 \wedge y_2$. However, the values of y_1 and y_2 are not inputs to the learning algorithm. Suppose the learner has pac learned formulas $\hat{y}_1$ and $\hat{y}_2$ for y_1 and y_2. It may well be that the learner calls EXAMPLES and gets back a particular $x_1 \cdots x_n$ and the information that $y_3(x_1 \cdots x_n)$ is true, but both $\hat{y}_1(x_1 \cdots x_n)$ and $\hat{y}_2(x_1 \cdots x_n)$ are false even though $y_1(x_1 \cdots x_n)$ and $y_2(x_1 \cdots x_n)$ are both true.

Our job is to show how this learning can be done in such a manner that at the end, when we have some method of computing y_s in terms of $x_1 \cdots x_n$,

the value computed for $y_s(x_1 \cdots x_n)$ ε-approximates the target concept with probability at least $1-\delta$.

In fact, as we said above, we do something stronger. Our learner not merely pac learns, but reliably and probably usefully learns.

A Key Technique. The key technique we use is to have the learner maintain a list of all possible candidates for a given y_i. For each subconcept y_i we explicitly maintain the entire version space [17] for y_i, as a list F_i of possible candidate functions. The list F_i will be constructed in such a way that it always contains the true formula for y_i.

The reason the learner can maintain this list is that the set of all the possible candidates for any particular y_i is of polynomial size. Recall that the target function c_* is specified by a staight line program. Let K be the total number of possible distinct lines of the form $z_{i,1} \circ z_{i,2}$,

$$K = 8\binom{n+s-1}{2} + 2. \tag{5}$$

(There are $n+s-1$ choices for the two literals in the last line, either of which may be negated. Also, $\circ$ can take on two values. The additive factor of 2 comes from the additional possibilities *true* and *false*.) The important thing to notice is that K is polynomial in n and s, the size of (the representation of) the target concept.

Remark. If instead of EB we use some other polynomial-size concept class, we get a different value for the constant K specified in equation (5), but our geneal technique still works.

We exploit the technique of maintaining the list F_i of all the possibilities for y_i as follows. We say that an example, $(x_1 \cdots x_n, l)$, is "good" if for every previously learned y_j, $1 \leqslant j < i$, all the formulas in the list F_j for y_j take on the same truth value on $x_1 \cdots x_n$. Our algorithm has three fundamental parts:

1. In round i we get various examples of y_i. Since one of the formulas in the list F_j is the correct one, in every good example all the y_j's are computed correctly. We begin by filtering our examples to obtain a set of good examples.

2. Given good examples, we can be certain of the values of the y_j's, so we can proceed to learn y_i as a function of the attributes $x_1 \cdots x_n$, $y_1, \ldots, y_{i-1}$.

3. Finally, we need to specify the algorithm that we output at the end of round s.

3. Detailed Specification of our Learning Protocol

We assume to begin with that the learner is given the size s of the straight line program. This assumption maikes the presentation simpler and clearer. We show later that the learner need not be given s.

3.1. *Learning* y_i

During round i, the learner attempts to learn y_i as a function of the input literals and previous y_j and $\bar{y}_j$. The formula for y_i is in the class $\mathrm{EB}(V_i)$, and there are at most K candidates for the formula for y_i.

Given access to labeled examples, it is easy to (ε, δ)-pac learn any concept y_i that is one of at most K concepts by using the method of Blumer *et al.* [4]:

Theorem 1. *Let $\mathscr{C}$ be a finite concept class. For any $0 \leqslant \varepsilon, \delta \leqslant 1$, let*

$$m = \frac{1}{\varepsilon}\left(\ln |\mathscr{C}| + \ln\left(\frac{1}{\delta}\right)\right). \tag{6}$$

For any target concept $c_* \in \mathscr{C}$ and probability distribution $\mathbf{P}$ on the domain, with probability at least $1-\delta$ any concept $c \in \mathscr{C}$ that is consistent with m labeled instances of c_* chosen according to $\mathbf{P}$ is an ε-approximation of c_*.

In the case at hand, it is easy to just check all K possibilities for consistency in polynomial time, since K is small enough. Of course, Blumer *et al.*'s proof of the correctness of this algorithm depends on the assumption that the instances output by EXAMPLES are fully specified.

We could use that algorithm to ε', δ' learn y_1, since we always get the correct values of the instance attributes when we request a labeled example in round 1 of our learning. In fact, we use something much like that algorithm, except that we check all the possible formulas for y_1, and "output" the set of all the formulas that are consistent with all examples. (Learning y_1 is merely an internal subroutine used in the first stage of a multi-stage learning protocol; we do not really output anything at this point.)

The idea of using a set of possibilities for y_1 here is that if the set of guaranteed to contain the correct function, then when all functions in the set agree, we know we have the correct value of y_1. Otherwise we know we "don't know" y_1.

Definition. Let $F = \{f_1, ..., f_s\}$ be a set of boolean formulas defined in terms of the variables $x_1, ..., x_n$. We say F *is coherent* on an instance $x_1 \cdots x_n$ if $f_1(x_1 \cdots x_n) = f_2(x_1 \cdots x_n) = \cdots = f_s(x_1 \cdots x_n)$.

Let F_1 the set of formulas we learned for y_1. Note that for an arbitrary example, $x_1 \cdots x_n$, if F_1 is coherent on $x_1 \cdots x_n$, then the common value of the

Procedure CoherentSetLearner (hereinafter CSL)

Inputs: i; $F_1, \ldots, F_{i-1}$ (previously learned formula sets for $y_1, \ldots, y_{i-1}$), K, ϵ, and δ.
Output: F_i, a set of formulas for y_i, or "Fail."

