Machine Learning 2: 229–246, 1987

Learning Decision Lists

RONALD L. RIVEST (RIVEST@THEORY.LCS.MIT.EDU)

Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, Massachusetts 02139, U.S.A.

(Received: December 29, 1986)

(Revised: August 5, 1987)

Keywords: Learning from examples, decision lists, Boolean formulae, polynomial-time identification.

Abstract. This paper introduces a new representation for Boolean functions, called *decision lists*, and shows that they are efficiently learnable from examples. More precisely, this result is established for k-DL – the set of decision lists with conjunctive clauses of size k at each decision. Since k-DL properly includes other well-known techniques for representing Boolean functions such as k-CNF (formulae in conjunctive normal form with at most k literals per clause), k-DNF (formulae in disjunctive normal form with at most k literals per term), and decision trees of depth k, our result strictly increases the set of functions that are known to be polynomially learnable, in the sense of Valiant (1984). Our proof is constructive: we present an algorithm that can efficiently construct an element of k-DL consistent with a given set of examples, if one exists.

1. Introduction

One goal of research in machine learning is to identify the largest possible class of concepts that are learnable from examples. In this paper we restrict our attention to concepts that can be defined in terms of a given set of n Boolean attributes; each assignment of **true** or **false** to the attributes can be classified as either a *positive* or a *negative* instance of the concept to be learned.

To make precise the notion of "learnable from examples," we adopt the definition of *polynomial learnability* pioneered by Valiant (1984) and studied by a number of authors (e.g., Blumer, Ehrenfeucht, Haussler, & Warmuth, 1986, 1987). In this model a concept to be learned is first selected arbitrarily from a prespecified class $\mathcal{F}$ of concepts. Then, a learning algorithm may request a number of positive and negative examples of the unknown concept. These examples are drawn at random according to fixed

but unknown probability distributions.[1] The learning algorithm then produces an answer – its hypothesis about the identity of the concept being learned. If the learning algorithm needs relatively few examples and relatively little running time in order to produce a hypothesis from $\mathcal{F}$ that is likely to be "close" to the correct answer, then the prespecified class of concepts is said to be *polynomially learnable*. (We make this notion more precise later.)

A number of classes of concepts are already known to be polynomially learnable. For example, Valiant (1984) has shown that k-CNF (the class of Boolean formulae in conjunctive normal form with at most k literals per clause) is polynomially learnable.

It is also known that certain classes of concepts are *not* polynomially learnable, unless $P = NP$. For example, Kearns, Li, Pitt, and Valiant (1987) show that Boolean threshold formulae and k-term-DNF (Boolean formulae in disjunctive normal form with k terms) for $k \geq 2$ are not polynomially learnable, unless $P = NP$.

It is a somewhat surprising fact, however, that it is possible for a class $\mathcal{F}$, which is not polynomially learnable, to be a subset of a class $\mathcal{F}'$, which *is* polynomially learnable. That is, enlarging the set of concepts may make learning easier, rather than harder. The reason is that it may be easier for the learning algorithm to produce a nearly correct answer from a rich set of alternatives than from a sparse set of possibilities. This paper provides another illustration of this fact.

The paper introduces the notion of a *decision list* and shows that k-DL – the set of decision lists with conjunctive clauses of size at most k at each decision – is polynomially learnable. This provides a strict enlargement of the functions known to be polynomially learnable: we show that k-DL generalizes the notions of k-CNF, k-DNF, and depth-k decision trees.

2. Definitions

2.1 Attributes (Boolean variables)

It is common in the machine learning literature to consider concepts defined on a set of objects, where the objects are described in terms of a set of *attribute-value* pairs. We adopt this convention here, with the understanding that each attribute is Boolean – its associated value will be either **true** or **false**. We assume that there are n Boolean attributes (or variables) to be considered, and we denote the set of such variables as

[1] Accordingly, this model is sometimes called *distribution-free learning*, since the results do not depend upon any particular assumptions about the probability distributions.

$V_n = \{x_1, x_2, \ldots, x_n\}$. Of course, in a particular learning situation, each variable will have a preassigned meaning. For example, if $n = 3$, x_1 might denote *heavy*, x_2 might denote *red*, and x_3 might denote *edible*. We assume that the given set of attributes is sufficient to precisely define the concepts of interest.

2.2 Object descriptions (assignments)

As is common in computer science, we identify **true** with 1, and **false** with 0. An *assignment* is a mapping from V_n to $\{0, 1\}$: an assignment gives (or assigns) each variable in V_n a value - either 1 (**true**) or 0 (**false**). We may also think of an assignment as a binary string of length n whose i-th bit is the value of x_i for that assignment. We let X_n denote the set $\{0, 1\}^n$ of all binary strings of length n. We may think of an assignment as a *description of an object*. For example, if $n = 3$ the assignment 011 (i.e., $x_1 = 0$, $x_2 = 1$, and $x_3 = 1$) might describe a non-heavy, red, edible object.

