Learning Complicated Concepts Reliably and Usefully (Extended Abstract)

Ronald L. Rivest[*] and **Robert Sloan**[†]
MIT Lab. for Computer Science
Cambridge, Mass. 02139 USA

Abstract

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 which, 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.

1 Introduction

The field of inductive inference has been greatly broadened by Valiant's seminal paper [Valiant, 1984] 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 examples. Moreover, in that paper, and in a number of subsequent papers, (eg.: [Haussler, 1986; Kearns *et al.*, 1987a; Pitt and Valiant, 1986]) 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 [Valiant, 1984] that learning arbitrary polynomial size circuits is computationally intractable, and Pitt and Valiant [Pitt and Valiant, 1986] show that learning certain particular interesting classes of boolean functions, for instance, boolean threshold formulas, is NP-complete.

In this paper we will examine one path around this obstacle—a way that a suitably helpful teacher can teach *any* polynomial size boolean function.

1.1 Hierarchical learning

The way we will 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 [Miller, 1956].
- 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 future learning tasks [Laird *et al.*, 1984; Laird *et al.*, 1986]. Also, the SIERRA system learns how to do arithmetic in a manner broadly similar to what we will suggest; it learns "one subprocedure per lesson." [VanLehn, 1987]
- Within the framework of theoretical inductive inference, Angluin et. al. [Angluin *et al.*, 1987] recently showed how to learn certain otherwise unlearnable recursive functions by first learning relevant subconcepts.

1.2 Review of Valiant Model

Before we can 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 [Kearns *et al.*, 1987b].

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."

[*]This paper was prepared with support from NSF grant DCR-8607494, ARO Grant DAAL03-86-K-0171, and the Siemens Corporation.

[†]Supported by an NSF graduate fellowship and by the Siemens Corporation.

Authors' ARPAnet addresses are: rivest@theory.lcs.mit.edu, sloan@theory.lcs.mit.edu

In order to precisely define learnability, we must first specify what it is we are trying to learn. For the purposes of this paper, imagine a universe U of objects each having n attributes. In this paper we assume the attributes are binary, although this assumption is not crucial to the learnability model. For instance, U might be the inhabitants of the U. S., and the attributes might include Sex, Age<19, Income<1900, and IsInCollege. Formally, $U = \{0,1\}^n$.

A *concept*, c, is a rule that splits U into *positive instances* and *negative instances*. Given that we have $U = \{0,1\}^n$, possible representations for concepts would include truth tables, boolean formulas, and boolean circuits. (See [Haussler, 1987].) We say that the *length* of concept c, $|c|$, is the number of bits required to write down c in whatever representation we have chosen. A *concept class* is a set of concepts all defined on U.

If U is the inhabitants of the U. S., concepts would include both the rather simple concept *left-handed adult males*, defined by the obvious conjunction, and the doubtless more complicated concept, *people required to pay at least five hundred dollars in federal income tax*. An example of a concept class would be TAX BRACKETS which would include the concepts *people paying no income tax* and *people required to pay at least five hundred dollars in federal income tax*.

We assume that our algorithm trying to learn concept c has available to it a black box called EXAMPLES, and that each call to the black box returns a labeled example, (x, s), where $x \in U$ is an instance, and s is either $+$ or $-$ according to whether x is a positive or negative instance of the target concept c. Furthermore, the EXAMPLES box generates the instances x according to some probability distribution D on U. We make no assumptions whatsoever about the nature of D, and our learner will not be told what D is.

DEFINITION. Let $\mathcal{C}$ be a class of concepts. We say algorithm A *probably approximately correctly learn (pac learns)* $\mathcal{C}$ if and only if there is a polynomial P such that $(\forall n)(\forall c \in \mathcal{C})(\forall D)(\forall \epsilon > 0)(\forall \delta > 0)$, A, given only n, ϵ, δ and access to EXAMPLES(c), halts in time $P(n, |c|, \frac{1}{\epsilon}, \frac{1}{\delta})$, and outputs some representation of a concept, c' that with probability at least $1 - \delta$ has the property

$$\sum_{c'(x) \neq c(x)} D(x) < \epsilon.$$

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

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 most of the instances.