Procedure:
Pick m according to equation (6), using K for $|C|$.
Call GetSample(m, i), returning a result R.
If GetSample did not return "Fail"
  then output $F_i \leftarrow \{f \in EB(V_i) \mid f$ is consistent with all m filtered examples in $R\}$.
  else output "Fail."

Subroutine GetSample(m, i)
Repeat until either m "filtered" examples are obtained, or $2m$ attempts are made:
    Obtain an example, $x = x_1 \cdots x_n$, by calling EXAMPLES.
    For $j \leftarrow 1$ to $i - 1$ do
        If F_j is coherent for $x_1 \cdots x_n, y_1 \cdots y_{j-1}$
            then y_j is assigned the common value
            else exit the For j loop.
    If every F_j, $1 \leq j \leq i - 1$, was coherent on x
        then consider x to be successfully "filtered," and save it.
        else discard x.
If m filtered examples were obtained
    then return those m filtered examples
    else return "Fail".

FIG. 2. Procedure CSL.

formulas must be the true value for $y_1(x_1 \cdots x_n)$. The reason is that we know that the true formula for y_1 is contained in F_1.

Thus, in order to learn an arbitrary y_i, we are led to use Procedure CoherentSet-Learner (or CSL hereafter), which is specified in Fig. 2. The key thing to notice in Procedure CSL is that once an example has been successfully "filtered" (in Subroutine GetSample) then—for that example—we know not only the values of the instance attributes $x_1 \cdots x_n$, but also the values of $y_1, \ldots, y_{i-1}$.

To make formal statements about Procedure CSL, we need to extend the definition of coherence to sequences of sets of formulas.

DEFINITION. Let $F_1, \ldots, F_j$ be a collection of sets of boolean formulas, where all the formulas in set F_i specify functions of $n + i - 1$ boolean variables $x_1, \ldots, x_n, y_1, \ldots, y_{i-1}$ for some n. The collection is *coherent* on an instance $x_1 \cdots x_n$ if F_1 is coherent on $x_1 \cdots x_n$, and F_2 is coherent on $x_1 \cdots x_n y_1$, and so on through F_j being coherent on $x_1 \cdots x_n y_1 \cdots y_{j-1}$, where y_i is the common value of the formulas in the set F_i.

Algorithm Reliable Learner

Inputs: n, ϵ, δ, s, teacher for gate by gate learning of unknown concept c_*.
Output: Program to classify instances.

Procedure:
$K \leftarrow 8\binom{n+s-1}{2} + 2$;
$\epsilon' \leftarrow \epsilon/sK$;
$\delta' \leftarrow \delta/2s$;
For $i \leftarrow 1$ to s do
    $F_i \leftarrow \mathrm{CSL}(i, F_1, \ldots, F_{i-1}, K, \epsilon', \delta')$;
    If F_i = "Fail" then exit For i loop;
Output the standard program to classify instances specified in Figure 4.

FIG. 3. Algorithm Reliable Learner.

3.2. *Learning the Target Concept*

Procedure CSL "learns" the individual y_i, but we need another program to achieve the overall goal of learning the target concept, because we need a program whose output is a program to classify instances. That program, which we call "Reliable Learner," is specified in the algorithm in Fig. 3.

THEOREM 2. *Algorithm Reliable Learner reliably and usefully learns gate by gate any concept in PC.*

Before proving Theorem 2, we first introduce a lemma from probability theory that we will need for this proof, as well as for later proofs.

LEMMA 1 (Hoeffding's Inequality). *Let* $X_1, X_2, \ldots, X_m$ *be independent* 0–1 *random variables, each with probability* p *of being* 1. *Let* $S = \sum_i^m X_i$. *Let* r, α, *and* β *be positive constants with* $\beta \leqslant p \leqslant \alpha$. *Then*

$$\Pr[S \geqslant pm + rm] \leqslant e^{-2mr^2} \tag{7}$$

$$\Pr[S \geqslant \alpha m] \leqslant e^{-2m(\alpha - p)^2} \tag{8}$$

$$\Pr[S \leqslant \beta m] \leqslant e^{-2m(p - \beta)^2}. \tag{9}$$

Proof. See Hoeffding [8]. ∎

Hoeffding's Inequality says that if we run m Bernoulli trials each with probability of success p, then the chance of getting a number of successes different from pm by a constant fraction of m is exponentially small.

Proof of Theorem 2. Fix a target concept $c_* \in PC$ that is a function of n variables, and some straight line program for c_* of size s. It will suffice to prove the theorem to show, for any $0 < \varepsilon, \delta \leqslant 1$, that

1. with probability at least $1 - \delta$,

- no call to CSL ever returns "Fail," and,
- with probability at least $1 - \varepsilon$ the collection of sets $F_1, ..., F_s$ is coherent on a randomly drawn instance, $x_1 \cdots x_n$ and,

2. if that collection of sets is coherent on $x_1 \cdots x_n$, then $y_s(x_1 \cdots x_n)$ (making the appropriate substitutions for intermediate y_i) correctly classifies $x_1 \cdots x_n$.

Our instance space, X, is $\{0, 1\}^n$. Let $X_i(F_1, ..., F_{i-1})$ be all instances $x \in X$ such that the collection $F_1, ..., F_{i-1}$ is coherent on x. The set $X_i(F_1, ..., F_{i-1})$ contains exactly those instances that might be successfully filtered and saved by the call to GetSample within CSL(i).