2.3 Representations of concepts (Boolean formulae)

Boolean formulae are often used as representations for concepts. This section reviews the definitions of some common classes of Boolean formulae.

A Boolean formula can be interpreted as a mapping from objects (assignments) into $\{0, 1\}$. If the formula is 1 (**true**) for an object, we say that the object is a *positive* example of the concept represented by the formula; otherwise we say that it is a *negative* example. The simplest Boolean formula is just a single variable. Each variable x_i is associated with two *literals*: x_i itself and its negation $\overline{x}_i$. An assignment naturally assigns a value to $\overline{x_i}$: $\overline{x}_i = 0$ if $x_i = 1$, otherwise $\overline{x}_i = 1$. We let $L_n = \{x_1, \overline{x}_1, \ldots, x_n, \overline{x}_n\}$ denote the set of $2n$ literals associated with variables in V_n.

A *term* is a conjunction (and) of literals and a *clause* is a disjunction (or) of literals. The truth of a term or clause at an assignment is a simple function of the truth of its constituents. For example, the term $x_1\overline{x}_3x_4$ is **true** if and only if x_3 is **false** and both x_1 and x_4 are **true**. Similarly, the disjunction $(\overline{x}_2 \vee x_3 \vee x_5)$ is **true** if and only if x_2 is **false** or either of x_3 or x_5 is **true**.

The *size* of a term or clause is the number of its literals. We let **true** be the unique term of size 0, and **false** be the unique clause of size 0. We let C_k^n denote the set of all terms (Conjunctions) of size at most k with literals drawn from L_n. Similarly, we let D_k^n denote the set of all clauses

(Disjunctions) of size at most k with literals drawn from L_n. Thus

$$|C_k^n| = |D_k^n| = \sum_{i=0}^{k} \binom{2n}{i} = O(n^k).$$

For any fixed k, C_k^n and D_k^n have sizes that are polynomial functions of n.

A Boolean formula is in *conjunctive normal form* (CNF) if it is the conjunction (and) of clauses. We define k-CNF to be the set of Boolean formulae in conjunctive normal form where each clause contains at most k literals. For example, the following formula is in 3-CNF:

$$(x_1 \vee x_4 \vee \overline{x}_5)(x_2 \vee \overline{x}_3)(x_4 \vee x_6)(x_7).$$

Similarly, a Boolean formula is in *disjunctive normal form* (DNF) if it is the disjunction (or) of terms. We define k-DNF to be the set of Boolean formulae in disjunctive normal form where each term contains at most k literals. For example, the following formula is in 2-DNF:

$$x_1x_3 \vee \overline{x}_1x_4 \vee \overline{x}_3 \vee x_2x_7.$$

We define *k-term-DNF* to be the set of all Boolean formulae in disjunctive normal form containing at most k terms, and *k-clause-CNF* to be the set of Boolean formulae in conjunctive normal form with at most k clauses. (The previous examples were from 4-clause-CNF and 4-term-DNF, respectively.) Extending our definition of *size* for terms and clauses, we define the *size* of a Boolean formula to be the number of literals it contains.

Each Boolean formula defines a corresponding Boolean function from X_n to $\{0,1\}$ in a natural manner. In general, we do not distinguish between formulae and the Boolean functions they represent. That is, we may interchangeably view a formula as a syntactic object (so that we might consider its size) or a Boolean function (so that we may consider its value at a particular point $\mathbf{x} \in X_n$).

The four classes of functions defined above (k-CNF, k-DNF, k-clause-CNF, and k-term-DNF) are not independent, as we now show. By duality (DeMorgan's Rules) k-DNF is the set of negations of functions in k-CNF. For example, the negation of the 2-CNF formula:

$$(\overline{x}_1 \vee \overline{x}_2)(\overline{x}_3 \vee x_4)(x_5)$$

is the 2-DNF formula:

$$x_1x_2 \vee x_3\overline{x}_4 \vee \overline{x}_5.$$

Similarly, k-term-DNF is the set of negations of functions in k-clause-CNF.

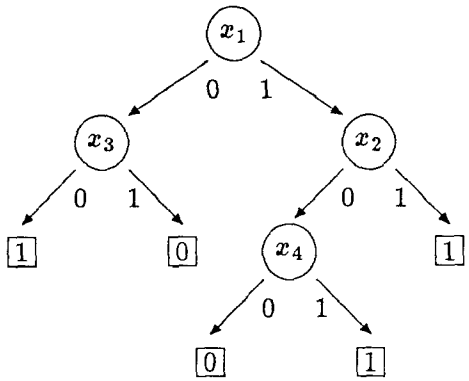

Figure 1. A decision tree.

