New Algorithms for Learning and Pruning Oblique Decision Trees

Shesha Shah and P. S. Sastry, *Senior Member, IEEE*

***Abstract*—In this paper, we present methods for learning and pruning oblique decision trees. We propose a new function for evaluating different split rules at each node while growing the decision tree. Unlike the other evaluation functions currently used in literature (which are all based on some notion of *purity* of a node), this new evaluation function is based on the concept of *degree of linear separability*. We adopt a correlation-based optimization technique called the Alopex algorithm for finding the split rule that optimizes our evaluation function at each node. The algorithm we present here is applicable only for 2-class problems. Through empirical studies, we demonstrate that our algorithm learns good compact-decision trees.**

We suggest a representation scheme for oblique decision trees that makes explicit the fact that an oblique decision tree represents each class as a union of convex sets bounded by hyperplanes in the feature space. Using this representation, we present a new pruning technique. Unlike other pruning techniques, which generally replace heuristically selected subtrees of the original tree by leaves, our method can radically restructure the decision tree. Through empirical investigation, we demonstrate the effectiveness of our method.

***Index Terms*—Learning, linear separability, optimization, tree induction, tree pruning.**

I. Introduction

DECISION-tree classifiers have been popular in pattern recognition, concept learning, and other AI branches [1]. They enable a divide-and-conquer strategy to be applied to classification problems, and they enable context-sensitive feature-subset selection to tackle high-dimensionality problems. Decision-tree learning started receiving increased attention after Quinlan's work on ID3 [2] and the CART methodology of Breiman *et al.* [3].

In this paper, we present a new algorithm called the Alopex Perceptron Decision Tree (APDT) algorithm for learning a decision tree given a set of (preclassified) training patterns. This is a top-down, tree-growing algorithm, and its main novel feature is a new approach to evaluating goodness of various possible split rules at every node. We also present a new technique for pruning learned decision trees that is often needed in order to mitigate problems of overfitting. All algorithms presented in this paper are applicable only for 2-class pattern-recognition problems. In the final section, we briefly indicate how these techniques may be extended to handle multiclass problems.

Manuscript received June 12, 1997; revised April 8, 1999.

The authors are with the Department of Electrical Engineering, Indian Institute of Science, Bangalore, 560 012 India.

Publisher Item Identifier S 1094-6977(99)08193-6.

A decision tree is a classification rule represented as a binary tree in which each nonleaf node is associated with a decision rule (or a "split rule" or a "test") of the form $f_i(X) > \theta_i$. This decides whether a given feature vector X will go to the left or right subtree. Each leaf node is assigned a class label, and all the patterns landing in that node are to be classified in that class.

Given a sample of classified patterns, most learning algorithms construct a decision tree by recursively determining split rules at each node while growing the tree in top-down fashion [1]–[4]. Let $\mathcal{F}$ denote the family of possible split rules, and let $F(\cdot, S^t)$: $\mathcal{F} \rightarrow \Re$ be an evaluation function that measures the goodness of any $f \in \mathcal{F}$ given a subset of classified patterns S^t. Different algorithms make different choices of $\mathcal{F}$ and $F(\cdot, \cdot)$.

One popular choice for $\mathcal{F}$ is rules of the form $x_i > \theta$ where x_i is the ith attribute (or feature) value and θ is a constant. Decision trees obtained by using this class of split rules are known as *axis-parallel decision trees* because here, each class region in the feature space is represented by a union of hyper-rectangles with sides parallel to feature axes.

A more general class contains rules of the form $\Sigma_{i=1}^{n} \; a_i x_i > \theta$, which gives rise to the so-called oblique decision trees [3], [4]. Here, each split rule is characterized by the parameters $a_1, a_2, \cdots, a_n$, and θ. Oblique decision trees effect polyhedral partitioning of the feature space. In general, when classes are separated by a piecewise linear surface, oblique splits result in more compact decision trees. This is because, using axis-parallel splits, we need to approximate an arbitrary hyperplane with a staircase-like structure.