For $f \in F_i$, define

$$\text{err}_i(f) = \{x \in X_i(F_1, ..., F_{i-1}) \mid f(x) \neq y_i(x)\}.$$

The set $\text{err}_i(f)$ consists of those instances that might be seen as training instances for y_i that f misclassifies.

Let us call $f \in F_i$ *accurate* if

$$\mathbf{P}(\text{err}_i(f)) \leqslant \frac{\varepsilon}{sK},$$

where $\mathbf{P}$ is the unknown probability distribution on the instance space. We will classify every F_i in one of three ways: fail, inaccurate, or accurate. An F_i which is not fail is called accurate if every $f \in F_i$ is accurate, and inaccurate otherwise. Finally, let us say that CSL(i)[3] *wins* if its output is an accurate F_i.

To begin with, note that F_1 can never be fail. If thus follows from Theorem 1 that

$$\Pr[F_1 \text{ is inaccurate}] \leqslant \delta/2s. \tag{10}$$

Hence the probability that F_1 is accurate is at least $1 - \delta/2s$.

There are two reasons that CSL(2) could output "Fail." The first is that F_1 could be inaccurate. We ignore ths possibility, because what we ultimately want to calculate is

$$\Pr[F_1 \text{ and } F_2 \text{ are accurate}]$$
$$= \Pr[F_1 \text{ is accurate}] \Pr[F_2 \text{ is accurate} \mid F_1 \text{ is accurate}].$$

[3] We will use CSL(i) as shorthand for CSL($i, F_1, ..., F_{i-1}, K, \varepsilon, \delta$).

Thus we need only worry about the case where CSL(2) outputs "Fail" but F_1 is accurate.

If F_1 is accurate, then

$$\Pr[\text{one fixed } f \in F_1 \text{ disagrees with } y_1 \text{ on a random } x] \leqslant \frac{\varepsilon}{sK} \tag{11}$$

$$\Pr[\text{any } f \in F_1 \text{ disagrees with } y_1 \text{ on a random } x] \leqslant \frac{\varepsilon}{s}, \tag{12}$$

since there are at most K formulas in F_1. Hence, the probability (given that F_1 is accurate) that F_1 is not coherent for a random instance is at most ε/s. We can use Hoeffding's Inequality (Lemma 1, Inequality (9) above) to show that the probability of F_1 not being coherent on m or more instances out of $2m$ randomly drawn instances is at most

$$e^{-4m(1/2-\varepsilon/s)^2}.$$

Now choosing m according to Eq. (6) and substituting ε/sK for ε and K for $|\mathscr{C}|$, we get

$$e^{-4m(1/2-\varepsilon/s)^2} = \left(\frac{2sK}{\delta}\right)^{-(sK/\varepsilon)(1-2\varepsilon/s)^2}$$

$$\leqslant \left(\frac{2sK}{\delta}\right)^{-sK}$$

which for our purposes is vanishingly small.[4]

This means that to determine whether F_2 is accurate we need only determine whether the output F_2 is accurate given that the sample determining F_2 was filtered by an accurate F_1. This probability is at least $1-\delta/2s$, and the reason is almost the same as the reason that the probability that F_1 is accurate is at least $1-\delta/2s$. If CSL(2) does not fail, then the probability that F_2 is inaccurate is again at most $\delta/2s$. The difference is that now the probability distribution on instances is not $\mathbf{P}$ but the conditional probability distribution $\mathbf{P}'$ defined by $\mathbf{P}'(x) = \mathbf{P}(x \mid x \in X_1)$. However, since $\mathbf{P}'(x) \geqslant \mathbf{P}(x)$ for all x in the sample used to pick the formulas in F_2, this difference can only cause the probability that F_2 is accurate to be larger.

Let $\mathscr{E}_i$ be the event that CSL(i) wins. The argument for an arbitrary call to CSL(i) is similar to that for CSL(2), and it shows that

$$\Pr[\text{CSL}(i) \text{ outputs "Fail"} \mid \mathscr{E}_1 \wedge \cdots \wedge \mathscr{E}_{i-1}] \leqslant e^{-4m(1/2-\varepsilon/n)^2},$$

[4] To get this inequality, we need to assume that $\varepsilon \leqslant 1/5$, if s can be as small as 1. Hence, throughout, our algorithms should really use the value $\min(1/5, \varepsilon)$ where the value ε is specified.

and that assuming CSL(i) does not output "Fail,"

$$\Pr[F_i \text{ is not accurate} \mid \mathscr{E}_1 \wedge \cdots \wedge \mathscr{E}_{i-1}] \leqslant \frac{\delta}{2s}. \tag{13}$$

Thus we have that

$$\Pr[\mathscr{E}_i \mid \mathscr{E}_1 \wedge \cdots \wedge \mathscr{E}_{i-1}] \geqslant 1 - \left(\frac{\delta}{2s} + \left(\frac{2sK}{\delta}\right)^{-sK}\right). \tag{14}$$

Now we compute the probability that all calls to CSL win. Then

$$\begin{aligned}\Pr[\mathscr{E}_1 \wedge \mathscr{E}_2 \wedge \cdots \wedge \mathscr{E}_s] &= \Pr[\mathscr{E}_1] \cdot \Pr[\mathscr{E}_2 \mid \mathscr{E}_1] \cdots \Pr[\mathscr{E}_s \mid \mathscr{E}_1 \wedge \cdots \wedge \mathscr{E}_{s-1}] \\ &\geqslant \left(1 - \left(\frac{\delta}{2s} + \left(\frac{2sK}{\delta}\right)^{-sK}\right)\right)^s \\ &\geqslant 1 - \delta, \end{aligned} \tag{15}$$

since $(2sK/\delta)^{-sK} < \delta/2s$.

