Machine Learning, 5, 71-99 (1990)

Boolean Feature Discovery in Empirical Learning

GIULIA PAGALLO (giulia@saturn.ucsc.edu)
DAVID HAUSSLER (haussler@saturn.ucsc.edu)
Department of Computer and Information Sciences, University of California, Santa Cruz CA 95064

Editor: Paul Rosenbloom

Abstract. We investigate the problem of learning Boolean functions with a short DNF representation using decision trees as a concept description language. Unfortunately, Boolean concepts with a short description may not have a small decision tree representation when the tests at the nodes are limited to the primitive attributes. This representational shortcoming may be overcome by using Boolean features at the decision nodes. We present two new methods that adaptively introduce relevant features while learning a decision tree from examples. We show empirically that these methods outperform a standard decision tree algorithm for learning small random DNF functions when the examples are drawn at random from the uniform distribution.

Keywords. concept learning, dynamic bias, DNF functions, decision trees, decision lists.

1. Introduction

In the standard model of learning from examples the *target concept* is a subset of the given *instance space*, and the teacher presents to the learner a set of instances from the instance space which are labelled positive if they are elements of the target concept (*positive examples*) and negative otherwise (*negative examples*) [Carbonell *et al.*, 1983]. The task of the learning algorithm is to induce a rule that will accurately predict the class label of future instances from the instance space.

In this setting, the interchange of information and knowledge between the learning algorithm and the teacher is based on the instance and concept description languages [Carbonell *et al.*, 1983]. The first one is used by the teacher to present the examples, while the second one is used by the learning algorithm to represent its hypothesis for the target concept.

A simple way to describe an instance is by a measurement vector on a set of predefined attributes and a class label. An attribute comprises a (possibly) relevant characteristic of the target concept, and an attribute measurement reveals the state of the variable in an observed instance. Such a description is called *attribute based.* For example, a Boolean attribute is used to indicate the presence or absence of a quality in the instance.

In an attribute-based learning system with a fixed concept representation language the grain of the attributes determines the complexity of the classification rule. Appropriate high-level attributes facilitate concept representation, but if the attributes are low-level primitives for a target concept, the learning system will often have to compile a complex rule to express its hypothesis [Flann and Dietterich, 1986]. Such complex rules are usually difficult to find, and when they are found, they are often opaque to humans.

The creation of appropriate attributes is not an easy task. Quoting Breiman *et al.*: "Construction of features is a powerful tool for increasing both accuracy and understanding of structure, particularly in high dimensional problems. The selection of features is an art guided by the analyst's intuition and preliminary exploration of the data" ([Breiman *et al.*, 1984], page 140).

One approach to partially automate the formation of high-level attributes for learning is to extend the primitive set of attributes with all Boolean combinations of the primitive attributes of a fixed type and size. For example, Valiant [1985] essentially uses conjunctions up to a fixed size as *high-level* attributes to learn a restricted class of DNF formulae (a DNF formula is a Boolean expression in Disjunctive Normal Form). Rivest uses the same high-level attributes to learn decision lists [1987]. Breiman *et al.* use limited conjunctions of the attributes to find meaningful decision trees in medical and chemical domains [1984]. Usually, only a small number of the features proposed this way are meaningful for learning, unless domain knowledge is used to select relevant combinations for the problem at hand.

Another approach is to let the learning algorithm *adaptively* define its own features while learning. The capability of a learning system to adaptively enlarge the initial attribute set is called *dynamic bias* [Utgoff and Mitchell, 1982]. General purpose learning systems with this capability have been proposed, for example, for incremental learning [Schlimmer, 1986] or for learning sets of Horn clauses [Muggleton, 1987].

In this paper we present two learning methods that dynamically modify the initial attribute bias for learning decision trees. Traditional learning algorithms choose the test to place at a node from a fixed set of attributes. In the simplest case, this set coincides with the set of primitive attributes [Breiman *et al.*, 1984, Quinlan, 1986], and it has also been extended to include conjunctions [Breiman *et al.*, 1984, Rivest, 1987].

Our first learning method (called FRINGE) builds a decision tree using primitive attributes, analyzes this tree to find candidates for useful higher level attributes, then redescribes the examples using the new attributes in addition to the primitive attributes and builds a decision tree again. This process is iterated until no new candidate attributes are found.

Our second method uses as concept description language, a restricted type of decision tree called a decision list. The method uses a top-down sample refinement strategy to form a list of conjunctions of the primitive literals. A greedy heuristic chooses the next literal in each conjunction according to an attribute merit function. A sample refinement method, called *separate and conquer,* models the conditional choices made by the greedy heuristic. The subdivision process presented in [Haussler, 1988] resembles one step of this method. We present two specific algorithms based on this strategy: GREEDY3 and GROVE.

Our separate and conquer algorithms use essentially the same control structure as the CN2 algorithm [Clark and Niblett, 1989] with beam size one (the beam size is a user defined parameter to control the amount of parallel search that the algorithm performs). However, we use different attribute merit functions and a different strategy to avoid overfitting the data. Clark and Niblett use a significance measure to stop growing a conjunction. By contrast, we use a simpler stopping rule, but after learning we prune the hypothesis using an independent data set. Pruning methods have been successfully applied to decision tree algorithms [Breiman *et al.*, 1984, Quinlan, 1987b].

Both methods (separate and conquer and CN2) use a stepwise approach to form a conjunction to place at the root of a decision list. A similar technique is used by the CART method [Breiman *et al.*, 1984], to construct a conjunction (or disjunction) to place at a node of a decision tree. However, the decision list heuristics search for a conjunction that covers a large portion of the examples that belong to the same class, while the decision tree heuristic searches for a conjunction that divides the examples into sets with a small class mixture. Each heuristic is appropriate for its own structure.

The outline of this paper is as follows: In Section 2 we introduce some notation and terminology; in Section 3 we review the standard decision tree method and describe our implementation (called REDWOOD), which is a new hybrid of diverse ideas. In Section 4 we discuss a shortcoming of the decision tree representation for DNF functions. We call this representational problem the *replication problem* because of the duplication of the test in the tree. Then, we discuss the impact of the replication problem on a learning system based on decision trees. In Section 5, we describe the FRINGE algorithm and its implementation. In Section 6 we review the concept of decision lists and the learning algorithm proposed by Rivest. In Section 7 we present the top down greedy heuristic to learn decision lists, and present the algorithms GREEDY3 and GROVE based on this method. In Section 8 we present some preliminary results on the performance of the algorithms in three Boolean domains, and show that they compare very favorable to the decision tree approach. In Section 9 we summarize our results and discuss future research directions.

2. Definitions

In this section we introduce the notation and definitions that formalize the basic notions we use in this paper.

Let n be a positive integer and let V be a set of n Boolean attributes. We denote by 0 and 1 the two values of a Boolean attribute and by $\mathbf{B}_n$ the set of n dimensional Boolean vectors. The set $\mathbf{B}_n$ defines all the *instances* that can be described by the attributes in V, and it is called the *instance space.*

A *concept* C is a subset of $\mathbf{B}_n$. The elements of C are the positive examples of the concept, while the elements in $\mathbf{B}_n$ but not in C are the negative examples. We denote an example as the ordered pair $<\mathbf{x}, c>$ where $\mathbf{x} \in \mathbf{B}_n$ and $c \in \{+, -\}$. A set of examples is called a *sample.*

A *hypothesis* is also a subset of $\mathbf{B}_n$. Let S be a sample of a concept C. Let S_+ be the set of instances in S that are labelled positive, and let S_- be the set of instances that are labelled negative. We say that H is a hypothesis of C *consistent* with S if $S_+ \subseteq H$ and $S_- \cap H = 0$. The *error* between a hypothesis H and a concept C is the symmetric difference of the two sets. We use as a measure of the error the cardinality of this symmetric difference divided by 2^n (that is, we assume a uniform distribution over the instance space.)

Let f be a Boolean function, f: $\mathbf{B}_n \rightarrow \{0, 1\}$. The function f defines the concept $C = \{\mathbf{x} \in \mathbf{B}_n \,|\, f(\mathbf{x}) = 1\}$. Conversely, a concept C defines a Boolean function: the Boolean indicator function of the set. In this paper, we will refer to a concept by its Boolean indicator function.

Finally, we introduce some terminology to refer to combinations of Boolean attributes. A *literal* is an attribute or its complement. A *term* is a conjunction of literals, a *clause* is a disjunction of literals. In general, a *feature* is any Boolean combination of the attributes obtained by applying Boolean operators to the primitive attributes. We use the term *variable* to refer to a primitive attribute or to a feature. We use •, + and − to denote the Boolean operators *and, or* and *not* respectively. The *size* of a feature is the number of literals in its (smallest) expression. For example the feature, $x_1 \bullet \bar{x}_2 + x_3$ has size 3.