Almost all of the decision tree algorithms in literature use an evaluation function $F(\cdot, S^t)$, which is a measure of the degree of impurity of children nodes resulting from a split.[1] That is, a rule that splits S^t into S^{t_l} and S^{t_r}, such that each of them contains an approximately equal number of patterns from all classes will be judged to be greatly inferior to a split rule that results in S^{t_l} and S^{t_r}, each predominantly containing patterns of one class. While such impurity measures may be good for axis-parallel trees, they are not very good for oblique trees, especially in 2-class problems. This is because we can use a hyperplane as a split rule, and a single split can complete the classification of any subset of linearly separable patterns. This situation is illustrated in Fig. 1. For a 2-class problem in $\Re^2$ with the class regions as shown, consider the two hyperplane splits, as in Fig. 1(a) and 1(b). The split of

[1] A node is called *pure* if all the training patterns landing at the node are of the same class.

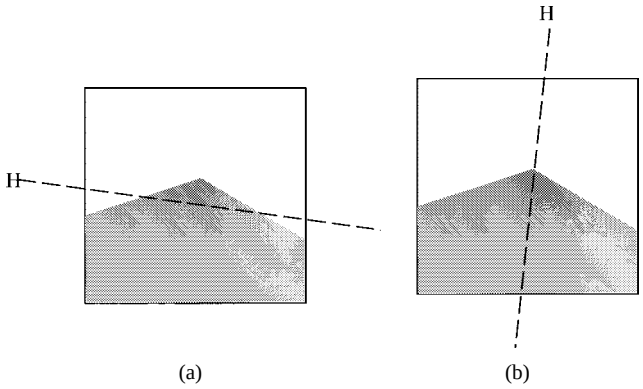


Fig. 1. Split rule based on (a) purity measure versus (b) separability measure.

Fig. 1(a) results in S^{t_l} and S^{t_r}, each having a much higher degree of purity compared to those in Fig. 1(b). However, two more hyperplanes for each of the children nodes are needed in Fig. 1(a) to complete the classification, while only one hyperplane is needed in each child for Fig. 1(b), as both S^{t_l} and S^{t_r} are linearly separable here. Because oblique decision trees can realize arbitrary piecewise linear separating surfaces, it seems better to base the evaluation function $F(\cdot, S^t)$ on the degree of separability of S^{t_l} and S^{t_r} rather than on the degree of purity of S^{t_l} and S^{t_r}. In this paper, we propose a new function that evaluates a split rule based on the "degree of linear separability" of the children nodes resulting from the split rule. We use ideas from [5] to characterize the degree of linear separability.

Here, in our tree-growing algorithm, we employ a correlation-based optimization technique called Alopex [6]. Empirically, the Alopex algorithm is found to be effective in many neural networks and other optimization problems [6]. We show that using our algorithm, we learn smaller decision trees with better classification accuracy than is possible with other standard methods.

The rest of the paper is organized as follows. In Section II, we present the complete details of the APDT algorithm. Through simulation results presented in Section III, we show that the APDT algorithm learns compact and accurate decision trees. In Section IV, we present a new pruning technique, along with some simulation results to show its effectiveness. Section V presents discussion and conclusions.

II. Algorithm for Learning Oblique Decision Trees

Consider a top-down, tree-growing algorithm for learning oblique decision trees. Here, the main computation (at each node) is as follows. Given a training sample of classified patterns, $S = \{(X_1, C_1), (X_2, C_2), \cdots, (X_m, C_m)\}$ where $X_i = (x_{i1}, x_{i2}, \cdots, x_{in}) \in \Re^n$ is the feature vector, and $C_i \in \{0, 1\}$ is the class label of the ith sample, we want to determine the best hyperplane to subdivide the sample at that node. If the sample is linearly separable, then of course the best hyperplane is the separating hyperplane. However, in general, the pattern-set given to the algorithm at most nodes will be nonseparable. As discussed in the previous section, we want to base our evaluation of hyperplanes on the degree of separability of resulting children nodes.