Thus the probability that all calls to CSL win is at least $1-\delta$. By definition, if all calls to CSL win, then no call outputs "Fail," and all the F_i are accurate. Now we must show that if evey F_i is accurate, then the total probability weight assigned by **P** to instances x for which some F_i is not coherent is at most ε.

We showed above in inequality (12) that the probability weight **P** assigns to instances on which F_1 is not coherent (given that F_1 is accurate) is at most ε/s. Similar reasoning shows that the probability weight **P** assigns to instances in $X_i(F_1, ..., F_{i-1})$ for which F_i is not coherent is at most ε/s. (As was the case for the examples for F_2, the examples for F_i are not drawn from distribution **P**, but from a distribution that may put more weight than **P** does on the examples we care about. Thus the error measured by **P** can be *at most* ε/s.) Hence total probability weight assigned to instances for which some F_i is not coherent is at most $s(\varepsilon/s) = \varepsilon$ as desired. ∎

Thus we have as simple program (see Fig. 4) that with probability $1-\delta$ classifies most examples correctly, and "knows," because it found some incoherent F_i, when it is given one of the rare examples it cannot classify.

On the other hand, if we really want to simply pac learn, and output a boolean circuit, we can do that as well by doing the following: Pick any formula for y_1 from F_1 to obtain a gate computing y_1. Use this gate wherever y_1 is called for later. In the same manner, pick any formula from F_2 to be a gate for computing y_2. Continue in this fashion until we finally have a circuit for y_s taking only variables $x_1 \cdots x_n$ as inputs.

```
Program to classify instances

Input: instance x_1 ... x_n.
Output: 1, 0, or "I don't know."
Note: This program depends on the global variables F_1, ..., F_s.


For j ← 1 to s do
    If F_j = "Fail"
      then halt and output "I don't know."
      else
          If F_j is coherent on x_1 ... x_n, y_1, ..., y_{j-1}
            then y_j ← common value of the functions in F_j.
            else halt and output "I don't know."
Output y_s.
```

FIG. 4. Program to classify instances.

COROLLARY 1. *The output program of Reliable Learner can be converted to a boolean circuit, resulting in pac learning.*

3.3. *Removing the Circuit Size as an Input*

This section shows that we can still use Procedure CSL as a subroutine for learning reliably and probably usefully even if the size of the target concept is not known by the learner.

The idea of the previous method is that when we know s we can spread out our tolerance ε by ε/s per line for a total of ε, and we can similarly spread out δ. Now, not knowing s, we allocate $6\varepsilon/\pi^2 i^2$ for line i, for a total of

$$\sum_{i=1}^{s} \frac{6\varepsilon}{\pi^2 i^2} < \frac{6\varepsilon}{\pi^2} \sum_{i=1}^{\infty} \frac{1}{i^2} = \varepsilon,$$

and we can similarly allocate δ. (This idea of spreading error out like $1/i^2$ goes back at least to Angluin [1].) Of course that is merely the intuition; the details are specified in the description of Algorithm Reliable Learner 2 (Fig. 5), which we now prove to be correct.

THEOREM 3. *Algorithm Reliable Learner 2 reliably and usefully learns gate by gate any concept in PC.*

Proof. We simply give the changes that need to be made to the proof of Theorem 2. We now define $f \in F_i$ to be accurate only if

$$\sum_{x \in \mathrm{err}_i(f)} \mathbf{P}(x) \leq \frac{6\varepsilon}{K\pi^2 i^2}.$$

Algorithm Reliable Learner 2

Inputs: n, ϵ, δ, teacher for gate by gate learning of unknown concept c_*.
Output: Program to classify instances.

Procedure:
For $i \leftarrow 1$ to s do
$K \leftarrow 8\binom{n+i-1}{2} + 2$;
$\epsilon' \leftarrow 6\epsilon/K\pi^2 i^2$; $\delta' \leftarrow 3\delta/\pi^2 i^2$;
$F_i \leftarrow \mathrm{CSL}(i, F_1, \ldots, F_{i-1}, K, \epsilon', \delta')$;
If F_i = "Fail" then exit For i loop;
Output the standard program specified in Figure 4.

FIG. 5. Algorithm Reliable Learner 2.

We continue to say that F_i is accurate if every $f \in F_i$ is accurate, and that CSL(i) wins if its output is an accurate F_i as opposed to either an inaccurate F_i or "Fail."