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 for 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 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 received from Nature might all have been 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.

Further discussion of the motivation for this model may be found in [Blumer *et al.*, 1986] and, of course, [Valiant, 1984] which introduced this model.

1.3 A new 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 must give as output a concept in whatever representation is being worked with—say circuits. Our learner is instead supposed to give a (polynomial time) program taking instances as input, and having three possible outputs: "Yes," "No," and "I don't know."

DEFINITION. We call learning algorithm A *reliable* if the program output by A says "Yes" only on positive instances, and says "No" 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. Thus we are led to the following definition, analogous to pac learning:

DEFINITION. Let $\mathcal{C}$ be a class of concepts. We say algorithm A *reliably probably almost always usefully learns* $\mathcal{C}$ if and only if there is a polynomial P such that $(\forall n)(\forall c \in \mathcal{C})(\forall D)(\forall \epsilon > 0)(\forall \delta > 0)$, A, given only n, ϵ, δ and access to EXAMPLES(c), halts in time $P(n, |c|, \frac{1}{\epsilon}, \frac{1}{\delta})$, and outputs a program Q such that

1. $(\forall x) Q(x) = \text{Yes} \Rightarrow c(x) = 1$, and $Q(x) = \text{No} \Rightarrow c(x) = 0$. ($A$ is reliable.)
2. With probability at least $1 - \delta$,

$$\sum_{Q(x) = \text{I don't know}} D(x) < \epsilon.$$

(A is probably almost always 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 most of the time. 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 will present an algorithm that reliably probably almost always usefully learns.

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 what have you. 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 functions that can be represented by a polynomial size boolean formula (or, similarly, a polynomial size boolean circuit).

Unfortunately, this goal is unlikely to be attainable. As is explained in [Valiant, 1984], assuming that one way functions exist (an assumption which we feel is likely to be correct), the class of polynomial size boolean formula is not pac learnable.

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

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

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

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 ∨ MyParent," and "Age<19 ∨ 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 will achieve reliable, probably almost always 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.

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

Input variables: FiledJointReturn, $>\frac{1}{2}$SupportFromMe, MyChild, MyParent, Age<19, IsInCollege, Income<1900.
Output: *dependent* $= y_7$.

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

Figure 1: A straight line program for *dependent*

Our learner can, if desired, do precisely that—output a short fast program taking instances as its input and having the three outputs, "Yes" (*dependent*), "No" (not a *dependent*), and "I don't know." This program is guaranteed to be correct whenever it gives a "Yes" or "No" classification, and moreover, with probability $1-\delta$ it says "I don't know" about at most a fraction ϵ of all people. In short, it meets our definition of reliable, probably almost always useful learning.

2.1 Notation

Before showing how to break our target concept, t, into pieces, we must first specify the problem more precisely. For convenience' sake only, we will assume t is represented as a straight-line program: Let the inputs to t be $x_1,\ldots,x_n$, and call the output y_l. (l being the number of *l*ines.) The i-th line of the program for t, for $1\le i\le l$ is of the form:

$$y_i = z_{i,1}\circ z_{i,2} \qquad (2)$$

where $\circ$ is one of the two boolean operators $\vee$ and $\wedge$, and every $z_{i,k}$ is either a literal, or else y_j or $\bar{y}_j$ for some previously computed y_j (i.e., $j<i$).[2] We say l is the *size* of such a straight line program. In Figure 1 we show a straight line program for the *dependent* concept defined in equation 1 above.

2.2 An easy but trivial way to learn

As a first attempt to develop a protocol for learning our arbitrary t piece by piece, we might try the following: Have the teacher supply not only examples, but also the pieces—the y_i. In particular, let the y_i be rearranged in some arbitrary order, $y_{j_1},\ldots,y_{j_l}$. Now, each time the learner requests an example, he gets more than just a labeled example drawn according D. The learner receives $x_1,\ldots,x_n\#y_{j_1},\ldots,y_{j_l}$ (and its label). Given all this help,

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

it turns out to be easy to learn. It is not hard for the learning algorithm to determine which of the other variables a given variable depends on.