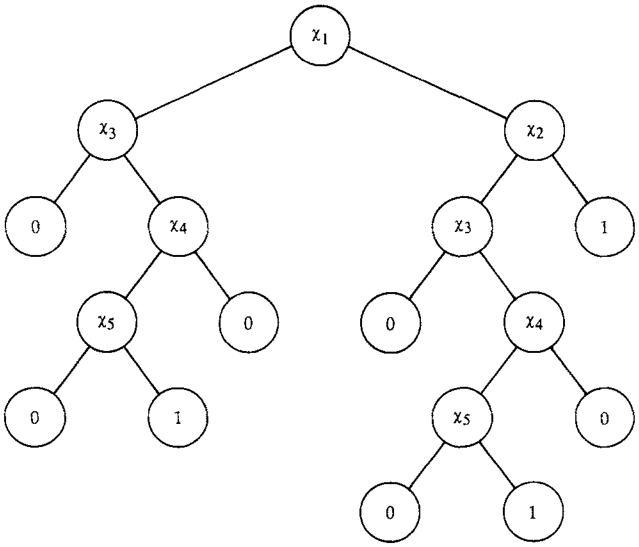

Figure 1. Smallest decision tree for $x_1 \cdot x_2 + x_3 \cdot \bar{x}_4 \cdot x_5$.

3. Decision trees

The goal of a system that learns from examples is to induce from a relatively small number of examples a hypothesis that will accurately predict the class of new instances. One popular methodology has been developed using decision trees as the concept representation language. An excellent introduction to this topic is given in [Quinlan, 1986], and a more complete discussion on the subject is given in [Breiman *et al.*, 1984]. In this section we review the salient aspects of the rule generation procedure to learn Boolean functions from examples using decision trees.

In a Boolean decision tree, an internal node defines a binary test on a variable. For example, the root node of the tree in Figure 1 defines the test "is $x_1 = 1$?". Each edge represents an outcome of the test. We adopt the convention that the left edge represents the negative outcome, and the right one represents the positive outcome. The label of a leaf (positive or negative) represents the class that is assigned to any example that reaches this node.

It is easy to see that a Boolean decision tree represents a Boolean function. Each path from the root node to a positive leaf defines a term in the function as follows: initially, the term is empty; then, for each node in the path concatenate the attribute at the node to the current term if the path proceeds to the right of the node, otherwise concatenate the negation of the attribute to the current term. The function represented is the disjunction of all the terms generated in this manner.

For learning however, we are interested in the inverse problem: how to generate a decision tree that represents a Boolean function specified by a partial truth table. The common procedure is based on successive subdivisions of the sample. This process aims to discover sizable subsets of the sample that belong to the same class. The tree structure is derived by a top-down refinement process on the sample.

Initially, we begin with an empty decision tree and a sample set. A test is applied to determine the best attribute to use to partition the sample. In Section 3.1 we discuss a criterion to measure the *merit* of an attribute with respect to the subdivision process. For the time being, assume that the best attribute can be determined using some statistical tests on the examples. Then, the best attribute is placed at the root of the tree, and the examples are partitioned according to their value on that attribute. Each subset is assigned to one subtree of the root node, and the procedure is applied recursively. The subdivision process, on one branch of the tree, ends when all the examples associated with the leaf of that branch belong to the same class. This leaf is labelled with the common class of these examples.

A statistical test requires a sample of significant size. However, the samples may be quite small towards the end of the subdivision process. Furthermore, the presence of noise in the data may force an unnecessary refinement of the examples (fitting the noise), which may actually decrease the tree classification performance on independent examples. Because of this, it is common to prune back the tree created by the above recursive subdivision process in an attempt to improve the accuracy of the hypothesis. Quinlan [1987b] proposes a strategy to search for the smallest subtree of the initial tree with the highest accuracy on an independent data set (*pruning set*). The method is appropriately called *Reduced Error Pruning* and it proceeds as follows.

For every nonleaf subtree S of the initial tree T, we examine the change in classification error on the pruning set that would occur if the subtree S were replaced by a leaf. We assume this leaf is labelled with the class of the majority of the pruning examples classified by S. If the accuracy of the pruned tree is the same or increases, the subtree S is removed. The process ends when no further improvement is possible.

3.1. Mutual information

We now return to the question of measuring the merit of an attribute with respect to the subdivision process. The measure of attribute merit we use is based on the reduction in uncertainty about the example class if we test the attribute value. This measure is formalized by the information theoretic concept of mutual information. To state this notion more precisely, we need to introduce some concepts and notation.

Suppose Y is a discrete random variable with range R_y. For any $y \in R_y$, let $p(y) = P\{Y = y\}$. The *entropy of* Y is defined as

$$H(Y) = \sum_{y \in R_y} p(y) \log \left(\frac{1}{p(y)} \right).$$

The entropy $H(Y)$ measures the information provided by an observation of Y, or the amount of uncertainty about Y. Our next goal is to define for a pair of discrete random variables X and Y, a measure for the uncertainty about X after we observe Y. Let x be a value in the range of X and y a value in the range of Y. If we denote by

$$p(y|x) = P\{Y = y|X = x\},$$
$$p(y, x) = P\{Y = y, X = x\}$$

then we define the *conditional entropy* as

$$H(Y|X) = \sum_{x \in R_X,\, y \in R_y} p(x, y) \log \left(\frac{1}{p(y|x)} \right).$$

For a pair of discrete random variables this quantity represents the amount of uncertainty about Y after X has been observed. Now since, $H(Y)$ represents our uncertainty about Y before we observe X, and $H(Y|X)$ represents our uncertainty afterwards, the difference $H(Y) - H(Y|X)$ represents the amount of information about Y given by X. This quantity is called *mutual information,* and it is defined as

$$I(Y; X) = H(Y) - H(Y|X).$$

In our context, we can regard a Boolean variable as a discrete random variable X, and the class as a discrete random variable Y. Then, the mutual information of the attribute X and class Y measures the amount of information we gain about the class value of an example after we test the variable. As in [Quinlan, 1986] we take the mutual information as our attribute merit function. Hence, the attribute chosen at each subdivision step is the one that maximizes the mutual information.

The central quantities in the computation of the mutual information are $p(y)$, $p(x, y)$, and $p(y|x)$. We review how to estimate them from a sample S. We denote by

$$S_y = \{<\mathbf{x}, c> | c = y\}$$
$$S_{x,y} = \{<\mathbf{x}, c> | c = y, \mathbf{x} \textit{ has value } x \textit{ for attribute } X\},$$

and we use the hat notation to indicate empirical probability estimates (for example, $\hat{p}$ represents the empirical estimate for p).

In decision theory, the expression $p(y)$ is called the *prior probability* of class y, and we will denote it by π_y. It reflects our *a priori* belief that an object will belong to class y. If the sample S is indicative of the class frequency, the prior probability that an example will have class y is estimated from the relative frequency of the classes as:

$$\pi_y = \frac{|S_y|}{|S|}.$$

In this case the priors are usually called *data priors.* However, the sample may not reflect the distribution of the classes that we shall observe after training, or the sample may not indicate correctly the frequency of the classes. In this case, the knowledge of the decision maker can be taken into account by setting the value of the prior probabilities. For example, when all classes are equally likely, we should choose *uniform priors.* In the two class case the uniform priors would be $\{1/2, 1/2\}$.

The quantities $p(x, y)$ and $p(y|x)$ are estimated from the conditional probability $p(x|y)$ and the priors. The conditional probability that the attribute X has value x given that class variable Y has value y, is defined as

$$p(x|y) = \frac{p(x, y)}{p(y)}. \tag{1}$$

Since

$$\hat{p}(x|y) = \frac{|S_{x,y}|}{|S_y|},$$

we obtain from Equation (1) and the above expression that

$$\hat{p}(x, y) = \pi_y \bullet \frac{|S_{x,y}|}{|S_y|}.$$

The unconditional probability $p(x)$ is the sum over y of the marginal probabilities $p(x, y)$, therefore

$$\hat{p}(x) = \sum_{y \in R_y} \pi_y \bullet \frac{|S_{x,y}|}{|S_y|}.$$

Finally

$$\hat{p}(y|x) = \frac{\hat{p}(x, y)}{\hat{p}(x)}.$$

3.2. REDWOOD

We implemented the heuristics described above in the *REDWOOD* system. The central design aspects of the system are summarized as follows:

- class priors can be specified by the user or estimated from the data;
- the subdivision process uses the criteria of maximum mutual information to choose the best attribute;
- the sample refinement process terminates when the examples belong to the same class or the examples have identical attribute values but different class label (this may occur for example, in the presence o^f noise);
- the tree is pruned using the Reduced Error Pruning Method.

The system is a new hybrid of existing ideas. ID3 [Quinlan, 1986] uses an attribute merit function based on mutual information. We borrow from the CART system [Breiman *et al.*, 1984] the flexibility given by the class priors. We borrow from Quinlan [1987b] the pruning method.