Equation (10) becomes

$$\Pr[F_1 \text{ is not accurate}] \leqslant \frac{3\delta}{\pi^2}. \tag{16}$$

Its generalization, Eq. (13), now states that assuming that the call to CSL(i) does not output "Fail," we have

$$\Pr[F_i \text{ is inaccurate} \mid \mathrm{CSL}(1), \ldots, \mathrm{CSL}(i-1) \text{ win}] \leqslant \frac{3\delta}{\pi^2 i^2}. \tag{17}$$

The chance that a call to CSL(i) outputs "Fail" if all the F_j for $1 \leqslant j \leqslant i-1$ are accurate is still exponentially vanishing. Thus Eq. (14) becomes, again letting $\mathscr{E}_i$ be the event that CSL(i) wins,

$$\Pr[\mathscr{E}_i \mid \mathscr{E}_1 \wedge \cdots \wedge \mathscr{E}_{i-1}] \geqslant 1 - \left(\frac{3\delta}{\pi^2 i^2} + \left(\frac{\pi^2 i^2}{3\delta}\right)^{-Ki^2}\right). \tag{18}$$

Equation (15) now becomes

$$\begin{aligned}
\Pr[\mathscr{E}_1 \wedge \mathscr{E}_2 \wedge \cdots \wedge \mathscr{E}_s] &= \Pr[\mathscr{E}_1] \cdot \Pr[\mathscr{E}_2 \mid \mathscr{E}_1] \cdots \Pr[\mathscr{E}_s \mid \mathscr{E}_1 \wedge \cdots \wedge \mathscr{E}_{s-1}] \\
&\geqslant \prod_{i=1}^{s} \left(1 - \left(\frac{3\delta}{\pi^2 i^2} + \left(\frac{\pi^2 i^2}{3\delta}\right)^{-Ki^2}\right)\right)^s \\
&> 1 - \sum_{i=1}^{s} \frac{6\delta}{\pi^2 i^2} \\
&> 1 - \delta.
\end{aligned}$$

The argument for the parameter ε is similar to the argument for the parameter δ. ▮

4. Noise

The previous section showed how to reliably and probably usefully learn any concept in *PC*. However, the algorithm given there requires that the examples given by the teacher be completely free of noise. This section shows how that algorithm can be modified to tolerate noise in the data, albeit with a slight degradation in the learning.

From perfect data we saw that we could achieve

$$\text{Guaranteed reliable and } [\Pr[\text{useful}] \geqslant 1 - \delta].$$

In the presence of noise we achieve

$$\Pr[\text{Reliable and useful}] \geqslant 1 - \delta.$$

Note that for the purpose of applications, this degradation is not as bad as it seems at first. All our algorithms have sample complexity and running time polynomial in $\log(1/\delta)$, so we can afford to make δ very small.

We begin with a discussion of various ways of modeling noisy data, and then move on to algorithms to cope with noisy data.

4.1. *Models of Noise*

We consider two different sorts of noise: one in which instances may be incorrectly labeled (e.g.: a positive instance labeled "$-$"), and one in which data may be completely corrupted. In general it is possible to tolerate much higher levels of classification noise [3, 9, 22] than arbitrary noise, and that is also the case here.

Formally we model noise by imagining that the good oracle EXAMPLES is replaced by some other oracle which sometimes gives "bad" data. The "desired," noiseless output of one of these oracles would thus be a correctly labeled point (x, l), where x is drawn according to **P**. In all cases we will assume that the oracle is given a correctly labeled (x, l) where x is drawn according to **P**, and that with probability $1 - \nu$ for some $\nu < 1/2$, the oracle must return this noiseless example. Now we describe what happens in the probability ν noisy case for each oracle.

- The oracle MAL_ν returns an example $(\hat{x}, \hat{l})$ about which no assumptions whatsoever may be made. In particular, this example may be maliciously selected by an adversary who has infinite computing power, and has knowledge of c_*, **P**, ν, and the internal state of the algorithm calling this oracle. This oracle is meant to model the situation where the learner usually

gets a correct example, but some small fraction ν of the time the learner gets noisy examples and the nature of the noise is unknown or unpredictable.

- The random misclassification oracle RMC_νreturns $(x, \bar{l})$ (i.e., x with the wrong label). In this model the only source of noise is random misclassification.
- The malicious misclassification oracle $\mathrm{MMC}_\nu(x, \hat{l})$ where l is a label about which no assumptions whatsoever may be made. As with MAL_ν we assume an omnipotent, omniscient adversary; but in this case the adversary only gets to choose the label of the example. This oracle is meant to model a situation where the only source of noise is misclassification, but the nature of the misclassification is unknown or unpredictable.

For some learning from noisy data, it is necessary to give the learning algorithms an upper bound, ν_b, on the noise rate, though not the actual noise rate itself.

4.2. *Classification Noise*

We begin by showing how to modify Procedure CSL to tolerate noise coming from RMC_ν, and then go on to argue that this modification in fact tolerates noise coming from MMC_ν. Call this new procedure CSL2.

In the case of classification noise, the learner receives one additional input, ν_b, an upper bound on the noise rate. The learner knows that all the examples come from RMC_ν for some fixed $0 \leqslant \nu \leqslant \nu_b$. Of course the learner is now allowed time polynomial in $1/(1-2\nu_b)$ in addition to the other parameters [3].

There are two differences between Procedure CSL and Procedure CSL2. The first is that the value of m is changed to

$$m = \frac{8}{\varepsilon^2(1-2\nu_b)^2} \ln\left(\frac{3sK}{\delta}\right). \tag{19}$$

The second difference between CSL and CSL2 is that the formulas for F_i are chosen in a different way once a sample of m "good" examples have been obtained. In Procedure CSL2, for every condidate formula f for y_i, we calculate d_f, the number of disagreements between f and the labels of the teacher for the sample. Clearly $0 \leqslant d_f \leqslant m$. In Procedure CSL we simply set F_i to be the set of all f such that $d_f = 0$. We knew there would be at least one such f, because in the noise-free case, $d_{y_i} = 0$. Now that there is classification noise, however, there may not be any f for which d_f is equal to 0.