This solution is not very satisfying, however, since it requires that the learner receive a large amount of "extra help" with each and every example. In essence, it would mean that every time our learner was given an example while learning ***dependent***, he would have to be told whether it was a child in college, whether it was a relative, and so on. Our approach will be 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 will be one fixed probability distribution, D, on examples throughout; the teacher is *not* allowed to help the student by altering it.

There will be l rounds. The teacher will move from round i to round $i+1$ when the learner tells him to do so. In round i, the learner is going to learn y_i.

When our learner requests an example during round i, the teacher will give the learner a pair, $(x_1, \ldots, x_n, s)$, where $x_1, \ldots, x_n$ is drawn according to D, and s tells whether $x_1, \ldots, x_n$ is a positive or negative instance *of* y_i. In other words, in round i, s gives the truth value of $y_i(x_1, \ldots, x_n)$ (rather than the truth value of $t(x_1, \ldots, x_n)$).

During each round i, the learner tries to $\epsilon' = \epsilon/p_1(n)$, $\delta' = \delta/p_2(n)$ learn the concept y_i where p_1 and p_2 are polynomials to be determined. This learning task at first glance appears to be extremely simple, because y_i must be a simple conjunction or disjunction of $x_1, \ldots, x_n$ and $y_1, \ldots y_{i-1}$ (and perhaps their negations). The catch is that while the learner gets the true values for $x_1, \ldots, x_n$ he only gets his computed values for $y_1, \ldots, y_{i-1}$, and these computed values are at best probably approximately correct.

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

Our job will be to show how to do this learning in such a manner that at the end, when we have a representation for y_l in terms of all the x_i and y_i, and we substitute the x_i back in for the y_i, the final expression, $y_l(x_1, \ldots, x_n)$ ϵ-approximates the target concept with probability $1 - \delta$.

In fact, as we said above, we will do something stronger. Our learner will not merely pac learn, but will reliably, probably almost always usefully learn.

A key technique

The technique we use to achieve this goal is having the learner learn and maintain a list of all possible candidates for a given y_i. For each subconcept y_i we explicitly maintain the "most specific" list of the version space representation [Mitchell, 1977].

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

$$K = 8\binom{n+l}{2}.$$

The important thing to notice is that K is polynomial in n and l, the size of (the representation of) the target concept. K takes on the particular value it does because each y_i is in the class 1CNF $\cup$ 1DNF, but the only thing special about 1CNF $\cup$ 1DNF is that it is of polynomial size. Our technique will work equally well using any polynomial-size concept class.

We exploit this technique by designing an algorithm with three fundamental parts:

1. In round i we get various examples of y_i. We will say that an example, $(x_1, \ldots, x_n, s)$, is "good" if for every previously learned y_j, $1 \leq j < i$, all the formulas in the list for y_j take on the same truth value on $x_1, \ldots, x_n$. Since one of the formulas in the list for y_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, so we can proceed to learn y_i as a function of the attributes $x_1, \ldots, x_n, y_1, \ldots, y_{i-1}$.
3. Finally, we need something to specify the algorithm that we output at the end of round l.

3 Detailed specification of our learning protocol

We assume in this paper that the learner is given l, the length of the straight line program, at the beginning of the learning protocol. This assumption will make the presentation simpler and clearer. The learner can in fact do equally well without being given l. (Details omitted.)

3.1 Learning y_i

During each round i, the learner simply needs to learn y_i as a function of the input literals and previous y_j and $\bar{y}_j$. Moreover, the formula for y_i will be in the class 1CNF $\cup$ 1DNF. There are at most K candidates for the the formula for y_i.

The traditional method of ϵ', δ' pac learning a concept y_i that is one of at most K functions of $x_1, \ldots, x_n, y_1, \ldots, y_{i-1}$, where the values of those attribute variables is known perfectly is the following [Valiant, 1984; Blumer *et al.*, 1987]:

The learner chooses

$$m \geq \frac{1}{\epsilon'}\left(\ln(K) + \ln\left(\frac{1}{\delta'}\right)\right) \tag{3}$$

and obtains m labeled instances from EXAMPLES. The learner then checks the candidates for the formula for y_i (at most K) one by one until one is found that is one consistent with all m examples, and outputs that candidate.