4. The replication problem

A decision tree representation of a Boolean function with a small DNF description has a peculiar structure: the same sequence of decision tests leading to a positive leaf is replicated in the tree. To illustrate the problem, consider a Boolean function with a two term representation, and to simplify the example assume that the attributes in the two terms are disjoint. The smallest decision tree for the function has the following structure. Each decision test in the rightmost path of the tree is an attribute of the shortest term in the formula. The path leads to a positive leaf and it corresponds to the truth setting for the first term. The left branch from each of the nodes is a partial truth assignment of the attributes that falsifies the shortest term. Then, to complete the function representation, we have to duplicate the sequence of decisions that determine the truth setting of the second term on the left branch of each one of the nodes. We call this representational shortcoming the *replication problem.* In general, there are shared attributes in the terms so some replications are not present, but the duplication pattern does occur quite frequently. Figure 1 illustrates the replication problem in the smallest decision tree for the Boolean function $x_1 \cdot x_2 + x_3 \cdot \bar{x}_4 \cdot x_5$. Observe that this tree has six negative leaves which is equal to the product of the length of the terms in the DNF formula. We show in Appendix A that any decision tree equivalent to a μDNF formula (a DNF formula where each attribute appears only once) has size greater or equal than the product of the length of the terms in the smallest equivalent DNF formula.

Due to the replication problem, while learning a decision tree, the partition strategy has to fragment the examples of a term into many subsets. This causes the algorithm to require a large number of examples in order to ensure that the subsets are large enough to give accurate probability estimates; otherwise the subdivision process branches incorrectly or terminates prematurely.

One way to solve the replication problem is to use conjunctions of the primitive literals at the decision nodes. To illustrate this solution, consider the two term formula discussed above. Assume we use the conjunction $x_1 \cdot x_2$ as the test at the root node. Now, the right branch leads to a positive leaf that corresponds to the examples that satisfy the first term. The left branch leads to a representation of the second term (see Figure 2). We can represent this term by single attributes as before, but now the replication problem has disappeared. If, in addition, we have a feature for $x_3 \cdot \bar{x}_4 \cdot x_5$ we can obtain the even more compact representation shown on the right.

Rivest [1987] and Quinlan [1987a] have investigated different versions of this solution by using conjunctions to represent a Boolean function. Rivest uses conjunctions up to a certain size as tests in a novel representation for Boolean functions called *decision lists.* Informally, we can think of a decision list as a decision tree where each decision variable is a term, and each internal node has at most one non-leaf child. For example, both representations in Figure 2 are a decision list description of the formula. In practice, the algorithm he presents have to limit *a priori* the maximum size of the conjunctions that can be considered while learning. If this value is not appropriate for the problem at hand Rivest's algorithm will fail to find a solution (see Section 6). Quinlan's method generates a collection of production rules from a decision tree. Then, each rule is simplified by removing its irrelevant literals, and the set of rules is reduced using a greedy heuristic. Hence, the method attempts to detect and remove the duplication patterns. However, this approach still requires a large number of examples to build an accurate initial tree.

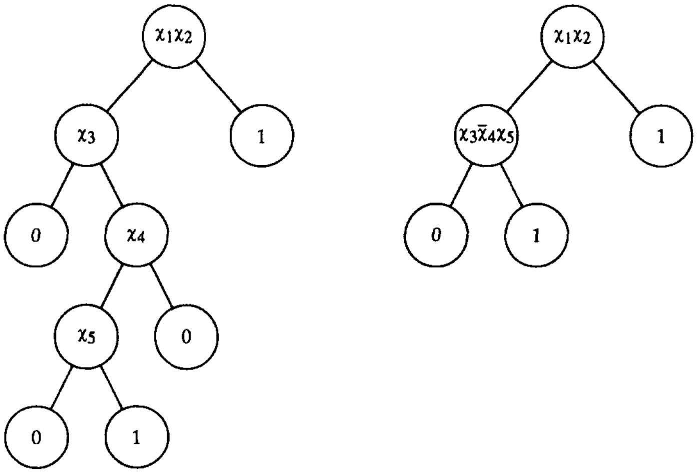

Figure 2. Two representations for $x_1 \cdot x_2 + x_3 \cdot \bar{x}_4 \cdot x_5$ using features.

To solve these problems, we adopt an approach in which the learning algorithm defines *adaptively* the type or size of the features to use in the representation. In the following sections we give three learning algorithms based on this idea.

5. Learning features

In Section 3 we discussed the formation of a hypothesis from examples using a decision tree representation where each test was defined by a single attribute. In this section we present a new algorithm that dynamically creates and uses features as test variables in the tree.

The feature set is defined through the following iterative learning method. The algorithm begins with a set V of primitive attributes, and creates a decision tree for a set of examples, choosing its decision variables from the set V. Then, a *find-features* procedure generates new features as Boolean combinations of the variables that occur near the fringe of the tree. We describe this heuristic in more detail below. The set of new features is added to the variable set, and the execution of the decision tree algorithm and the find-features procedure is repeated. We will call a single execution of both processes an *iteration*. The iterative process terminates when no new features can be added to the variable set, or a maximum number of variables is reached. Table 1 presents a specific algorithm called FRINGE that implements this method. We present a *find-features* procedure in Table 2.

The find-features procedure defines a feature for each positive leaf in the decision tree of depth at least two, counting the root at depth zero. For each such leaf, the procedure examines the last two decision nodes in the path from the root of the tree to the leaf. It forms a feature that is the conjunction of two literals, one for each of the decision nodes. If the path to the leaf proceeds to the right from a decision node, then the literal associated

Table 1. FRINGE algorithm.

```
input : $V$ an attribute set and $S$ a sample.
        $M$ a positive integer such that $M \geq | V |$.

initialize $k := 0$, and $V_1 := V$
repeat
    $k := k + 1$
    form decision tree $T_k$ using sample $S$ described using variables in $V_k$
    $F :=$ find-features( $T_k$ )
    $V_{k+1} := V_k \cup F$
until ( $V_{k+1} = V_k$ or $| V_{k+1} | \geq M$ )
output $T_k$, $V_k$ and halt.
```

Table 2. Find-features procedure.

```
input : a decision tree $T$

$F$ := empty set
for each positive leaf $l$ at depth $\geq 2$ in $T$
    initialize: $feature$ := true
    let $p$ and $g$ be the parent and grandparent nodes of $l$
    let $v_p$ and $v_g$ be their test variables
    if $l$ is on the right subtree of $p$ then
      form conjunction of $v_p$ and $feature$
    else
      form conjunction of $\bar{v}_p$ and $feature$
    if $l$ is on the right subtree of $g$ then
      form conjunction of $v_g$ and $feature$
    else
      form conjunction of $\bar{v}_g$ and $feature$
    add $feature$ to $F$
end
return $F$
```

with this node is just the variable in the decision node, otherwise it is the negation of this variable. Thus the find-features procedure applies the term formation rule used to obtain a DNF formula from a decision tree (see Section 3), but only to the last two variables in the path to each positive leaf.

For example, the find-features procedure generates the features $\bar{x}_4 \cdot x_5$, $\bar{x}_4 \cdot x_5$ and $x_1 \cdot x_2$ (one feature appears twice) for the decision tree in Figure 1. They correspond to the positive leaves from left to right.

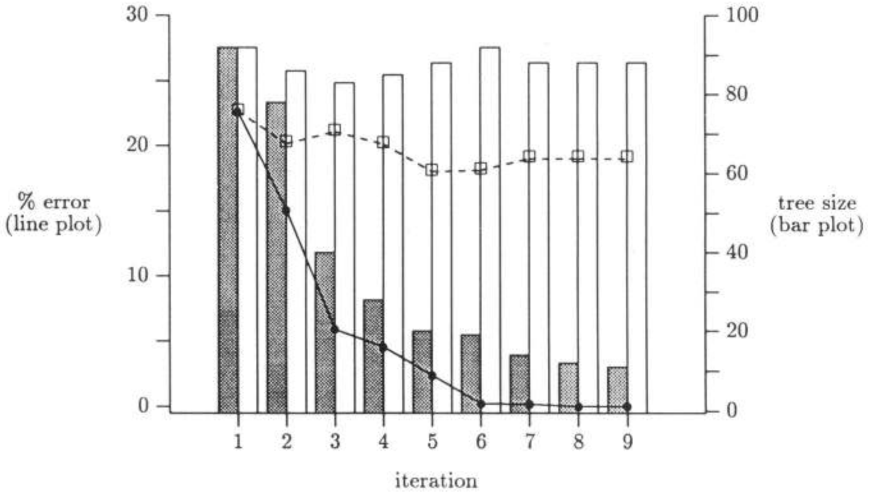

Figure 3. Performance comparison between FRINGE and random heuristic as defined by: • solid line: % error FRINGE; □ dashed line: % error random; gray bar: decision tree size for FRINGE; white bar: decision tree size for random. The left scale measures the % error, the right scale measures the hypothesis size in number of internal nodes.