Also, by distributivity, k-term-DNF is a subset of k-CNF. For example, we can rewrite the 3-term-DNF formula:

$$x_1x_2 \vee x_3\overline{x}_4 \vee \overline{x}_5$$

as the 3-CNF formula:

$$(x_1 \vee x_3 \vee \overline{x}_5)(x_1 \vee \overline{x}_4 \vee \overline{x}_5)(x_2 \vee x_3 \vee \overline{x}_5)(x_2 \vee \overline{x}_4 \vee \overline{x}_5).$$

Similarly, k-clause-CNF is a subset of k-DNF.

2.4 Decision trees

Another useful way to represent a concept is as a *decision tree.* A decision tree is a binary tree where each internal node is labelled with a variable, and each leaf is labelled with 0 or 1. The *depth* of a decision tree is the length of the longest path from the root to a leaf.

Each decision tree defines a Boolean function as follows. An assignment determines a unique path from the root to a leaf: at each internal node the left (respectively right) edge to a child is taken if the variable named at that internal node is 0 (respectively 1) in the assignment. The value of the function at the assignment is the value at the leaf reached.

Figure 1 shows an example of a decision tree of depth 3; the concept so defined is equivalent to the Boolean formula:

$$\overline{x}_1\overline{x}_3 \vee x_1\overline{x}_2x_4 \vee x_1x_2.$$

Table 1. Truth table for example decision list function.

$x_1x_2x_3$	x_4x_5 00	01	10	11
000	1	1	(0)	(0)
001	(0)	(0)	(0)	(0)
010	1	1	(0)	1
011	(0)	1	(0)	1
100	0	0	0	0
101	(0)	(0)	(0)	(0)
110	0	0	0	0
111	(0)	(0)	(0)	(0)

We let k-DT denote the set of Boolean functions defined by decision trees of depth at most k.

2.5 Additional notation

When we wish to emphasize the number of variables upon which a class of functions depends, we will indicate this in parentheses after the class name, as in k-CNF(n), k-DNF(n), or k-DT(n). We use $\lg(n)$ to denote the base-2 logarithm of n and $\ln(n)$ to denote the natural logarithm of n. We may denote a function of one argument as $f(\cdot)$, a function of two arguments as $f(\cdot,\cdot)$, and so on.

2.6 Decision lists

We now introduce a new technique for representing concepts: *decision lists*. Decision lists are a strict generalization of the concept representation techniques described in the previous section. And they are, as we shall see, easy to learn from examples.

A *decision list* is a list L of pairs

$$(f_1, v_1), \ldots, (f_r, v_r) \tag{1}$$

where each f_j is a term in C_k^n, each v_i is a value in $\{0,1\}$, and the last function f_r is the constant function **true**. A decision list L defines a Boolean function as follows: for any assignment $\mathbf{x} \in X_n$, $L(\mathbf{x})$ is defined to be equal to v_j where j is the *least* index such that $f_j(\mathbf{x}) = 1$. (Such an item always exists, since the last function is always **true**.)

We may think of a decision list as an extended "**if** – **then** – **elseif** – ... **else** –" rule. Or we may think of a decision list as defining a concept

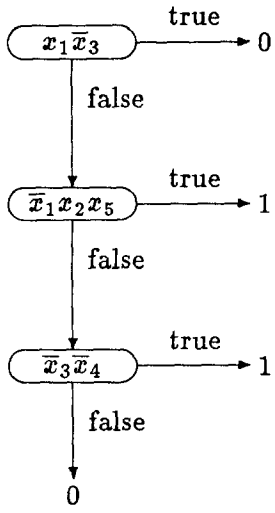

Figure 2. Diagram of decision list corresponding to Table 1.

by giving the general pattern with exceptions. The exceptions correspond to the early items in the decision list, whereas the more general patterns correspond to the later items. For example, consider the decision list

$$(x_1\bar{x}_3, 0), (\bar{x}_1x_2x_5, 1), (\bar{x}_3\bar{x}_4, 1), (\textbf{true}, 0) \tag{2}$$

which has the truth table given in Table 1.[2] For example, $L(0,0,0,0,1) = 1$; this value is specified by the third pair in the decision list. The zeros in parentheses indicate the positions filled by the last (default) item. This decision list may also be diagrammed as in Figure 2. (No decision box is given for the last item, since it is always **true**.)

We remark that decision lists always form a "linear" sequence of decisions; the "true" branch of a decision always points to either **1** or **0**, never to another decision node. Compared to decision trees, decision lists have a simpler structure, but the complexity of the decisions allowed at a node is greater.

We let k-DL denote the set of all Boolean functions defined by decision lists, where each function in the list is a term of size at most k. As k increases, the set k-DL becomes increasingly expressive.

[2]The format of this table is that of a standard Karnaugh map as used in logical design; three of the input variables are used to select the row, and the remaining two variables are used to select the column, of the desired value.

We note that - unlike k-DNF or k-CNF - k-DL is closed under complementation (negation): if f is in k-DL, then so is the complement of f. The proof is trivial: merely negate the second element of each pair in the decision list, or - equivalently - change all the 1's to 0's and vice-versa in the leaves of the decision list.