Now instead we find a formula f_* such that d_{f_*} is minimal, and set

$$F_i = \{f \mid d_f \leqslant d_{f_*} + 2t\}, \tag{20}$$

Procedure CSL2

Inputs: i; $F_1, \ldots, F_{i-1}$ (previously learned formula sets); ν_b, m, and ϵ.
Output: F_i, a set of formulas for y_i, or "Fail."

Procedure:
Call GetSample(m, i); (Specified above in Figure 2.)
If GetSample returned "Fail"
  then output "Fail"
  else $t \leftarrow \epsilon(1 - 2\nu_b)m/4$;
    for every $f \in \mathrm{EB}(V_i)$ compute mistake number d_f;
    Set $dmin \leftarrow \min\{d_f\}$;
    $F_i \leftarrow \emptyset$;
    for every $f \in \mathrm{EB}(V_i)$
      If $d_f \leq dmin + 2t$ then put f in F_i.

FIG. 6. Procedure CSL2.

where

$$t = \frac{\varepsilon(1 - 2\nu_b)m}{4sK}.$$

We formally specify CSL2 in Fig. 6. (Also, we have slightly altered the parameter passing, so we specify Algorithm Noisy Learner to call CSL2 in Fig. 7, although it is only slightly different from Algorithm Reliable Learner.)

Algorithm Noisy Learner

Inputs: $\nu_b, n, \epsilon, \delta$, s, teacher for gate by gate learning of unknown concept c_*.
Output: Program to classify instances.

Procedure:
$K \leftarrow 8\binom{n+s-1}{2} + 2$;
$\epsilon' \leftarrow \epsilon/sK$;
Set m according to equation (19);
For $i \leftarrow 1$ to s do
  $F_i \leftarrow$ CSL2$(i, F_1, \ldots, F_{i-1}, \nu_b, m, \epsilon')$;
  If $F_i =$ "Fail" then exit For i loop;
Output the standard program specified in Figure 4.

FIG. 7. Algorithm Noisy Learner.

THEOREM 4. *Given inputs* $0 \leq \nu_b < 1/2$, n, $0 < \varepsilon$, $\delta \leq 1$, s, *and with examples from* RMC_ν *for some* $\nu \leq \nu_b$, *Algorithm Noisy Learner produces an output that, with probability at least* $1 - \delta$, *is reliable and classifies at most probability weight* ε *of instances "I don't know."*

Proof. We want to show that the proof of Theorem 2 still goes through, with slight modifications. We proved two key facts about each F_i in that proof. The first was that the true formula for y_i was guaranteed to be in F_i. The second was that with high probability each F_i was accurate, where accurate was defined to mean that every formula in F_i agreed with y_i with probability at least $1 - \varepsilon/s$. Let us say that an F_i is "very good" if it has both of those two properties. We now show that with high probability every F_i is very good. (Because of the noise, we can no longer hope to *guarantee* that the true formula for y_i is in F_i.) In particular we show that

$$\Pr[y_i \in F_i \text{ and } F_i \text{ is accurate} \mid F_1, \ldots, F_{i-1} \text{ are all very good}] \geq 1 - \delta/s. \quad (21)$$

Assume now that $F_1, F_2, \ldots, F_{i-1}$ are all very good. That means that we have the full set of correct attribute bits in the m examples we use to determine F_i (although, of course, some of the labels may be wrong).

Note that $\varepsilon'(1 - 2\nu_b)$ is the expected gap between the disagreement rate for the true formula for y_i and the disagreement rate for any formula that is not an ε'-approximation of the true formula. Thus if we were to make F_i the set of all f such that $d_f \leq \mathrm{E}[d_{y_i} + 2t]$, then we would have that with high probability, F_i is very good. Now $\mathrm{E}[d_{y_i}] = \nu m$, but unfortunately we do not know the value ν.

Nevertheless, to prove that Eq. (21) holds it suffices to show that the sum of the probability of the negation of each of the following three events is at most δ/s.

1. No formula has a number of disagreements less than $\mathrm{E}[d_{y_i}] - t$.
2. For the true formula y_i we have $d_{y_i} \leq \mathrm{E}[d_{y_i}] + t$.
3. Every (ε/sK)-bad formula has more than $\mathrm{E}[d_{y_i}] + 3t$ disagreements with the sample.

If both of the first two events occur, then we have that $d_{y_i} \leq d_{f_*} + 2t$, so y_i is placed in F_i. In such a case we also have that $d_{y_i} \leq \mathrm{E}[d_{y_i}] + t$, and thus that $d_{f_*} \leq \mathrm{E}[d_{y_i}] + t$. Together with the occurrence of event 3 this guarantees that no (ε/sK)-bad formula is put in F_i.

For every formula f we have that $\mathrm{E}[d_f] \geq \mathrm{E}[d_{y_i}]$. Thus we get from Hoeffding's Inequality (Eq. (7)) that

$$\Pr[f \text{ has } d_f \leq \mathrm{E}[d_{y_i}] - t] \leq e^{-2m(\varepsilon(1-2\nu_b)/4sK)^2}$$

$$\leq \frac{\delta}{3Ks}.$$

Thus the probability of event 1, that any f has a d_f that is too small, is at most $\delta/3s$.

A similar argument shows that the probability of event 2 is at most $\delta/3Ks$, and that the probability of event 3 is at most $\delta/3s$.

Given this, the rest of the proof is completely analogous to the proof of Theorem 2. ▮

THEOREM 5. *Procedure* CSL2 *has the performance specified in Theorem* 4 *even if the examples come from* MMC_ν.