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

The idea of using a set here is that, 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 $F = \{f_1, \ldots, f_s\}$ be a set of boolean formulas, each of the same number of variables, say n. We say F *is coherent on* $x_1, \ldots, x_n$ if $f_1(x_1, \ldots, x_n) = f_2(x_1, \ldots, x_n) = \cdots = f_s(x_1, \ldots, x_n)$.

Let $F_1 = \{f_{1,1}, f_{1,2}, \ldots f_{1,i_1}\}$ be the set of formulas we learned for y_1. Notice that for an arbitrary example, $x_1, \ldots, x_n$, if F_1 is coherent on $x_1, \ldots, x_n$, then the common value of the formulas must be the true value for $y_1(x_1, \ldots, x_n)$. The reason is that we know 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 ConsistentSetLearner (hereinafter CSL)
Inputs: i; $F_1, \ldots, F_{i-1}$ (previously learned formula sets for $y_1, \ldots, y_{i-1}$); n, ϵ', and δ'.
Output: F_i, a set of formulas for y_i, or "Fail."
Pick m according to equation 3. Repeat the following until either m "good" examples have been obtained, or else $2m$ attempts have been made. In the latter case, output "Fail."
Obtain an example, $x_1, \ldots, x_n$, by calling EXAMPLES. If every F_j, $1 \leq j \leq i-1$, is coherent on $x_1, \ldots, x_n$, consider $x_1, \ldots, x_n$ to be "good," and save it. If not, discard it. Once m "good" examples have been obtained, output all candidate formulas for y_i (as a function of $x_1, \ldots, x_n$ and $y_1, \ldots, y_{i-1}$) that are consistent with all m "good" examples.

The key thing to note in Procedure CSL is that once an example has been found to be good, then—for that example—we know not only the values of the instance attributes $x_1, \ldots, x_n$, but also the values of $y_1, \ldots y_{i-1}$.

3.2 Learning the target concept

Procedure CSL does indeed give us a way to learn our target concept t once we calculate appropriate values for ϵ' and δ'.

Theorem 1 *Let* $\epsilon' = \epsilon/lK$. *Let* $\delta' = \delta/l$. *Call* CSL(1), CSL(2), ..., CSL(l). *Then,*

1. *with probability at least* $1 - \delta$,
   - *no call ever returns "Fail," and,*
   - *with probability at least* $1-\epsilon$ *every* F_i *is coherent on a randomly drawn instance,* $x_1, \ldots, x_n$ *and,*
2. *if every* F_i *is coherent on* $x_1, \ldots, x_n$, *then* $y_l(x_1, \ldots, x_n)$ *(making the appropriate substitutions for intermediate* y_i*) correctly classifies* $x_1, \ldots, x_n$.

(Proof omitted.)

Thus we get as our output a simple program 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 can't 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_l taking only variables $x_1, \ldots, x_n$ as inputs.

Corollary 1 *If we run the process described in Theorem 1, and then convert to a boolean circuit as described above, this process pac learns.*

3.3 Noise

We note here that the above procedure can be modified to tolerate a small amount of malicious noise [Valiant, 1985] or a somewhat larger amount of random labeling noise [Angluin and Laird, 1988], although the behavior we get from our algorithm is not quite as good as probably almost always useful.

4 Summary and conclusions

In this paper, we have shown how to learn complicated concepts by breaking them into subconcepts. The key idea we used was maintaining a list of all possible candidates (the "version spaces") for the correct subconcept, instead of simply picking some one candidate. For the purposes of this paper, we were concerned with the class 1CNF ∪ 1DNF, but our method is applicable to any polynomial size class. We expect that this particular method will prove to have other applications.

We believe this general approach is the philosophically correct 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. (See [Kugel, 1977].) 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 born fruit elsewhere in the field of inductive inference as well, in a new model of recursion theoretic inductive inference [Rivest and Sloan, 1988], and in a method for inference of simple assignment automata [Schapire, 1988].

Another contribution of this paper has been to introduce the notion of learning that is reliable, and probably almost always 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 would like to thank the anonymous readers whose comments contributed to both the clarity and content of this paper.