The items of a decision lists are essentially the same as the production rules studied by many people. For example, Quinlan (1987) has examined production rules as a means of simplifying decision trees. One may think of decision lists as a "linearly ordered" set of production rules.

3. Relationship of decision lists to other classes of functions

In this section we prove that decision lists are more expressive than the other forms of concept representation discussed in section 2. We first examine the relationship of k-DL to k-CNF and k-DNF. We see that the sequential **if - then - else** combinator used to construct decision lists is more powerful than the associative and commutative $\vee$ and $\wedge$ operators.

3.1 Decision lists generalize k-CNF and k-DNF formulae

Theorem 1 *For $0 < k < n$, k-CNF(n) and k-DNF(n) are proper subsets of k-DL(n).*

Proof: To see that k-DNF is a subset of k-DL, observe that we can easily transform a given formula in k-DNF into a decision list representing the same function. Each term of a DNF formula is converted into an item of a decision list with value 1. For example, the term $x_1\overline{x}_2$ would yield the item $(x_1\overline{x}_2, 1)$. The last (default) item of the decision list has the value 0.

To see that k-CNF is a subset of k-DL, recall that every k-CNF formula is the complement of some k-DNF formula, and that k-DL is closed under complementation.

The subset relations are proper, since neither k-CNF nor k-DNF is a subset of the other if $0 < k < n$. ■

Corollary 1 *For $0 < k < n$, k-clause-CNF(n) and k-term-DNF(n) are proper subsets of k-DL(n).*

Proof: This follows from the previous theorem and the previously noted facts that k-clause-CNF(n) is a subset of k-DNF(n), and that k-term-DNF(n) is a subset of k-CNF(n). ■

We can strengthen the above result to show that not only is k-DL a generalization of k-CNF and k-DNF; it is also a generalization of their

union (i.e., the set of functions that have either a k-CNF representation or a k-DNF representation) for $n > 2$.

Theorem 2 *For $0 < k < n$ and $n > 2$, (k-CNF(n) $\bigcup$ k-DNF(n)) is a proper subset of k-DL(n).*

Proof: Our proof makes essential use of the notion of the *prime implicants* of a Boolean function f. An *implicant* of f is any term that logically implies f. A *prime implicant* is an implicant all of whose literals are essential: if any literal is dropped it is no longer an implicant. For example, for the function represented by Table 1, $\bar{x}_1\bar{x}_3\bar{x}_4$ is a prime implicant. We will also use the fact that if a function f has a prime implicant of size t, then f has no k-DNF representation if $k < t$. It follows from this that if the negation of a function f has a prime implicant of size t, then f has no k-CNF representation if $k < t$.

We break our proof into the cases $k = 1$ and $1 < k < n$. For $k = 1$, we claim that the following decision list from 1-DL(n) represents a function f that is not in 1-CNF(n) $\bigcup$ 1-DNF(n):

$$(x_1, 0), (x_2, 1), (x_3, 1), (\textbf{true}, 0). \tag{3}$$

To see that f is not in 1-DNF(n), note that $\bar{x}_1x_2$ is a prime implicant of f. Similarly, to see that f is not in 1-CNF(n); note that $\bar{x}_2\bar{x}_3$ is a prime implicant of the negation of f.

For $1 < k < n$, we consider the decision list:

$$(\bar{x}_1x_{k+1}, 1), (x_1x_{k+1}, 0), (x_1 \oplus x_2 \oplus \cdots \oplus x_k, 1)(\textbf{true}, 0). \tag{4}$$

This is not a proper decision list according to our definition, since the third item does not have a term as its first part, but rather the "exclusive-or" of $x_1, \ldots, x_k$, where "$\oplus$" denotes the exclusive-or (mod 2 addition) operator. However, it is simple to convert the function into a proper k-DL representation, since we are only taking the exclusive-or of k variables. The conversion would involve replacing the third item above with a sequence of 2^{k-1} items whose first parts are all possible terms of length k involving variables $x_1, \ldots, x_k$ having an odd number of unnegated variables, and whose second parts are always 1.

The function f represented above has the prime implicant

$$x_1\bar{x}_2\bar{x}_3 \cdots \bar{x}_k\bar{x}_{k+1}, \tag{5}$$

so that it has no k-DNF representation. Similarly, its negation $\bar{f}$ has the prime implicant

$$\bar{x}_1\bar{x}_2\bar{x}_3 \cdots \bar{x}_k\bar{x}_{k+1}, \tag{6}$$

so that f has no k-CNF representation either. ■

3.2 Decision lists generalize decision trees

Theorem 3 *k-DT(n) is a proper subset of k-DL(n), for $0 < k < n$.*

Proof: By theorem 1 it suffices to observe that

$$k\text{-DT}(n) \subset k\text{-DNF}(n), \tag{7}$$

since one can create a k-DNF(n) formula equivalent to a given decision tree in k-DT(n) by creating one term for each leaf labelled with "1". One could also create a k-CNF(n) formula equivalent to the given tree by creating one clause for each leaf labelled with "0"; this additional fact suffices to prove that $k\text{-DT}(n) \subset k\text{-CNF}(n) \cap k\text{-DNF}(n)$. ■

4. Polynomial learnability

We are interested in algorithms for learning concepts from examples, where the notion of "concept" is identified with that of "Boolean function." In this section we review the notion of the "polynomial learnability" of a class of concepts, as pioneered by Valiant (1984).