There are some interesting geometric characterizations of a set of feature vectors (in $\Re^n$) that are not linearly separable (see [5] for a detailed account). For the purposes of the algorithm presented here, we note the following. Suppose we run the Perceptron algorithm over a training set S of pattern vectors. In the Perceptron algorithm, at each iteraton we classify the next pattern (feature vector) using current hyperplane, and if the classification is wrong, then we update the hyperplane using that pattern. When the set S is not linearly separable, the algorithm goes into an infinite loop. However, not all the patterns in the training set S need participate in updating the hyperplane infinitely often. In general, there is a subset of vectors from S called "nonseparable" patterns of S that update the hyperplane infinitely often, while the rest of the patterns of S update only a finite number of times [5, Theorem 6]. For our algorithm, we take the number of nonseparable patterns of S as an indication of the degree of linear separability of S. We can estimate the number of nonseparable patterns of a set of training patterns S as follows. We run the Perceptron algorithm on S for a fixed number of iterations, keeping a count of the number of times each pattern participated in updating the hyperplane. At the end of the algorithm, if for instance, M is the maximum count, we then take all the patterns that participated in updation at least ρM times as nonseparable patterns, where $\rho < 1$ is a parameter (in our algorithm, we take $\rho = 0.6$). What we need for our algorithm is the number of such nonseparable patterns of S.

A. New Function to Evaluate Split Rules

Let, S^t be the sample set at node t with n_t patterns. The split at node t is parametrized by weight vector $\boldsymbol{W} = (w_0, w_1, \cdots, w_n) \in \Re^{(n+1)}$. Let $\boldsymbol{X} = (x_1, x_2, \cdots, x_n) \in \Re^n$ be a given feature vector. Let $S^{t_l} = \{\boldsymbol{X} \in S^t : \sum_{i=1}^{n} w_i x_i + w_0 > 0\}$ and $S^{t_r} = S^t - S^{t_l}$. When $\boldsymbol{W}$ is the split rule at t, S^{t_l} and S^{t_r} are the pattern-sets that go into the left and right children t_l and t_r of node t. Let $n_{t_l} = |S^{t_l}|$ and $n_{t_r} = |S^{t_r}|$. Let nsp_{t_l} and nsp_{t_r} be the number of nonseparable patterns in the sets S^{t_l} and S^{t_r}, respectively.

Our aim is to have an evaluation function for split rules that minimizes nonseparable patterns in children nodes. When a child is linearly separable, we prefer the split that puts more patterns in that child, preserving linear separability. To do this, define an evaluation function $F(\cdot, \cdot)$ as

$$F(W, S^t) = \underbrace{\frac{R}{M}\left\{I_1 \frac{n_t - n_{t_l}}{n_t} + I_2 \frac{n_t - n_{t_r}}{n_t} - I_1 I_2\right\}}_{A} + \underbrace{(1 - I_1)(1 - I_2)E}_{B} \quad (1)$$

where

W	parameter vector of the hyperplane;
S^t	the set of patterns at node t and $n_t = \|S^t\|$ (the number of patterns in S^t);
n_{t_l}	$\|S^{t_l}\|$, where S^{t_l} is the set of a pattern at left child of t obtained by splitting S^t using W;
n_{t_r}	$\|S^{t_r}\|$, where S^{t_r} is the set of a pattern at right child of t obtained by splitting S^t using W;

nsp_{t_l}, number of nonseparable patterns in S^{t_l} and S^{t_r};
nsp_{t_r} which is calculated using the Perceptron altorithm for a fixed number of iterations as explained earlier;
E $\big(((nsp_{t_l}+nsp_{t_r})/n_t)((1/2)-(n_{t_l}/n_t)(n_{tr}/n_t))\big)$;
R minimum value of $E=(1/2n_t)$.

If $nsp_{t_l}=0$, then $I_1=1$, else $I_1=0$.
If $nsp_{t_r}=0$, then $I_2=1$, else $I_2=0$.
$M \geq 1$ is a constant (we have used $M=4$ in our simulations).

In (1), when any of the children is linearly separable, only term A will contribute. But when both children are not linearly separable, $F(\cdot,\cdot)$ is given only by term B. From (1), we observe the following.