Proof. We want to show that the condition specified by inequality (21) still holds in this case. To do this, we use a proof technique from Sloan [22]. The only difference between MMC_ν and RMC_ν is that there may be some examples that were labeled incorrectly by RMC_ν that now the adversary decides to label correctly. We show that in spite of any such change of label, if y_i was in F_i when the examples came from RMC_ν then it still is in F_i, and that no ε'-bad formula that was not placed in F_i before is now.

The only way for y_i not to be in F_i is if some other formula has $ms/2$ fewer disagreements with the sample than y_i has. Changing incorrect labels to correct labels cannot affect whether that is true. Changing an incorrect label to a correct label may or may not decrease the number of disagreements for any formula but y_i by 1; such a change must decrease d_{y_i} by 1.

The way we showed that no ε'-bad formula was placed in F_i was by showing that every such formula had a number of disagreements at least $ms/2$ greater than d_{y_i}. Again, changing incorrect labels to correct labels cannot eliminate such a gap. ▮

4.3. *Malicious Noise*

The strategy for learning in the presence of malicious noise is similar to the strategy for learning in the presence of classification noise. We only consider malicious noise rates $\nu \leqslant \varepsilon/4sK$. Given a sample from MAL_ν, the expected mistake rate of any concept that is not (ε/sK)-bad must be at most $(\varepsilon/sK) - \nu$. Therefore, there must be a gap g in expected mistake rates between the target concept and any (ε/sK)-bad concept of at least

$$g = \frac{\varepsilon}{sK} - 2\nu$$

$$\geqslant \frac{\varepsilon}{2sK},$$

since $\nu \leqslant \varepsilon/4sK$.

Procedure CSL3

Inputs: i; $F_1, \ldots, F_{i-1}$ (previously learned formula sets); K, ϵ, and δ.
Output: F_i, a set of formulas for y_i, or "Fail."

Procedure:
Pick m_{MAL} according to equation (22).
Call GetSample(m_{MAL}); (Specified in Figure 2.)
If GetSample returned "Fail"
  then $F_i \leftarrow$ "Fail"
  else $F_i \leftarrow \{f \in \mathrm{EB}(V_i)$: mistake rate of $f \leq \epsilon/2sK\}$.

FIG. 8. Procedure CSL3.

We exploit this gap by getting a sample of size

$$m_{\mathrm{MAL}} = \frac{8s^2K^2}{\varepsilon^2} \ln\left(\frac{3sK}{\delta}\right), \tag{22}$$

and putting into our set any rule with a disagreement rate less than $v + g/2$. The precise algorithm, CSL3, is specified in Fig. 8. We specify Algorithm Malicious Learner in Fig. 9.

THEOREM 6. *Given inputs* n, $0 < \varepsilon$, $\delta \leqslant 1$, s, *and with examples of any concept in PC from* MAL_v *for some* $v \leqslant \varepsilon/4sK$ (*where* K *is from Eq.* (5)), *the output of Algorithm Malicious Learner, with probability at least* $1 - \delta$, *correctly classifies all but at most probability weight* ε *of instances.*

Proof Sketch. The proof is similar to the proof of Theorem 4. We still say the set F_i is very good if $y_i \in F_i$ and if the set of instances for which any $f \in F_i$ disagrees with y_i has probability weight at most ε/s.

Algorithm Malicious Learner

Inputs: n, ϵ, δ, s, access to MAL_v.
Output: Program to classify instances.

Procedure:
$K \leftarrow 8\binom{n+s-1}{2} + 2$;
For $i \leftarrow 1$ to s do
  $F_i \leftarrow$ CSL3($i, F_1, \ldots, F_{i-1}, K, \epsilon, \delta$);
  If $F_i =$ "Fail" then exit For i loop;
Output the standard program specified in Figure 4.

FIG. 9. Algorithm Malicious Learner.

Assume now that $F_1, F_2, ..., F_{i-1}$ are all very good, and that in CSL3(i) the call to GetSample did not return "Fail." In order to show that F_i is very good we need the following two events to occur.

1. The mistake rate of y_i must be less than $\varepsilon/2sK$.

2. Every (ε/sK)-bad formula $f \in \mathrm{EB}(V_i)$ must have a mistake rate of at least $\varepsilon/2sK$.

Since the samples come from the malicious error oracle, we cannot simply say that d_{y_i}, the mistake number of y_i, is the sum of Bernoulli trials. Nevertheless, the worst thing that the malicious error oracle can do to d_{y_i} is add one to it every time it affects the examples. Thus, d_{y_i} has expectation no greater than the sum of m_{MAL} Bernoulli random variables each with probability $\varepsilon/4sK$ of being 1. As usual we use Hoeffding's Inequality to show that the probability of getting a value as high as $2(\varepsilon/2sK)\, m_{\mathrm{MAL}}$ is at most $(\delta/2K)$.

The argument for the second event, an (ε/sK)-bad formula having too low a mistake rate, is similar. ▮

5. Summary and Conclusions

In this paper, we have shown how to learn complicated concepts subconcept by subconcept. The key technique is to maintain a list of *all* possible condidates (the "version space") for the subconcept, instead of simply picking some one candidate. For the purposes of this paper, we were concerned with the class EB, but our method is applicable to any polynmial size class. We expect that this particular method will prove to have other applications.