4.1 Examples and samples

An *example* is a description of an object, together with its classification as either a positive or a negative instance of the concept f being learned. More formally, an example of f is a pair $(\mathbf{x}, v)$ where $\mathbf{x}$ is an assignment and $v \in \{0, 1\}$. The example is considered to be a *positive* example if $v = 1$, and a *negative* example otherwise.

We will be interested in learning algorithms which produce as answers Boolean functions that do not contradict any of the given training data; in this case we say that the answer produced is *consistent* with the training data. More formally, we say that a Boolean function f is *consistent* with an example $(\mathbf{x}, v)$ if $f(\mathbf{x}) = v$.

A *sample* is a set of examples. We say that a function is consistent with a sample if it is consistent with each example in the sample. If f is a Boolean function, then a *sample of f* is any sample consistent with f.

4.2 Hypothesis space

The learning algorithms we consider will have a *hypothesis space* $\mathcal{F}$ - a set of concepts that contains the concept being learned. The goal of the learning algorithm is to learn *which* concept in $\mathcal{F}$ is actually being used to classify the examples it has seen.

We partition the hypothesis space $\mathcal{F}$ according to the number n of variables upon which the concept depends. Typically the learning algorithm will be told at the beginning that the concept to be learned depends on a given set V_n of variables. Let F_n denote the subset of $\mathcal{F}$ that depends on V_n, for any integer $n \geq 1$, so that $\mathcal{F} = \bigcup_{n=1}^{\infty} F_n$. (We assume that each function in F depends on a finite set of variables.)

Of course, the difficulty of learning a concept that has been selected from F_n will depend on the size $|F_n|$ of F_n. We let

$$\lambda_n = \lg(|F_n|) \tag{8}$$

denote the logarithm of the size of F_n; this may be viewed as the number of bits needed to write down an arbitrary element of F_n, using an optimal encoding.

We are interested in characterizing when a class $\mathcal{F}$ of functions is "easily learnable from examples." Clearly this is a characteristic of the set $\mathcal{F}$ and not of any particular member of $\mathcal{F}$; identifying any *particular* function from examples would be trivial if there were no alternatives.

4.3 Discussion – learnability from examples

Informally, to say that a class $\mathcal{F}$ of functions is "learnable from examples" is to say that there exists a corresponding learning algorithm A such that for any function f in $\mathcal{F}$, as A is given more and more examples consistent with f, A will converge upon f as the function it has "learned" from the examples.

However, any class $\mathcal{F}$ of Boolean functions is clearly learnable from examples in principle, since each function $f \in F_n$ is defined on a finite domain (the 2^n assignments X_n). The algorithm A need only wait until it sees an example corresponding to each element of X_n; then A knows the complete truth table for f.

Thus, the interesting question is not whether a class $\mathcal{F}$ is learnable from examples, but whether it is "easily" learnable from examples. This involves two aspects, since a learning algorithm has *two* resources available: examples and computational time. We say that a learning algorithm is *economical* if it requires *few examples* to distinguish the correct answer from other alternatives, and say that it is *efficient* if it requires *little computational effort* to identify the correct answer from a sufficient number of examples. We would like our learning algorithms to be both economical and efficient.

We observe that each example provides at most one bit of information to the learning algorithm. Since λ_n bits are required to describe a typical

concept in F_n, we see that any learning algorithm will have to examine at least λ_n examples on the average, if it is to make a correct identification.

4.4 Polynomial-time identifiability

Our first formal notion of learnability is "polynomial-time identifiability," which captures the notion of efficiency but not economy. We say that $\mathcal{F}$ is *polynomial-time identifiable* if there exists an *identification algorithm* A and a polynomial $p(\lambda, m)$ such that algorithm A, when given the integer n and a sample S of size m, will produce in time $p(\lambda_n, m)$ a function $f \in F_n$ consistent with S (if one exists, otherwise it will output "impossible").

To say that a set F is polynomial-time identifiable is not necessarily very interesting. For example, the class DL of all decision lists is clearly polynomial-time identifiable: it is easy to construct a decision list consistent with a given sample of size m in time polynomial in λ_n and m, by turning each example into an item of the decision list. These decision lists are uninteresting because they are as long as the data itself. A "good" identification procedure should produce an answer that is considerably shorter than the data, if possible. Answers which are not short may be poor predictors for future data (see Pearl, 1978, or Rissanen, 1978).