- When either of the children is separable (for instance if t_l is linearly separable, i.e., $nsp_{t_l}=0$), then $F(W,S^t)=(n_{t_r}/n_t)(R/M)$. That is, by minimizing $F(\cdot,\cdot)$, we decrease the patterns falling in right child t_r.
- When both of the children are linearly separable (i.e., $nsp_{t_l}=0$ and $nsp_{t_r}=0$), $F(W,S^t)=0$, which is a global minimum.
- In the case in which both children are not linearly separable, $F(W,S^t)=((nsp_{t_l}+nsp_{t_r})/n_t)\big((1/2)-(n_{t_l}/n_t)(n_{t_r}/n_t)\big)$. By minimizing $F(\cdot,S^t)$, we minimize nonseparable patterns in both children. But here we have an additional product term $\big((1/2)-(n_{t_l}/n_t)(n_{t_r}/n_t)\big)$, which ensures that a split that puts all patterns at t in one child node will not result. Also note that R in term A in (1) will ensure that the value of term A is always smaller than that of term B.

We use the above $F(\cdot,\cdot)$ function as an evaluation function in our APDT algorithm. In this algorithm, in order to find the split rule at each node that minimizes $F(\cdot,\cdot)$, we use the Alopex optimization algorithm, which is described in the next section.

B. Alopex Algorithm

To learn the best hyperplane, we need to minimize $F(\cdot,S^t)$ at each node t. From the definition of $F(\cdot,\cdot)$ it is clear that given a sample S^t at node t and a weight vector W of the hyperplane, we can calculate $F(W,S^t)$. However, it is not possible to get gradient information directly. We need to estimate the gradient or use some algorithm that does not require gradient information. Here, we use the Alopex algorithm [6], which does not require gradient information. Below, we describe the Alopex algorithm for a general optimization problem, following [6].

Let E:$\Re^n \rightarrow \Re$ be the objective function. We are interested in finding a $W=(w_1,w_2,\cdots,w_n)\in\Re^n$ at which E attains a minimum. Let $W(k)$ be the estimate of minimum at kth iteration. Then, $W(k)$ is updated as

$$w_i(k+1)=w_i(k)+\delta x_i(k), \qquad 1\leq i\leq n \tag{2}$$

where δ is the step size, and $\boldsymbol{X}(k)=[x_1(k),x_2(k),\cdots,x_n(k)]$ is a random vector with $x_i(k)\in\{+1,-1\},\forall i,k,$ having the following distribution

$$x_i(k)=\begin{cases} x_i(k-1), & \text{with probability } p(k) \\ -x_i(k-1), & \text{with probability } 1-p(k) \end{cases}. \tag{3}$$

Here

$$p(k)=\frac{1}{1+\exp\left(\dfrac{\Delta E(k)}{T(k)}\right)} \tag{4}$$

where $T(k)$ is a positive "temperature" parameter, and $\Delta E(k)$ is the changes in the objective function E over the previous two iterations

$$\Delta E(k)=E(W(k))-E(W(k-1)).$$

The temperature T is updated at every N iterations (where N is a parameter of the algorithm) using the following "annealing schedule"

$$T(k)=\begin{cases} \dfrac{\delta}{N}\displaystyle\sum_{k'=k-N}^{k-1}|\Delta E(k')|, & \text{if } k \text{ is multiple of } N \\ =T(k-1), & \text{otherwise.} \end{cases} \tag{5}$$

The algorithm is initialized by randomly choosing $W(0)$ and obtaining $W(1)$, done by making $x_i(0)$ take values $+1$ and -1 with equal probability for each i. From (3) and (4) it is clear that at each iteration, the algorithm makes a random choice of direction $\boldsymbol{X}(k)=[x_1(k),x_2(k),\cdots,x_n(k)]$, which is computed in a distributed fashion and the algorithm makes a step of constant size along that direction. Note that each $x_i(k)$ is calculated independently based on $p(k)$. If over the previous two iterations E has decreased, then each w_i is changed in the same direction as in the previous iteration with a probability greater than 0.5. How fast these probabilities go to zero or one with increasing magnitude of ΔE depends on the temperature. From the annealing schedule given in (5), it is clear that the algorithm has some self-scaling property. This is because, in determining $p(k)$, essentially we are comparing current ΔE with its recent averaged value. The free parameters of the algorithm are δ, the step size, and N (the number of iterations over which ΔE is averaged to get temperature in an annealing schedule). The initial temperature $T(0)$ can be arbitrary. The value of N should be chosen so that sufficient averaging of ΔE takes place. The algorithm is not very sensitive to N as long at it is not too small. The parameter δ is step size in updating w_i. The algorithm is not very sensitive to δ as long as it is small. Also, it is observed empirically that reducing the value of δ toward the end helps. For a multidimensional input pattern with each attribute having values in different ranges, normalizing pattern vectors improve learning accuracy and learning rate. This problem also can be handled by choosing a different value of δ for each component w_i in Alopex.