References

[Angluin and Laird, 1988] Dana Angluin and Philip Laird. Learning from noisy examples. *Machine Learning*, 2(4):343–370, 1988.

[Angluin *et al.*, 1987] Dana Angluin, William I. Gasarch, and Carl H. Smith. *Training Sequences.* Technical Report UMIACS-TR-87-37, University of Maryland Institute for Advanced Computer Studies, August 1987.

[Blumer *et al.*, 1986] Anselm Blumer, Andrzej Ehrenfeucht, David Haussler, and Manfred K. Warmuth. Classifying learnable geometric concepts with the Vapnik-Chervonenkis dimension. In *Proceedings of the Eighteenth Annual ACM Symposium on Theory of Computing*, pages 273–282, Berkeley, California, May 1986.

[Blumer *et al.*, 1987] Anselm Blumer, Andrzej Ehrenfeucht, David Haussler, and Manfred K. Warmuth. Occam's razor. *Information Processing Letters*, 24:377–380, April 1987.

[Haussler, 1986] David Haussler. Quantifying the inductive bias in concept learning. In *Proceedings AAAI-86*, pages 485–489, American Association for Artificial Intelligence, August 1986.

[Haussler, 1987] David Haussler. Bias, version spaces and Valiant's learning framework. In *Proceedings of the Fourth International Workshop on Machine Learning*, pages 324–336, University of California, Irvine, June 1987.

[Kearns *et al.*, 1987a] Michael Kearns, Ming Li, Leonard Pitt, and Leslie Valiant. On the learnability of boolean formulae. In *Proceedings of the Nineteenth Annual ACM Symposium on Theory of Computing*, pages 285–295, New York, New York, May 1987.

[Kearns *et al.*, 1987b] Michael Kearns, Ming Li, Leonard Pitt, and Leslie Valiant. Recent results on boolean concept learning. In *Proceedings of the Fourth International Workshop on Machine Learning*, pages 337–352, University of California, Irvine, June 1987.

[Kugel, 1977] Peter Kugel. Induction, pure and simple. *Information and Control*, 35:276–336, 1977.

[Laird *et al.*, 1984] John Laird, Paul Rosenbloom, and Allen Newell. Towards chunking as a general learning mechanism. In *Proceedings AAAI-84*, pages 188–192, August 1984.

[Laird *et al.*, 1986] John Laird, Paul Rosenbloom, and Allen Newell. Chunking in Soar: the anatomy of a general learning mechanism. *Machine Learning*, 1(1):11–46, 1986.

[Miller, 1956] G. Miller. The magic number seven, plus or minus two: some limits on our capacity for processing information. *Psychology Review*, 63:81–97, 1956.

[Mitchell, 1977] Thomas. M. Mitchell. Version spaces: a candidate elimination approach to rule learning. In *Proceedings IJCAI-77*, pages 305–310, International Joint Committee for Artificial Intelligence, Cambridge, Mass., August 1977.

[Pitt and Valiant, 1986] Leonard Pitt and Leslie G. Valiant. *Computational Limitations on Learning from Examples.* Technical Report, Harvard University Aiken Computation Laboratory, July 1986.

[Rivest and Sloan, 1988] Ronald L. Rivest and Robert Sloan. A new model for inductive inference. In Moshe Vardi, editor, *Proceedings of the Second Conference on Theoretical Aspects of Reasoning about Knowledge*, pages 13–27, Morgan Kaufmann, March 1988.

[Schapire, 1988] Robert E. Schapire. *Diversity-Based Inference of Finite Automata.* Master's thesis, MIT Lab. for Computer Science, May 1988.

[Valiant, 1984] Leslie G. Valiant. A theory of the learnable. *Communications of the ACM*, 27(11):1134–1142, November 1984.

[Valiant, 1985] Leslie G. Valiant. Learning disjunctions of conjunctions. In *Proceedings IJCAI-85*, pages 560–566, International Joint Committee for Artificial Intelligence, Morgan Kaufmann, August 1985.

[VanLehn, 1987] Kurt VanLehn. Learning one subprocedure per lesson. *Artificial Intelligence*, 31(1):1–40, January 1987.