4.5 Polynomial learnability

Valiant (1984) introduced a more elegant notion of learnability, which makes explicit the notion that the concept acquired by the learner should closely approximate the concept being taught, in this sense that the acquired concept should perform well on new data. To do so, he presumed the existence of an arbitrary probability distribution P_n defined on the set X_n; the learning algorithm's examples are drawn from X_n according to the probability distribution P_n. When it has produced its answer f, one measures the quality of its answer by measuring the probability that a new example drawn according to P_n will be incorrectly classified by f.

To formalize this, we define the *discrepancy* $f \odot g$ between two functions f and g in F_n as the probability (according to P_n) that f and g differ:

$$f \odot g = \sum_{\mathbf{x} | f(\mathbf{x}) \neq g(\mathbf{x})} P_n(\mathbf{x}). \tag{9}$$

The discrepancy will always be a number between 0 and 1, inclusive. We also define the *error* of a function g to be $f \odot g$, where f is the "correct" function. A hypothesis g with error at most ϵ will incorrectly classify at most a fraction ϵ of the objects drawn according to the probability distribution P_n.

We would like our learning algorithm A to be able to produce an answer whose error is less than an arbitrary prespecified *accuracy parameter* ϵ. (Here $0 < \epsilon \leq 1$.) However, we cannot expect A to be sufficiently accurate always, since it may suffer from "bad luck" in drawing examples according to P_n. Thus, we will also supply A with a *confidence parameter* δ (here $0 < \delta \leq 1$), and require that the probability that A produces an accurate answer be at least $1 - \delta$. Of course, as ϵ and δ approach zero, we should expect A to require more examples and computation time.

We now make this formal. We say that a set of Boolean functions $\mathcal{F}$ is *polynomially learnable* if there exists an algorithm A and a polynomial $s(\cdot, \cdot, \cdot)$ such that for all n, ϵ, and δ, all probability distributions P_n on X_n, and all functions $f \in F_n$, A will with probability at least $1 - \delta$, when given a sample of f of size $m = s(\lambda_n, \frac{1}{\epsilon}, \frac{1}{\delta})$ drawn according to P_n, output a $g \in F_n$ such that $g \odot f < \epsilon$. Furthermore, A's running time is polynomially bounded in n and m.

Thus, if $\mathcal{F}$ is polynomially learnable, then it is possible to get "close" to the right answer in a reasonable amount of time, independent of the probability distribution that is used to generate the examples. Here, as is only fair, the *same* probability distribution P_n that is used to generate the examples is used to judge the error of the program's answer.

There are several variations on the above definition in the literature. In Valiant's original paper (1984), the probability distribution P_n was constrained so that only positive examples were shown to the learning algorithm. In other papers (e.g., Kearns et al., 1987), the algorithm may choose whether it would like to see a positive example or a negative example. (In this case there are two unknown probability distributions, one for each type of example.) The formulation we have adopted here, using a single probability distribution, is identical to that used by Blumer et al. (1986, 1987). It is possible to prove that the model we use here is in fact formally equivalent to the two-distribution model used by Kearns et al. (1987); we use the single-distribution model because it is simpler.

A number of important classes of Boolean functions have been proved to be polynomially learnable. For example, Valiant (1984) proved that k-CNF is polynomially learnable from positive examples only. It follows that k-DNF is also polynomially learnable (using an algorithm that makes use of the negative examples only). Pitt and Valiant (1986) have shown that k-CNF $\bigcup$ k-DNF is polynomially learnable, but that k-term-DNF and k-clause-CNF are not polynomially learnable unless $P = NP$. (To identify a k-term-DNF formula that is consistent with the examples is NP-hard in general, so the problem is *efficiency*. However, since k-term-DNF is a subset of k-CNF, it is easy to learn a k-CNF formula equivalent to the unknown k-term-DNF formula. Enlarging the set of formulae helps.)

4.6 Proving polynomial learnability

Although proving that a class $\mathcal{F}$ of formulae is polynomially learnable can be difficult in general, Blumer et al. (1986, 1987) have developed some elegant methods for doing so. They show how one can prove polynomial learnability by proving polynomial-time identifiability, if the class $\mathcal{F}$ is polynomial-sized (i.e., if $\lambda_n = O(n^t)$ for some t). For the reader's convenience we repeat a key theorem and its proof from Blumer et al. (1987).