In each iteration, the find-features procedure forms very simple features; namely conjunctions of two variables or their negations. The creation of longer terms or more complex features occurs through the iterative process. Initially, the variable set contains only the attributes. After the k-th iteration, the variable set may contain features of size up to 2^k. For example, after the second iteration, a feature is either a term of size up to 4 or a conjunction of two clauses, each of size 1 or 2. Negated features include clauses of size up to 4, and disjunctions of two terms of size 1 or 2. In the limit, the find-features procedure has the capability of generating any Boolean function of the literals, since the *negation* and *and* operators are a complete set of Boolean operators.

Figure 3 illustrates the learning performance results for a single execution of FRINGE on a small random DNF (dnf4, see Section 8) and how it compares to a strategy where features are proposed at random. The random proposal heuristic works as follows. A feature is defined as the conjunction of two variables chosen at random from the current variable set. In each iteration, this heuristic adds the same number of features as FRINGE does. So, both methods work with the same number of variables in each iteration.

The graph shows the change in percentage error and in size of the tree as the iteration proceeds. The error is measured on a sample drawn independently from the training sample (see Section 8).

The shape of the error and the size graph for FRINGE are typical. After a few iterations a very accurate hypothesis is developed, and the remaining steps are used to find a more concise hypothesis by introducing more meaningful features. After nine iterations, in this case, the process ends because no new features are found. Note that the random guess heuristic was not successful at all. The small fluctuations occur because by chance some of the features defined were useful.

Another interesting aspect of FRINGE's learning behavior is the form of the final hypothesis. In this example, the smallest DNF description of the target concept has 10 terms and the final decision tree has 10 internal nodes. The decision variable at each node is a term of the target concept. A similar behavior was observed for all DNF concepts with a small description when all terms have approximately the same size. For DNF concepts with terms of highly varying size, the final hypothesis tends not to include the longer terms. However, the hypothesis is still quite accurate because the positive examples that only satisfy these terms are very rare when the examples are drawn from the uniform distribution.

FRINGE solves the replication problem by forming meaningful features through an iterative process and using them at the decision nodes. Unfortunately, it is a difficult algorithm to analyze. The observation that the hypotheses generated by FRINGE for random DNF were decision lists suggested to us a second approach to overcome the replication problem. This approach is described in Section 7 below. First we review the decision list representation.

6. Decision lists

In a recent paper, Rivest [1987] introduced a new representation for Boolean concepts called *decision lists*, and showed that they are efficiently learnable from examples. More precisely, he showed that the class k-DL, the set of decision lists with terms of length at most k, are learnable in the sense of Valiant [1984].

Let V be a set of n Boolean variables and L its set of literals. For any non-negative integer k, we denote by $T(n, k)$ the set of conjunctions of literals drawn from L of size at most k.

A decision list DL is a list of pairs

$$(T_1, c_1), \ldots ,(T_r, c_r),$$

where each T_i is an element of $T(n, k)$, each c_i is a value in $\{0, 1\}$, and the last term T_r is the constant term *true*. The last pair in the list is called the default pair. We call a pair $(T, 1)$ a *positive* pair, and a pair $(T, 0)$ a *negative pair.*

A decision list defines a Boolean function as follows. An n dimensional Boolean vector x satisfies at least one term in the list. Let j be the index of the first such term. The value of the function on x is set to the value c_j. For example, the following list

$$(x_1x_2, 0), (x_1\bar{x}_2, 1), (\bar{x}_1\bar{x}_2, 0), (\mathit{true}, 1).$$

represents the *exclusive-or* function for two variables.

It is easily verified that the class k-DL generalizes the class k-DNF of all DNF formulae with at most k literals per term ([Rivest, 1987], see also Section 7.1).

The algorithm, presented by Rivest, to learn a decision list from examples requires, as input, the number of attributes n, a set of positive and negative examples from a target concept, and a value for k to define the length of the longest term allowed. The algorithm will find a k-DL description that is consistent with the examples if such description exists, otherwise it will return failure.

The algorithm begins with an empty decision list *DL* and a set of training examples *S*. If all the examples in *S* belong to the same class *c*, it adds the node (*true*, *c*) to the list and terminates. Otherwise, it selects a term *T* from the set *T*(*n*, *k*) that covers a portion of examples that all have the same class *c* and it adds the pair (*T*, *c*) to the list. The examples covered by the term T are removed from S, and the process is repeated until all the examples are accounted for. If no term can be found then the algorithm returns failure.

Selecting an appropriate value for *k* presents practical difficulties. A small value of *k* may prevent the algorithm from finding a solution, while a large value of *k* may render the algorithm inefficient, since the size of the search space grows exponentially on *k*.

In the next section we propose a new paradigm to learn a decision list from examples that dynamically defines the value of *k* for the problem at hand. The method is based on a greedy approach to determine the literals in every term.

7. Separate and conquer

The smallest decision list representation for a concept is a decision list that uses the fewest literals. Unfortunately, the problem of finding the smallest decision list that is consistent with a sample is NP-hard [Rivest, 1987]. We can avoid this problem by choosing the literals in a decision list top-down by a greedy heuristic. In a top-down approach the greedy heuristic chooses the best literal for the sample set given that a set of literals has already been chosen. A simple way to model this conditional choice is by sample subdivision.

In this section, we present an algorithm that forms a decision list from examples using a top-down greedy approach. First, we discuss the sample refinement strategy, called *separate and conquer*, that defines the control structure of the algorithm, and then we discuss two possible greedy functions to choose the literals.

Table 3 presents a specific implementation of the separate and conquer learning algorithm. The algorithm begins with a non-empty set of examples *S*, an empty auxiliary set called the *pot*, and an empty decision list *DL*. If all the examples in *S* belong to the same class, the default pair with the common class of these examples is appended to the list and the learning process ends. Otherwise, the algorithm finds the first term using the greedy heuristic. A *select-literal* function assigns to each literal a value of merit with respect to the subdivision process, and then chooses the literal with the highest merit. We discuss in detail two select-literal functions in the next two sections. After a literal is selected, the examples that have a zero entry for that literal (that is, do not satisfy the literal) are removed from the set *S* and are added to the pot. The pot contains the examples that do not satisfy the term, and that will be used later to learn the next terms. The literal selection and sample subdivision process terminates when all the examples in *S* share a common class label. At this point, the pair formed by the current term and the common class of these examples is appended to the list *DL*. Before learning the next term, the examples covered by this term are discarded, and the examples in the pot are moved to the set *S*, while the pot becomes empty. Then, the algorithm proceeds as described above.

After learning, the hypothesis is pruned to attempt to increase its classification performance on new instances of the concept. In the next sections, we discuss two greedy methods to find a sublist of the hypothesis *DL* that has at least the same accuracy as *DL* on an

Table 3. Control structure for a separate and conquer algorithm.

```
input : a non-empty sample $S$

initialize $DL$ := empty list
while not all examples in $S$ belong to the same class do
    initialize $pot$ := empty set and $term$ := $true$
    while not all examples in $S$ belong to the same class do
        $x$ := select-literal ($S$, $term$)
        form conjunction of $x$ and $term$
        remove from $S$ all examples with value 0 for $x$ and add them to $pot$
    end
    $c$ := class of remaining examples in $S$
    append pair ($term$, $c$) to $DL$
    $S$ := $pot$
end
append default pair with the class common to all examples in $S$
$DL$ := prune-list ($DL$)
output $DL$ and halt.
```

independent sample set. We can reduce the size of a decision list by removing one of its pairs or by removing a literal from one of its terms. The first heuristic decides which pairs of *DL* to keep and which pairs to remove from the list, while the second heuristic decides which literals to delete from the list.

To end this section, let us contrast separate and conquer and the divide and conquer method used to learn decision trees. The divide and conquer strategy recursively subdivides a sample into parts until each part contains examples with a common class label. Instead, the separate and conquer strategy recursively subdivides only one part at the time, until all the examples in this part share the same class label. Then, the remaining examples are gathered together before subdividing them. This helps to keep the sample size larger, so that more accurate statistics are available to guide the learning algorithm.

7.1. GREEDY3

In this section, we focus on the use of the separate and conquer paradigm to learn a DNF description for the target concept. A decision list where the default pair has class 0, and the remaining pairs have class 1 is equivalent to the DNF representation formed by the disjunction of all terms in the list except the default one.

We use the *validity* measure to implement the *select-literal* function for GREEDY3 (a.k.a. greedy literal, greedy term, greedy prune). The validity of a given literal is defined as the probability that an instance is a member of a class or category given that it has that cue (that is, satisfies the literal). The concept of literal, or cue, validity has long been a part of theories in perceptual categorization [Beach, 1964]. The basic idea is that organisms are sensitive

Table 4. Select-literal function for GREEDY3.

```
let $L$ be the set of primitive literals
input : a sample $S$, and a term $t$

let $T$ be the set of all literals in $t$
$L := L - T$
for each $l$ in $L$
    $m_l := \hat{p}\ (+ \mid l = 1)$
choose $\hat{l}$ such that $m_{\hat{l}}$ is maximized
return $\hat{l}$
```

to cues that allow them to make correct categorization. For example, in the special case that a literal has unitary validity, the single literal is enough to make a correct classification.