C. APDT Algorithm for Decision-Tree Induction

Our APDT algorithm consists of finding the split rule W at each node t to minimize $F(W,S_t)$ [given in (1)] using the Alopex algorithm (see Section II-B). The complete APDT algorithm is given in Table I.

The algorithm grows the decision tree by recursively calling *growTree*() procedure. At each node t, *growTree* is called with a subset of training samples S_t, which fall into that node. We use the Alopex algorithm to find the best W at that node. In

TABLE I
PART I OF COMPLETE APDT ALGORITHM

APDT Algorithm

Input : Sample $S = \{(X_i, c_i)\}_{i=1,\cdots,m}$ where $X_i \in \Re^n$ and $c_i \in \{0, 1\}$

Output : pointer to an oblique decision tree

```
begin
   Root = growTree( S );
   return(Root);
end
```

Procedure $growTree(S^t)$

Input : Sample at node t, S^t

Output : pointer to a sub-tree

```
begin
```

1. Initialize hyperplane parameters, $W(0)$, to a plane which is perpendicular to a vector joining any two patterns, chosen randomly from S^t, of different classes. Let $W^* = W(0)$.
2. Calculate $F^* = F(W^*, S^t)$ using equation (1).
3. Initialize the Alopex algorithm (see Section 2.2).
4. For $k = 2, \cdots, AloITE$
   begin
   - get hyperplane parameters, $W(k)$, by updating $W(k-1)$ using Alopex algorithm
   - Calculate $F_k = F(W(k), S^t)$, which returns values of nsp_{t_l} and nsp_{t_r}. If S^{t_l} (resp. S^{t_r}) is linearly separable, then separating hyperplane W^{t_l} (resp. W^{t_r}) is also returned.
   - If $F_k < F^*$ then assign $F^* \leftarrow F_k$ and $W^* \leftarrow W(k)$
   - If $F^* = 0$ then exit to step 5.

   end
5. Make node t, and assign $W_t \leftarrow W^*$ as the split rule at node t.

each iteration k of Alopex, we obtain S_{t_l} and S_{t_r} from S_t, using the current split W_k. We then run the Perceptron algorithm for a fixed number of iterations on S_{t_l} and S_{t_r} to obtain the number of nonseparable patterns nsp_{t_l} and nsp_{t_r} in them, respectively. Using these values, we calculate $F(W_k, S_t)$, which is used in the Alopex algorithm to update W_k. The Alopex algorithm runs for a fixed number of iterations. The same call to the Perceptron algorithm that returns the number of nonseparable patterns also returns the separating hyperplane W_{t_l} (respectively, W_{t_r}) if S_{t_l} (respectively, S_{t_r}) is linearly separable.

In APDT algorithm *growTree*($\cdot$), a routine is called recursively only when the sample at that node is not linearly separable. If any child node was linearly separable, it would have been detected by the $F(\cdot, \cdot)$ routine in step 4, and declared as a leaf node in step 7. Thus, in the main routine of the APDT code, we assume that sample is not linearly separable (which can be easily detected before calling *growTree*($\cdot$). Also note that in step 4, when we have $F^* = 0$ we have found a hyperplane that gives linearly separable children. That results in adding one or three nodes for each child as given in step 7, because the separating hyperplane for a linearly separable child node (when it is not pure node) is already found while running the Perceptron algorithm to count nonseparable patterns for that child. This is a kind of one-step look-ahead while recursively growing the tree. The APDT algorithm is a stochastic algorithm because of Alopex. On every run it gives a different tree, which is consistent with the training set. For this reason, we can use the "multiple trees" approach for classification [3].