Theorem 4 *Given a function $f \in F_n$ and a sample S of f of size m drawn according to P_n, the probability is at most*

$$|F_n|(1-\epsilon)^m$$

that there exists a hypothesis $g \in F_n$ such that

- *the error of g is greater than ϵ, and*
- *g is consistent with S.*

Proof: If g is a single hypothesis in F_n of error greater than ϵ, the chance that g is consistent with a randomly drawn sample S of size m is less than $(1-\epsilon)^m$. Since F_n has $|F_n|$ members, the chance that there exists *any* member of F_n satisying both conditions above is at most $|F_n|(1-\epsilon)^m$. ■

Since

$$m > \frac{1}{\epsilon}(\ln(|F_n|) + \ln(\frac{1}{\delta})) \tag{10}$$

implies that

$$|F_n|(1-\epsilon)^m \leq \delta, \tag{11}$$

if m satisfies equation (10) the chance is less than δ that a hypothesis in F_n which is consistent with S will turn out to have error greater than ϵ. Thus to achieve the bounds in the definition of polynomial learnability, the learning algorithm need only examine enough examples (as given by equation (10)), and then return any hypothesis consistent with the sample. We note that if $\mathcal{F}$ is polynomial-sized, then the bound of equation (10) is bounded by a polynomial in n, $\frac{1}{\epsilon}$, and $\frac{1}{\delta}$. Thus, *if $\mathcal{F}$ is polynomial-sized and polynomial-time identifiable, then it is polynomially learnable in the sense of Valiant* (see Blumer et al. (1987) for a fuller discussion).

The previous theorem shows that polynomial-sized classes of functions are *economically* learnable, in the sense that after a polynomial number of examples are seen, *any* hypothesis consistent with the observed data will be "close enough" to the correct answer with high probability. Furthermore, polynomial-time identifiability implies *efficient* learnability: such a

consistent answer can be found quickly. Together, these imply learnability in the sense of Valiant.

5. k-DL is polynomially learnable

From the previous section we conclude that to prove that k-DL is polynomially learnable in the sense of Valiant, it suffices to prove that k-DL is polynomial-sized and that it is polynomial-time identifiable. The following two subsections provide these proofs.

5.1 k-DL is polynomial-sized

To show that k-DL is polynomial-sized, we begin by noting that

$$|k\text{-DL}(n)| = O(3^{|C_k^n|}(|C_k^n|)!). \tag{12}$$

This follows because each term in C_k^n can either be missing, paired with 0, or paired with 1 in the decision list, and the order of the elements of the decision list is arbitrary. This implies that

$$\lg(|k\text{-DL}(n)|) = O(n^t) \tag{13}$$

for some constant t; i.e., k-DL is polynomial-sized. (Recall that $\lg(x)$ is the base-two logarithm of x.)

5.2 k-DL is polynomially identifiable

In this subsection we show that a "greedy" algorithm works for constructing decision lists consistent with a given set of data. We begin with the trivial observation that *if a function is consistent with a sample S, then it is consistent with any subset of S.*

The algorithm proceeds by identifying the elements of the decision list in order. That is to say, *we may select as the first element of the decision list any element of $C_k^n \times \{0,1\}$ that is consistent with the given set of data.* The algorithm can then proceed to delete from the given data set any data points that are explained by the chosen element of the decision list, and then to construct the remainder of the decision list in the same way to be consistent with the remaining data. Choosing a particular function in C_k^n "doesn't hurt", in the sense that explaining a certain subset of the data with this first item does not prevent or interfere with the ability of the remaining elements of C_k^n to explain the remaining data. If the original sample could be represented by an element of k-DL, then so can the remaining portion of the sample. Thus the greedy algorithm will find a function k-DL consistent with the sample if and only if such a function exists.

We now specify our algorithm more precisely. Let S denote a non-empty input sample of an unknown function $f \in F_n$, and let j denote an auxiliary variable that is initially 1.

1. If S is empty, halt.
2. Examine each term in C_k^n in turn until a term t is found such that all examples in S which make t true are of the same type (i.e., all positive examples or all negative examples). If no such term can be found, report "The given sample S is not consistent with any k-DL function" and halt. Otherwise proceed to step 3.
3. Let T denote those examples in S which make t true. Let $v = 1$ if T is a set of positive examples; otherwise let $v = 0$. Output the item (t, v) as the j-th item of the decision list being constructed, and increment j by one.
4. Set S to $S - T$, and return to step 1.

That this algorithm is polynomial-time is easily seen from the fact that the size of C_k^n is polynomial in n (for fixed k). This completes our proof that k-DL is polynomially learnable.

5.3 Variations and extensions

It is interesting to note that the proof here really only depends on the fact that the size of C_k^n is polynomial in n; we could replace C_k^n with any family of functions of polynomial size as long as it is possible to evaluate the functions quickly for any given example.

There is certainly some flexibility in step 2 of the algorithm that could be used to advantage, in an attempt to yield a shortest possible decision list. For example, one could choose t so that it "explained" the largest number of examples (i.e., so the corresponding set T was as large as possible). This would be analogous to the strategy used by Quinlan's (1986) ID3 system for constructing decision trees. However, it is NP-hard in general to compute the shortest decision list consistent with a given sample S; this result can be derived from the results of Masek (1976) or Kearns et al. (1987).

6. Open problems

Before closing, we should briefly describe four open problems regarding the learning of decision lists.