While learning, the separate and conquer algorithm maintains in the set S the examples that satisfy the current term. Then the select-literal function estimates the validity of each literal not already in the term using the set S and selects one with highest conditional probability. Table 4 presents an implementation of the select-literal function for GREEDY3. Recall that we use the hat notation to indicate a probability estimate.

Let us analyze briefly what literals the select-literal function would choose when the target function is a small DNF and examples are drawn from the uniform distribution. In this case it turns out that there are two reasons for a literal to have high-validity: it belongs to a short term or it belongs to many terms (or both). The first case leads the algorithm to identify a term in the formula. In fact, once it chooses a literal in a short term the validity of the remaining literals in the term increases, making them very likely to be selected in the next steps.

The second case may lead to *crossover terms.* A crossover term is a conjunction of literals that belong to different terms. If the function selects a literal that belongs to many terms, the validity of the literals that accompany it may not increase enough to assure that any of them will be chosen in the next step. We discuss an example of crossover term formation in Section 8.4.

In our experiments, the addition of crossover terms to the hypothesis does not significantly degrade the classification performance for random small DNF, and they do not occur at all when the target concept is a *μDNF* formula (a DNF where a literal appears at most once). However, they are a problem for functions with a strong term interaction, like the multiplexor functions.

Some of the crossover terms can be eliminated from the hypothesis by pruning. In the second phase of the algorithm, we aim to find the smallest subset of the term set that has the best classification performance on an independent data set. Unfortunately, this problem is NP-hard, so we find an approximate solution using the following greedy strategy to reduce the length of the list.

We denote by P a sample of the target concept drawn independently from the learning sample, and by DL we denote a hypothesis such that all pairs except the default one are

Table 5. Prune strategy for GREEDY3.

```
input : a decision list $DL$ with one or more positive pairs and a negative default pair

$\widehat{DL} := (true, 0)$
$R :=$ set of all pairs in $DL$ but default one
$\hat{E} :=$ number of errors $\widehat{DL}$ makes in classifying $P$
repeat
    for each $r$ in $R$ do
        create list $DL_r$ by inserting $r$ into $\widehat{DL}$
        $E_r :=$ number of errors $DL_r$ makes in classifying $P$
    end
    choose $\hat{r}$ such that $E_{\hat{r}}$ is minimized
    $\Delta E := E_{\hat{r}} - \hat{E}$
    if $(\Delta E \leq 0)$ then
        $\widehat{DL} := DL_{\hat{r}}$
        $\hat{E} := E_{\hat{r}}$
        $R := R - \{ \hat{r} \}$
    end
until ( $\Delta E \geq 0$ or $R$ is empty )
output $\widehat{DL}$ and halt.
```

positive. The algorithm begins with a decision list $\widehat{DL}$ that contains only the default pair (*true*, 0) and the set R that contains all of the pairs in DL but the default one. For each pair r in R we evaluate the change in classification error that would occur if the pair r is inserted into the list $\widehat{DL}$. The pair may be added anywhere before the default pair, since the relative order of the positive pairs is not relevant. The pair that reduces the error the most is added to the list and is removed from R. The process is repeated until no pair improves the accuracy of $\widehat{DL}$ or the set R becomes empty. Table 5 presents a specific implementation of the algorithm.

7.2. GROVE

We now turn to the problem of learning a general decision list using the separate and conquer paradigm. In a general decision list any pair may have a positive or negative class label, as opposed to a DNF like decision list where all the pairs except the default one have a positive class label.

Recall that the separate and conquer algorithm maintains in the set S the examples that satisfy the current term and in the *pot* the examples that will be processed later. When all the examples in S share the same class label the current term is complete. At this point, if all the examples in the *pot* belong to the same class, they define the class of the default pair, and the learning process ends. Intuitively, a good literal to choose should reduce the class mixture in S without increasing too much the class mixture in the *pot*. This is the basic idea of the select-literal function we present in this section to learn a general decision list.

Table 6. Select-literal function for GROVE.

```
input : sample $S$

for each attribute $x$
    $m_x$ := mutual information between class and attribute x
choose $\hat{x}$ such $m_{\hat{x}}$ is maximized
for $i = 0, 1$
    let $S_i$ be the set of examples in $S$ with value $i$ for $\hat{x}$
if( $\hat{H}_{S_0}(Y) \leq \hat{H}_{S_1}(Y)$ ) then
    return $\hat{x}$ negated
else
    return $\hat{x}$
```

Recall that the class mixture of a set can be measured by the estimated entropy of the class variable using the examples in the set (see Section 3.1). We define a new select-literal function as follows. First, we choose an attribute that maximizes the mutual information between the class and the attribute for *S*. Then, we subdivide the examples in *S* into two sets according to their value for that attribute. Second, if the set that contains the examples with value 1 for the attribute has the lowest entropy we select the attribute, otherwise we select its negation. Table 6 presents an implementation for this select-literal function.

In terms of the examples, the first choice minimizes the average class mixture of the two parts into which we divide the set *S*. The second choice establishes that we keep the purer set, and we throw the other set into the pot.

In the rest of this section, we present a greedy pruning strategy to find a sublist of the learned list *DL* that has at least the same accuracy as *DL* on an independent sample *P*. Table 7 presents a specific implementation of the following method.

We begin by measuring the classification error *E* of the decision list *DL* on the sample *P*. Then, for each pair *r* in the list, we consider the sublist DL_r obtained by pruning the pair *r* and by modifying the class of the default pair. To prune a pair we proceed as follows: If the term of the pair has size 1 then the pair is removed from the list, otherwise we prune the last literal in the term. Since the last literal is selected using a smaller sample size than any other literal in the same term, it is the most prone to statistical errors. After pruning the pair, the examples in *P* are reclassified according to the new list, and the class of the default pair is set to be the class of the majority of the examples that filter down to the default pair. This change is made to reduce the error rate of the pruned list over the sample *P*. Clearly, other strategies of adjusting the class labels could also be used.

Once the list DL_r is generated, we compute the classification error over the sample *P*. Finally, we choose to prune the pair, if any, that reduces the classification error the most. The process is repeated until no pair improves the classification rate or the list contains only the default pair.

Table 7. Pruning algorithm for GROVE.

```
input: a decision list $DL$, and a sample $P$

E := number of errors that $DL$ makes in classifying $P$
repeat
    for each pair r in $DL$ do
        $DL_r$ := *prune-pair* (r, $DL$, $P$)
        $E_r$ := number of errors that $DL_r$ makes in classifying $P$
    end
    choose $\hat{r}$ such that $E_{\hat{r}}$ is minimized
    $\Delta E := E_{\hat{r}} - E$
    if ( $\Delta E \leq 0$ ) then
        $DL := DL_{\hat{r}}$
        $E := E_{\hat{r}}$
    end
until ($\Delta E \geq 0$ or $DL$ has only the default pair)
output $DL$ and halt.


*prune-pair* (r, $DL$, $P$)

let $r = (t, c)$
if $size(t) = 1$ then
    remove pair r from list $DL$
else
    remove last literal from $t$
$D$ := set of examples in $P$ that satisfy only default pair
$dc$ := class of the majority of examples in $D$
assign $(true, dc)$ to default pair in $DL$
return $DL$
```

8. Experiments

The objective of our experiments is to explore the capabilities and limitations of the algorithms presented to learn Boolean functions that have a small DNF description in the presence of irrelevant attributes.

We use as a measure of learning performance the percentage of classification errors that the hypothesis makes on an independently drawn data set. We also report the size of the hypothesis.

We analyze the performance of the algorithms on six functions from three Boolean domains: random small DNF, multiplexor and parity. The first domain allows us to easily explore different levels of difficulty by simply changing the size of the formula. The last two domains have been used as benchmark problems for other learning algorithms.

The multiplexor family is a single parametric class of Boolean functions that represent concepts of increasing complexity as the value of the parameter increases. Wilson [1987] proposed it as a test function for his genetic algorithm. Quinlan [1988] showed that accurate decision classifiers can be obtained for three members of the family. We will consider the problem of learning two concepts from this class in the presence of irrelevant attributes.

The parity class of Boolean functions is a hard domain for any learning algorithm that uses a top-down, divide-and-conquer approach when the examples are drawn from the uniform distribution. In a complete sample for a function in this class, half of the examples are positive and half are negative. Furthermore, any attribute (relevant or not) is present in half of the positive examples and in half of the negative examples. Knowing the value of the attribute does not provide any information about the class. In practice, an heuristic search based on the relevance of the attributes is reduced, in this case, to a random guess.