III. SIMULATION RESULTS

We present some of the results of our empirical studies of the APDT algorithm both on synthetic problems and on some Machine Learning databases from UCI ML Database repository [7].

All our synthetic problems are two-dimensional (2-D) and the feature space is the unit square in $\Re^2$. For training, a fixed number of patterns are generated randomly using uniform distribution. To evaluate performance of learned decision tree we generated another test sample using uniform distribution.

In the APDT algorithm, we run the Perceptron algorithm for max $\{300, |S^t|\}$ iterations and the Alopex algorithm for 500 iterations. For the Alopex algorithm, we have taken $\delta = 0.05$, $N = 10$, and $T(0) = 1000$. The results presented from the APDT algorithm are without any pruning. We compared the

TABLE I
PART II OF COMPLETE APDT ALGORITHM (*Continued.*)

```
6. Let S^{t_l} = {X_i ∈ S^t | W_t^T · X_i > 0} and S^{t_r} = {X_i ∈ S^t | W_t^T · X_i ≤ 0}.
   If all patterns of S^{t_l} have same class
            then Pure_{t_l} = TRUE
            else Pure_{t_l} = FALSE.
   If all patterns of S^{t_r} have same class
            then Pure_{t_r} = TRUE
            else Pure_{t_r} = FALSE.

7. • If (nsp_{t_l} ≠ 0) then
         t_l = growTree(S^{t_l})
      else
      begin
         If (Pure_{t_l} = FALSE ) then
            assign W_{t_l} as split rule at t_l, and attach appropriate
            class labels to left and right children of t_l.
         else
            assign appropriate class label to t_l.
      end
    • If (nsp_{t_r} ≠ 0) then
         t_r = growTree(S^{t_r})
      else
      begin
         If (Pure_{t_r} = FALSE ) then
            assign W_{t_r} as split rule at t_r, and attach appropriate
            class labels to left and right children of t_r.
         else
            assign appropriate class label to t_r.
      end

8. return(t)
   end
```

APDT algorithm with four other algorithms: ID3, CART-AP and CART-LC, and OC1.

ID3, due to Quinlan, is a popular method for discovering classification rules from a sample of patterns over nominal attribute space [2], [8]. This is extended to handle linear attributes by Fayyad and Irani [9], and we use this extension. ID3 uses an impurity-based evaluation function motivated by some information theoretic considerations, and it does an exhaustive search at each node to come out with the best axis-parallel split rule. We used Utgoff's program[2] [10] to simulate ID3, which also includes pruning to avoid overfitting.

CART is another popular approach for inducing decision trees. Here, we termed the CART algorithm for learning axis-parallel decision trees as CART-AP, and the algorithm for learning oblique decision trees as CART-LC. Both algorithms use Gini impurity measure [3] for evaluating split rules at each node. CART-AP does an exhaustive search to find the best axis-parallel split at each node, while CART-LC uses a perturbation-based method [3]. In CART simulation, we employed 0-SE, CC pruning [3] on the learned decision tree.

[2]This program for ID3 does not give training and test accuracy separately. So Table II has the same values in corresponding columns.

The OC1 algorithm [4] is similar to CART-LC, and it also learns oblique decision trees. We have used Murthy's [4] programs for simulating OC1, and this program includes 0-SE, CC pruning. In our simulation, none of the methods employ any explicit feature-subset selection.

Performance of various algorithms is compared by evaluating the learned decision trees in terms of accuracy and size. Accuracy is measured by the classification error made on the test sample. The size of a tree is characterized by number of nodes and the depth of tree. A good tree is one with high accuracy, few nodes, and less depth. We give average (over five runs) and best results for all the problems. Note that in CART-LC, OC1, and the APDT algorithms we get different trees with different random seed, whereas for ID3 and CART-AP, we get only one tree. Hence, for ID3 and CART-AP, the average tree is same as the best tree learned.