Problem 1: *Can the functions in k-DL be learned from* positive *examples only, in the sense of Pitt and Valiant (1986)?* In this model the learning algorithm is not allowed to see any negative examples; instead it has

access to a supply of positive examples, drawn according to the unknown probability distribution restricted to the set of positive examples.

Problem 2: *Can one devise an efficient incremental (i.e., on-line) version of our algorithm for learning functions in k-DL?* In this model the algorithm receives examples one at a time, and after each example must produce an updated hypothesis consistent with all examples seen so far.

Problem 3: *Can the functions in k-DL be learned efficiently when the supplied examples are "noisy"?* In this model the classifications of the given examples may be erroneous with some small probability β: we may suppose that the algorithm is given β as part of its input.

Problem 4: *Can one learn functions in k-DL efficiently when the examples may only partially describe an object?* A *partial assignment* is a mapping from V_n to $\{0, 1, *\}$ where "*" denotes "unspecified" or "unknown". Quinlan (1986) has discussed this problem of "unknown attribute values," as well as techniques for coping with it when trying to learn a decision tree. We may also think of a partial assignment as a string in $\{0, 1, *\}^n$. A partial assignment is a partial specification of an object — some attributes are not specified. A partial example is a partial description of an object (i.e., a partial assignment), together with its classification. More formally, a *partial example* is a pair $(\mathbf{x}, v)$ where $\mathbf{x}$ is a partial assignment and $v \in \{0, 1\}$. An assignment $\mathbf{x}$ is said to be an *extension* of partial assignment $\mathbf{y}$ if $x_i = y_i$ whenever $y_i \neq$ "*". We say that f is consistent with a partial example if $f(\mathbf{y}) = v$ for all assignments y that are extensions of x.

We say that $\mathcal{F}$ is *polynomial-time identifiable from partial examples* if the definition of polynomial identifiability holds even when S may contain partial examples. Our previous greedy algorithm does not apply in a straightforward way to learning from partial examples, and we leave it as an open question as to whether k-DL is polynomial-time identifiable from partial examples.

7. Conclusions

We have presented a new class of Boolean concepts — the class k-DL: concepts representable by decision lists with a term of size at most k at each decision. Decision lists are interpreted sequentially, and the *first* term in the decision list that is satisfied by the input determines how the input is to be classified.

We have shown that decision lists represent a strict generalization of the classes of Boolean functions previously known to be polynomial-time identifiable from examples and polynomially learnable in the sense of Valiant.

In particular, k-DL generalizes the formula classes k-CNF and k-DNF, as well as decision trees of depth k.

The algorithm for determining an element of k-DL that is consistent with a given set of examples is particularly simple; it is basically a greedy algorithm that constructs the decision list one element at a time until all of the examples have been accounted for. When learning a concept in the class k-DL, our algorithm will, with high probability, produce as output an answer that is approximately correct; the running time is polynomial in the number of input variables, the parameter k, and the inverses of the accuracy and confidence levels that are desired.

Acknowledgements

This paper was prepared with support from NSF grant DCR-8607494, ARO Grant DAAL03-86-K-0171, and a grant from the Siemens Corporation. We wish to thank David Haussler, Barbara Moore, and the referees for their very helpful comments on an earlier version of this paper.

References

Blumer, A., Ehrenfeucht, A., Haussler, D., & Warmuth, M. K. (1986). Classifying learnable geometric concepts with the Vapnik-Chervonenkis dimension. *Proceedings of the Eighteenth Annual ACM Symposium on Theory of Computing* (pp. 273–282). Berkeley, CA.

Blumer, A., Ehrenfeucht, A., Haussler, D., & Warmuth, M. K. (1987). Occam's Razor. *Information Processing Letters*, *24*, 377–380.

Kearns, M., Li, M., Pitt, L., & Valiant, L. (1987). On the learnability of Boolean formulae. *Proceedings of the Nineteenth Annual ACM Symposium on Theory of Computing* (pp. 285–295). New York, NY.

Masek, W. (1976). Some NP-complete set-covering problems. Unpublished manuscript, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA.

Pearl, J. (1978). On the connection between the complexity and credibility of inferred models. *Journal of General Systems*, *4*, 255–264.

Pitt, L., & Valiant, L. G. (1986). *Computational limitations on learning from examples* (Technical Report). Cambridge, MA: Harvard University, Aiken Computation Laboratory.

Quinlan, J. R. (1986). Induction of decision trees. *Machine Learning*, *1*, 81–106.

Quinlan, J. R. (1987). Generating production rules from decision trees. *Proceedings of the Tenth International Joint Conference on Artificial Intelligence* (pp. 304–307). Milan, Italy: Morgan Kaufmann.

Rissanen, J. (1978). Modeling by shortest data description. *Automatica*, *14*, 465–471.

Valiant, L. G. (1984). A theory of the learnable. *Communications of the ACM*, *27*, 1134–1142.