8.1. Experiment design

We executed ten independent runs for each test function. In each execution, the learning and testing tasks were performed on two sets of examples independently drawn from the uniform distribution. The *learning set* was randomly partitioned into two subsets, *training* and *pruning* sets, using the ratios ⅔ and ⅓ [Breiman *et al.*, 1984]. The training set was used to generate a consistent hypothesis. The pruning set was used to reduce the size of the hypothesis and (hopefully) to improve its classification performance on new instances of the target concept.

Let N be the number of attributes and K the number of literals needed to write down the smallest DNF description of the target concept. Let ϵ be the percentage error that can be tolerated during the testing task. The number of learning examples we used for both training and pruning combined is given by the following formula:

$$\frac{K * \log_2(N)}{\epsilon}. \tag{2}$$

This formula represents roughly the number of bits needed to express the target concept times the inverse of the error. This is approximately the number of examples given in [Vapnik, 1982, Blumer *et al.*, 1987] which would suffice for an ideal learning algorithm that only considers hypotheses that could be expressed with at most the number of bits needed for the target concept, and always produces a consistent hypothesis. Qualitatively, the formula indicates that we require more training examples as the complexity of the concept increases or the error decreases. In our experiments we set $\epsilon = 10\%$. We used 2000 examples to test classification performance.

A random observation was generated using *random*, a *UNIX* utility. Each component of the Boolean vector was set to the value of the least significant bit of the random number returned by the procedure.

Table 8. Target functions.

target					*term length*		
concept	*description*	*monotone*	*attributes*	*terms*	*shortest*	*longest*	*average*
dnf1	random DNF	yes	80	9	5	6	5.8
dnf2	random DNF	yes	40	8	4	7	4.5
dnf3	random DNF	no	32	6	4	7	5.5
dnf4	random DNF	no	64	10	3	5	4.1
mx6	6-multiplexor	no	16	4	3	3	3
mx11	11-multiplexor	no	32	8	4	4	4
par4	4-parity	no	16	8	4	4	4
par5	5-parity	no	32	16	5	5	5

8.2. Test target concepts

We present in Table 8 a concise description of the test functions by listing the total number of attributes, the number of terms, the length of the shortest and longest term, and the average term length.

We present in Appendix B a DNF description for the test functions (dnf1, dnf2, dnf3, dnf4) from the first domain. We generated a small random DNF function using four parameters: the total number of terms, the average term length (μ), the standard deviation of a term length (σ), and a flag to indicate if the formula is monotone or not. First, we computed the length of each term according to the Gaussian distribution with parameters μ and σ. Second, for each term we selected its attributes according to the uniform distribution on the attribute set. If the formula was non-monotone, we flipped a fair coin independently for each attribute in a term, if the coin turned up heads we negated the attribute. This flag permits us to easily generate monotone random DNF (for otherwise most random DNF would be non-monotone). It is of interest to contrast the performance of the algorithms over these two types of DNF since, within the Valiant model, Kearns *et al.* [1987] have shown that the problem of learning monotone DNF is as hard as the general problem.

The next two functions were selected from the multiplexor domain. For each positive integer k, there exists a multiplexor function defined on a set of $k + 2^k$ attributes or bits. The function can be defined by thinking of the first k attributes as *address bits* and the last attributes as *data bits* [Wilson, 1987]. The function has the value of the data bit indexed by the address bits. We used 6-multiplexor ($k = 2$) and 11-multiplexor ($k = 4$) as test functions in the presence of 10 and 21 irrelevant attributes, respectively. The first attributes were used as the multiplexor bits.

The last two functions were chosen from the parity domain. For each positive integer k, there exists an even parity function defined on a set of k attributes or bits. The function has value true on an observation if an even number of attributes are present, otherwise it has the value false. Analogously, we can define an odd parity function. We used 4-parity and 5-parity as test functions in the presence of 12 and 27 irrelevant attributes, respectively. The first attributes were used as the parity bits.

Table 9 presents the number of learning examples we used for each test concept. Two thirds of the examples were used for training and one third was used for pruning.

Table 9. Number of learning examples.

target	*examples*	
concept	*training*	*pruning*
dnf1	2195	1097
dnf2	1457	728
dnf3	1100	550
dnf4	1760	880
mx6	480	240
mx11	1067	533
par4	853	427
par5	2667	1333

Table 10. FRINGE results.

target	*avg.% error*		*avg. tree size*		*avg. number*
concept	*first*	*last*	*first*	*last*	*iterations*
dnf1	12.4	0.0	48.7	9.0	13.1
dnf2	10.4	0.5	52.6	7.6	10.3
dnf3	7.4	0.3	45.4	6.1	9.9
dnf4	24.9	0.0	93.9	10.0	9.9
mx6	0.0	0.0	20.2	4.9	5.4
mx11	13.1	0.0	97.0	11.6	7.5
par4	38.3	0.0	124.3	4.9	8.9
par5	36.5	22.1	324.2	120.7	4.4

8.3 FRINGE results

Table 10 presents the results we obtained with FRINGE for each test function. The table reports the average percentage error and the average tree size for the first and last hypothesis. The average was taken over the results of ten executions. The percentage error is the number of classification errors divided by the size of the test sample. The deviation of the actual results from the average is within 7.3% in the first iteration and within 0.4% in the last one, if we do not include par5. The classification performance was highly variable for that concept. The size of the tree is the number of its internal nodes.

The results of the first iteration of FRINGE are the results for our implementation of the decision tree algorithm. By the last iteration FRINGE learned an exact representation for the test functions dnf1, dnf4, mx6, mx11 and par4, it discovered a very accurate description for dnf2 and dnf3, and it failed to learn the par5 concept. In each run of par5 the iterative process ended because the maximum number of variables (350) was reached.

Table 11. GREEDY3 results.

target	*avg. % error*		*avg. num. pairs*	
concept	*no prune*	*prune*	*no prune*	*prune*
dnf1	0.4	0.2	11.6	9.3
dnf2	1.1	0.7	9.8	7.0
dnf3	1.4	0.6	7.5	5.8
dnf4	0.0	0.0	10.4	10.0
mx6	0.4	0.0	6.2	4.3
mx11	1.3	0.5	17.6	9.8
par4	19.3	12.0	69.0	26.7
par5	48.4	45.8	299.1	106.0

The following remarks apply to all the test runs of the algorithm for all target concepts but par5. FRINGE learns an exact or very accurate representation of the target concept by discovering relevant combinations of the primitive literals and by using these features at the decision nodes. The final hypothesis for each test function has the following characteristic structure. The right branch of every decision node points to the positive leaf, and the left branch points to a decision node or to a negative leaf (analogous to the second representation given in Figure 2). For the test functions dnf1 and dnf4 every decision variable is a term in the formula, and all terms are present. For the test functions dnf2 and dnf3, only the most significant terms are represented by a decision variable. For example, for dnf3, FRINGE discovers all the terms but the longest one. The examples that satisfy only this term are very infrequent (about 0.3%). A favorable interpretation would be that FRINGE is treating these examples as a sort of noise in the data. For the multiplexor and parity functions the disjunction of the decision variables in the final tree is a DNF representation of the target concept. Further experiments with FRINGE are reported in [Pagallo, 1989].

8.4. GREEDY3 results

Table 11 summarizes the results we obtained with ten executions of GREEDY3. The percentage error is the number of classification errors divided by the size of the test sample. The percentage error for a single run differed from the average less than 1.2% without pruning and less than 0.6% with pruning, for all concepts but par4 and par5. The accuracy achieved in each run for these concepts was highly variable. The size of the hypothesis is the number of positive pairs in the list. GREEDY3 learned an exact representation for the target concepts dnf4 and mx6; it discovered an accurate hypothesis for the test functions dnf1, dnf2, dnf3 and mx11; but it failed to learn any of the members of the parity family.

The pruning strategy used by GREEDY3 simplifies and improves the hypothesis by removing inaccurate terms. There are two types of spurious terms. In the test functions dnf1, dnf2 and dnf3, the spurious terms were generated at the end of the learning process by statistically insignificant training sets. These terms could be avoided with a larger training set. In the multiplexor function the spurious terms removed by the pruning algorithm were crossover terms. The crossover terms are characteristic of the learning method for functions with strong attribute interaction like multiplexor.