We believe this general approach is the philosophically sound way to do inductive inference, since what distinguishes induction from deduction is that in induction one can never be completely certain that one has learned correctly. (Kugel [13] contains an interesting discussion of this point.) It is always possible that one will see a counterexample to one's current favorite theory. This idea of maintaining a list of all the candidates for the correct "answer" has recently borne fruit elsewhere in the field of inductive inference as well, in a new model of recursion theoretic inductive inference [20], and in a method for inference of simple assignment automata [21].

Another contribution of this paper has been to introduce the notion of learning that is reliable and probably useful, and to give a learning procedure that achieves such learning.

In fact, our learning procedure is in one sense not merely reliable, but even better; because it has maintained candidate sets for all subconcepts, it

need not simply output "I don't know" on difficult instances. It has maintained enough information to be able to know which subconcept is causing it to output "I don't know." Thus, in a learning environment where it is appropriate to do so, our learning procedure can go back and request more help from the teacher on that particular subconcept.

ACKNOWLEDGMENTS

We thank Dana Angluin and Manfred Warmuth for some interesting discussions.

RECEIVED December 7, 1990; FINAL MANUSCRIPT RECEIVED April 24, 1992

REFERENCES

1. ANGLUIN, D. (1988), Queries and concept learning, *Mach. Learning* **2**(4), 319–342.
2. ANGLUIN, D., GASARCH, W. I., AND SMITH, C. H. (1987), "Training Sequences," Technical Report UMIACS-TR-87-37, University of Maryland Institute for Advanced Computer Studies.
3. ANGLUIN, D., AND LAIRD, P. (1988), Learning from noisy examples, *Mach. Learning* **2** (4), 343–370.
4. BLUMER, A., EHRENFEUCHT, A., HAUSSLER, D., AND WARMUTH, M. K. (1987), Occam's razor, *Inform. Process. Lett.* **24**, 377–380.
5. GOLDREICH, O., GOLDWASSER, S., AND MICALI, S. (1986), How to construct random functions, *J. Assoc. Comput. Mach.* **33**(4), 792–807.
6. HAUSSLER, D. (1986), Quantifying the inductive bias in concept learning, *in* "Proceedings, AAAI-86," pp. 485–489, American Association for Artificial Intelligence.
7. HAUSSLER, D. (1987), Bias, version spaces and Valiant's learning framework, *in* "Proceedings of the Fourth International Workshop on Machine Learning," University of California, Irvine, pp. 324–336.
8. HOEFFDING, W. (1963), Probability inequalities for sums of bounded random variables, *J. American Statistical Association* **58**(301), 13–30.
9. KEARNS, M., AND LI, M. (1988), Learning in the presence of malicious errors, *in* "Proceedings of the Twentieth Annual ACM Symposium on Theory of Computing," Chicago.
10. KEARNS, M., LI, M., PITT, L., AND VALIANT, L. (1987), On the learnability of boolean formulae, *in* "Proceedings of the Nineteenth Annual ACM Symposium on Theory of Computing," New York, pp. 285–295.
11. KEARNS, M., LI, M., PITT, L., AND VALIANT, L. (1987), Recent results on boolean concept learning, *in* "Proceedings of the Fourth International Workshop on Machine Learning," University of California, Irvine, pp. 337–352.
12. KEARNS, M., AND VALIANT, L. G. (1989), Cryptographic limitations on learning boolean formulae and finite automata, *in* "Proceedings of the Twenty-First Annual ACM Symposium on Theory of Computing," Seattle, pp. 433–444.
13. KUGEL, P. (1977), Induction, pure and simple, *Inform. and Control* **35**, 276–336.
14. LAIRD, J., ROSENBLOOM, P., AND NEWELL, A. (1984), Towards chunking as a general learning mechanosm, *in* "Proceedings AAAI-84," pp. 188–192.

15. LAIRD, J., ROSENBLOOM, P., AND NEWELL, A. (1986), Chunking in Soar: the anatomy of a general learning mechanism, *Mach. Learning* **1**(1), 11–46.
16. MILLER, G. (1956), The magic number seven, plus or minus two: Some limits on our capacity for processing information, *Psychol. Rev.* **63**, 81–97.
17. MITCHELL, T. M. (1977), Version spaces: A candidate elimination approach to rule learning, *in* "Proceedings IJCAI-77," Cambridge, MA, pp. 305–310, International Joint Committee for Artificial Intelligence.
18. PITT, L., AND VALIANT, L.G. (1988), Computational limitations on learning from examples, *J. Assoc. Comput. Mach.* **35**(4), 965–984.
19. PITT, L., AND WARMUTH, M. K. (to appear), The minimum DFA consistency problem cannot be approximated within any polynomial, *J. Assoc. Comput. Mach.*, pending minor revision.
20. RIVEST, R. L., AND SLOAN, R. (submitted for publication), A new model for inductive inference, a preliminary version is available, *in* "Proceedings of the Second Conference on Theoretical Aspects of Reasoning about Knowledge," pp. 13–27. Morgan Kaufmann, 1988.
21. SCHAPIRE, R. E. (1988), "Diversity-Based Inference of Finite Automata," Master's Thesis, MIT Laboratory for Computer Science; Technical Report MIT/LCS/TR-413.
22. SLOAN, R. H. (1988), Types of noise in data for concept learning, *in* "First Workshop on Computatonal Learning Theory," pp. 91–96, Morgan Kaufmann.
23. VALIANT, L. G. (1984), A theory of the learnable, *Comm. ACM* **27**(11), 1134–1142.
24. VANLEHN, K. (1987), Learning one subprocedure per lesson, *Articial Intelligence* **31**(1), 1–40.