To illustrate the formation of crossover terms consider the 11-multiplexor function:

$$\bar{x}_1\bar{x}_2\bar{x}_3x_4 + \bar{x}_1\bar{x}_2x_3x_5 + \bar{x}_1x_2\bar{x}_3x_6 + \bar{x}_1x_2x_3x_7 +$$
$$x_1\bar{x}_2\bar{x}_3x_8 + x_1\bar{x}_2x_3x_9 + x_1x_2\bar{x}_3x_{10} + x_1x_2x_3x_{11}.$$

For $i = 1,\ldots,8$ let t_i denote the i-th term in the above expression. The initial conditional probability for x_4 is given by:

$$p(+|x_4 = 1) = p(\bar{x}_1\bar{x}_2\bar{x}_3 = 1) + \sum_{i=2}^{8} p(t_i = 1) =$$
$$= \frac{1}{8} + \frac{7}{16} = \frac{9}{16},$$

where the first equality follows from the definition of conditional probability and observing that a positive instance satisfies only one term. Similar calculations show that the initial conditional probability for any data bit is:

$$p(+|x_j = 1) = \frac{9}{16}, \text{ for } j = 4,\ldots,11,$$

while the conditional probability of any address bit is one half. Thus, the algorithm will choose as the first literal a data bit, say x_4. After this selection the conditional probability of a negated address bit $\bar{x}_i$, for $i = 1, 2, 3$ becomes:

$$p(+|\bar{x}_i = 1, x_4 = 1) = \frac{5}{8}.$$

The conditional probability of the address bits x_5 and x_6 also increases:

$$p(+|x_5 = 1, x_4 = 1) = p(+|x_6 = 1, x_4 = 1) = \frac{5}{8}.$$

while the conditional probability of the remaining data bits and the conditional probability of any address bit is smaller than ⅝. Now, the algorithm will tend to choose the negation of an address bit or one of the attributes x_5 or x_6. A small fluctuation in the probability estimation decides which. If, for example, x_5 is selected, the literals $\bar{x}_1$ and $\bar{x}_2$ are added to complete the term $\bar{x}_1 \bullet \bar{x}_2 \bullet x_4 \bullet x_5$, which is a crossover term.

Table 12. GROVE results.

target	*avg. % error*		*avg. num. pairs*	
concept	*no prune*	*prune*	*no prune*	*prune*
dnf1	3.2	4.0	44.7	40.3
dnf2	1.8	1.8	30.2	29.4
dnf3	1.4	1.4	25.5	23.8
dnf4	9.5	7.8	56.7	50.0
mx6	0.9	1.2	10.5	10.4
mx11	3.4	3.9	37.2	33.4
par4	10.4	13.0	45.6	45.6
par5	41.6	41.3	223.4	223.0

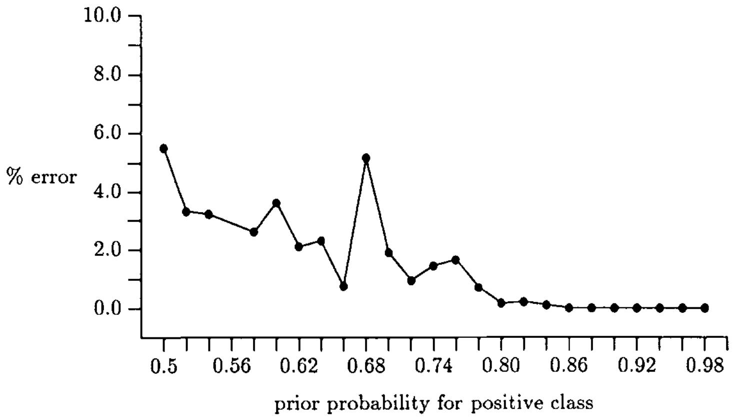

Figure 4. GROVE performance on dnf4.

8.5. GROVE results

Table 12 summarizes the results we obtained in ten executions of GROVE using uniform priors. The average percentage error differs from the result of a single run less than 3.7% without pruning and less than 3.6% with pruning, for all concepts but par4 and par5. GROVE learned a description within the established accuracy for the test functions in the small DNF and multiplexor domains, while it failed to obtain an accurate representation of the parity concepts.

The accuracy of the results can be improved in all the cases by selecting an *appropriate* value for the a-priori class probability. GROVE did not perform well using the data priors. Figure 4 illustrates the dependency of the accuracy of the hypothesis on the value of the class priors for test function dnf4.

For the prior range 0.86-0.98 for the positive class, the algorithm discovers an exact representation of the concept. We observed a similar dependency between priors and performance for the remaining concepts, but it does not have a characteristic pattern that we could detect.

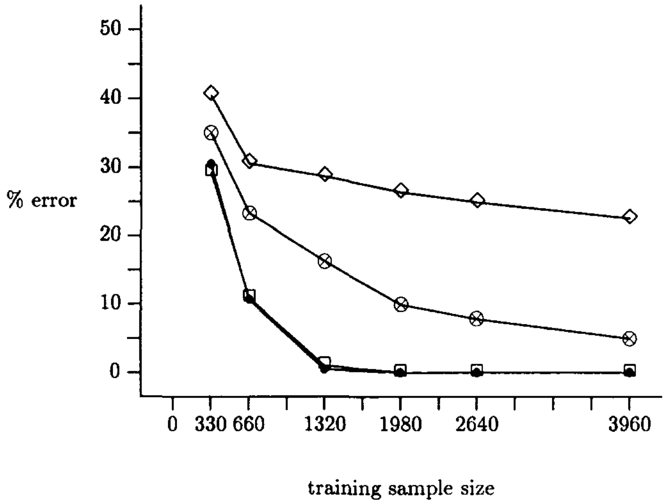

Figure 5. Learning Curves for dnf4. Learning curves are indicated as: ◇: Decision Tree, □: FRINGE, ●: GREEDY3, ⊗: GROVE.

For concept dnf4 and the prior probability of the positive class in the range 0.86-0.98 the hypothesis is a DNF like decision list (all pairs except the default one are positive pairs). This observation supports the suggestion made by Quinlan (personal communication) to learn a decision list representation where all the pairs but the default one belong to the less numerous class. Then, the default pair would have the class label of the more numerous class. However, more experiments are needed to validate this strategy to bias the term formation procedure.

8.6. Empirical comparisons among methods

The poor performance of the decision tree algorithm for the random DNF and mxll concepts is due to a representational limitation of decision trees as a concept description language for these types of concepts. While learning a decision tree the sample subdivision process has to fragment the examples into a large number of subsets to find a tree description of the hypothesis (see Section 3). Since the tree is so large, the examples are exhausted before all branches in the tree can be explored. The better performance of the new methods arises from a more adequate concept description language for concepts with a small DNF description.

To see if the replication problem can be overcome by varying the size of the training set, we present in Figure 5 the learning curves for dnf4. A learning curve shows how the classification accuracy varies as a function of the size of the training sample. Here, each point in the curve is the average classification error over ten runs. We use the same design criteria as those described in Section 8.1. A sample of size 2640 is the size of the learning set predicted by Formula (2) for $\epsilon = 10\%$.

FRINGE's hypothesis for any of the random DNF concepts was a DNF-like decision list, the same type of hypothesis that GREEDY3 generates. And for this domain, their classification performance is quite comparable.

Finally, the better performance of GREEDY3 over GROVE for all test domains except parity is due to the stronger bias built into GREEDY3. In GREEDY3 only the positive examples are explained by the hypothesis, while the negative examples are left as the default. By contrast, GROVE chooses which examples to cover and what class to leave as the default one. However, this design gives GROVE a greater flexibility in other domains.

9. Conclusions and future directions

In this paper, we view the problem of learning from examples as the task of discovering an appropriate vocabulary for the problem at hand. We present three new algorithms based on this approach. The first algorithm uses a decision tree representation and it determines an appropriate vocabulary through an iterative process. The last two algorithms use a decision list representation. These algorithms form their hypothesis from the primitive literals using a top-down greedy heuristic.

We also show empirically that they compare very favorably to an implementation of a decision tree algorithm. Since the difficulties of the standard decision tree method arise from a representational limitation, we expect that any reasonable implementation of the decision tree algorithm will give similar results.

Many more experiments could be done. We would like to test the methods in other domains that have been used as a benchmark for learning algorithms, and test their sensitivity to noise in the data. Some FRINGE tests on noisy domains are given in [Pagallo, 1989]. Other strategies to define Boolean features from a decision tree could also be explored. The heuristic used by FRINGE is compared to several different feature formation heuristics in [Matheus, 1989], with favorable results. He also reports learning time curves for the methods. We also would like to test and explore alternatives to bias the term generation procedure for GROVE. Finally, GROVE can be easily extended to problems with non-Boolean attributes and multiple concepts, and experiments need to be done in this direction.

We also have some theoretical results for GREEDY3. In [Pagallo and Haussler, 1989] we show that a variant of GREEDY3 is a polynomial learning algorithm for μDNF formulae (DNF where each attribute appears at most only once) under the uniform distribution in the sense of [Kearns *et al.*, 1987]. In the formal version of GREEDY3 we need to resample after each attribute is selected to ensure unbiased estimates of the conditional probabilities.

Finally, a number of problems remain open. Do any of the algorithms we presented (FRINGE, GREEDY3, GROVE) learn the class of all monotone DNF concepts under the uniform distribution in the sense of [Kearns *et al.*, 1987]? For any of the algorithms REDWOOD (or ID3), FRINGE, GREEDY3, GROVE, can one in any reasonable way characterize the class of target concepts and distributions that the algorithm works well on?

Acknowledgments

We would like to thank Ross Quinlan for the helpful discussion on the methods, and Wray Buntine, Tom Dietterich, Jeff Schlimmer and Paul Utgoff for their valuable comments on an earlier version of this paper. We would like to thank Doug Fisher for bringing Clark and Niblett's paper to our attention and for his useful comments on an earlier version of this paper. We would also like to thank Paul Rosenbloom and the reviewers for many useful comments. The support of the ONR grant N00014-86-K-0454 is gratefully acknowledged.

Appendix A

Let V be a set of n Boolean attributes, and let C be the set of all conjunctions formed with attributes drawn from the set V. We write a DNF formula f over the set V as

$$f = C_1 + C_2 + \ldots + C_m$$

where each $C_i \in C$, and m is a positive integer. For each i, $1 \leq i \leq m$, we denote by V_i the set of attributes in term C_i and by k_i the cardinality of this set.

We say that f is a μDNF formula if for any $1 \leq i, j \leq m$, $i \neq j$, $V(C_i) \cap V(C_j)$ is the empty set. Without loss of generality we assume that $\cup_{i=1}^{m} C_i = V$.

We define an *assignment* $x{:}V \rightarrow \{0, 1\}$ as an assignment of Boolean values to each attribute in V. Given an assignment x the resulting value of f is denoted by $f(x)$.

Let T be a decision tree over the set of variables V. An assignment defines a path from the root node of T to a leaf node. We denote by $L(x)$ the leaf node and by $T(x)$ the class label of the leaf. We say that T is equivalent to f if for every assignment x, $T(x) = f(x)$.

THEOREM. Let f be a μDNF formula over the set of variables V. Any decision tree equivalent to f has at least $k_1 \times k_2 \times \ldots \times k_m$ leaves.

Consider the set Z of all assignments of V where one variable of each term is assigned to 0 while all other variables are assigned to 1. Clearly, $|Z| = k_1 \times k_2 \times \ldots \times k_m$. Also, observe that two assignments in Z differ, at least, in the value assigned to two variables that belong to the same term. Furthermore, for all $z \in Z$, $f(z) = 0$.

Let T be a decision tree over V equivalent to f. Assume that there are less than $k_1 \times k_2 \times \ldots \times k_m$ negative leaves in T. Then, there exist two distinct assignments z_1, $z_2 \in Z$ such that $L(z_1) = L(z_2)$. In other words, the assignments z_1 and z_2 define the same path from the root node to a negative leaf. Let V_i be one of the variables sets where z_1 and z_2 differ (by construction there exists at least one of such set), and let $y \in V_i$ be the variable that z_1 assigns to 0. Then, z_2 must assign y to 1. Therefore, the variable y cannot be tested in the path from the root to $L(z_1)$.

Let r be the assignment that assigns the value 1 to y and coincides with z_1 elsewhere. Hence, $L(r) = L(z_1)$, and $T(r) = T(z_1)$. But this contradicts the fact that $f(r) = 1$. Therefore, the number of negative leaves in T must be greater or equal to $k_1 \times k_2 \times \ldots \times k_m$.

Appendix B: DNF description of test target concepts

$$
\begin{array}{lllll}
\textbf{dnf1} & x_5x_{28}x_{38}x_{72}x_{74}x_{76} & + \; x_2x_{16}x_{40}x_{52}x_{74} & + \; x_{10}x_{21}x_{23}x_{28}x_{30}x_{63} & + \\
 & x_{40}x_{56}x_{58}x_{60}x_{63}x_{72} & + \; x_6x_{24}x_{36}x_{37}x_{39}x_{48} & + \; x_3x_{17}x_{45}x_{55}x_{72}x_{75} & + \\
 & x_{11}x_{48}x_{50}x_{64}x_{69}x_{74} & + \; x_2x_{15}x_{27}x_{36}x_{50}x_{53} & + \; x_6x_{12}x_{22}x_{45}x_{60} & \\
\textbf{dnf2} & x_1x_3x_{14}x_{19}x_{26}x_{35}x_{36} & + \; x_8x_{15}x_{31}x_{37} & + \; x_5x_{10}x_{14}x_{27}x_{29} & + \\
 & x_{18}x_{20}x_{30}x_{36} & + \; x_2x_3x_9x_{19}x_{24} & + \; x_{24}x_{25}x_{27}x_{36}x_{37} & + \\
 & x_6x_7x_{14}x_{25}x_{26}x_{31}x_{34} & + \; x_1x_6x_{22}x_{30} & &
\end{array}
$$

$$
\begin{array}{lllll}
\textbf{dnf3} & x_1x_2x_6x_8x_{25}x_{28}\bar{x}_{29} & + \; x_2x_9x_{14}\bar{x}_{16}\bar{x}_{22}x_{25} & + \; x_1\bar{x}_4\bar{x}_{19}\bar{x}_{22}x_{27}x_{28} & + \\
 & \bar{x}_2x_{10}x_{14}\bar{x}_{21}\bar{x}_{24} & + \; x_{11}x_{17}x_{19}x_{21}\bar{x}_{25} & + \; \bar{x}_1\bar{x}_4x_{13}\bar{x}_{25} &
\end{array}
$$

$$
\begin{array}{lllll}
\textbf{dnf4} & x_1x_4x_{13}x_{57}\bar{x}_{59} & + \; x_{18}\bar{x}_{22}\bar{x}_{24} & + \; x_{30}\bar{x}_{46}x_{48}\bar{x}_{58} & + \\
 & \bar{x}_9x_{12}\bar{x}_{38}x_{55} & + \; \bar{x}_5x_{29}\bar{x}_{48} & + \; x_{23}x_{33}x_{40}x_{52} & + \\
 & x_4\bar{x}_{26}\bar{x}_{38}\bar{x}_{52} & + \; x_6x_{11}x_{36}\bar{x}_{55} & + \; \bar{x}_6\bar{x}_9\bar{x}_{10}x_{39}\bar{x}_{46} & + \\
 & x_3x_4x_{21}\bar{x}_{37}\bar{x}_{55} & & &
\end{array}
$$

References

Beach, L.R. (1964). Cue probabilism and inference behavior. *Psychological Monographs,* Whole. 582.

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

Breiman, L., Friedman, J.H., Olsen, R.A., & Stone, C.J. (1984). *Classification and Regression Trees.* Wadsworth Statistic/Probability Series.

Carbonell, J., Michalski, R., & Mitchell, T. (1983). An overview of machine learning. In *Machine Learning: An Artificial Intelligence Approach,* 3–24 Morgan Kaufmann.

Clark, P. & Niblett, T. (1989). The CN2 induction algorithm. *Machine Learning, 3*:251–283.

Flann, N. & Dietterich, T. (1986). Selecting appropriate representation for learning from examples. *Proceedings of AAAI-86,* (pp. 460–466). Morgan Kaufmann.

Haussler, D. (1988). Quantifying inductive bias: AI learning algorithms and Valiant's learning framework. *Artificial Intelligence,* 36: 177–221.

Kearns, M., Li, M. & Valiant, L. (1987). Recent results on Boolean concept learning. *Proceedings of 4th International Workshop on Machine Learning,* (pp. 337–352). Morgan Kaufmann.

Matheus, C. (1989). *Feature Construction: An Analytic Framework and An Application to Decision Trees.* Ph.D. thesis, University of Illinois, December 1989. In preparation.

Muggleton, S. (1987). Duce, an oracle-based approach to constructive induction. *Proceedings of IJCAI-87,* (pp. 287-292). Morgan Kaufmann.

Pagallo, G. and Haussler, D. (1989). A greedy method for learning μDNF functions under the uniform distribution. (Technical Report UCSC-CRL-89-12, Santa Cruz: Dept. of Computer and Information Sciences, University of California at Santa Cruz.

Pagallo, G. (1989). Learning DNF by decision trees. *Proceedings of IJCAI-89,* (pp. 639–644). Morgan Kaufmann.

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

Quinlan, J.R. (1987a). Generating production rules from decision trees. *Proceedings of IJCAI-87,* 1; (pp. 304–307). Morgan Kaufmann.

Quinlan, J.R. (1987b). Simplifying decision trees. *International Journal of Man-machine Studies,* 27:221–234.

Quinlan, J.R. (1988). An empirical comparison of genetic and decision tree classifiers. *Proceedings of the 5th International Conference on Machine Learning* (pp. 135–141). Morgan Kaufmann.

Rivest, R. (1987). Learning decision lists. *Machine Learning,* 2:229–246.

Schlimmer, J.C. (1986). Concept acquisition through representational adjustment. *Machine Learning*, 1:81–106.

Utgoff, P. and Mitchell, T.M. (1982). Acquisition of appropriate bias for inductive concept learning. *Proceedings of AAAI-82*, (pp. 414–417). Morgan Kaufmann.

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

Valiant, L.G. (1985). Learning disjunctions of conjunctions. *Proceedings of IJCAI-85*, (pp. 560–566). Morgan Kaufmann.

Vapnik, V.N. (1987). *Estimation of Dependencies Based on Empirical Data*. Springer Verlag.

Wilson, S.W. (1987). Classifier systems and the animat problem. *Machine Learning*, 2:199–228, 1987.