A. Synthetic Problems

Shown in Fig. 2 are three synthetic 2-class pattern classification problems over [0, 1] $\times$ [0, 1]. For each problem, we generated 1000 training samples and 500 test samples using uniform distribution. The simulations are done under no-noise conditions. Table II gives the best trees obtained (in terms of accuracy on test sample) and Table III gives average results (over five runs) for each algorithm.

B. Real-World Problems

In addition to the six synthetic problems given above, we also tested our algorithm on two problems from UCI machine-learning databases, the IRIS problem, and the WINE classification problem.

1) IRIS Problem: This domain contains three types of plants: Iris-setosa, Iris-versicolor and Iris-viginica. Each plant is characterized by four real-valued attributes (petal width, petal length, sepal width, and sepal length). It is known [11] that the class Iris-setosa is linearly separable from the other two, and that and the other two are not linearly separable. We consider the nonlinearly separable problem of determining whether or not the given plant is Iris-viginica.

Since the domain consists of 150 patterns, we divide the patterns into two parts: a training set with 120 patterns and a test set with 30 test patterns. In the APDT algorithm, the value of free parameters is chosen as given in Section III except for $\delta = 0.5$. The simulation results are given in Table IV.

2) WINE Classification Problem: These data are the result of a chemical analysis of wines grown in the same region but derived from three different cultivars. The analysis determined the quantities of 13 constituents found in each of the three types of wines, and these are provided as the values of 13 real-valued attributes. We consider the problem of classifying the first type of wine as opposed to the other two.

The domain consists of 178 patterns. We divide the data set into two sets of 103 and 75 patterns and use the first one as training and the second as a test set. Table IV summarizes the results. In the APDT algorithm, we had chosen a step size of $\delta = 0.1$. The patterns are normalized to take value from (0,

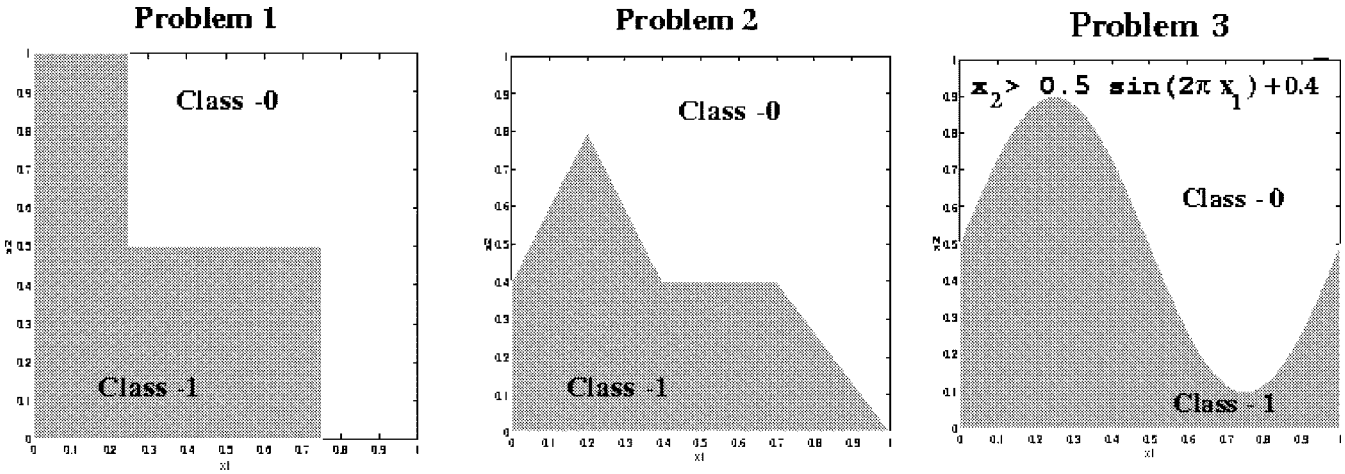


Fig. 2. Synthetic problems 1–3.

TABLE II
BEST RESULTS (OVER FIVE RUNS) FOR PROBLEMS 1–3

Classifier	Problem 1				Problem 2				Problem 3			
	% Accuracy		N	Depth	% Accuracy		N	Depth	% Accuracy		N	Depth
	Trn	Tst			Trn	Tst			Trn	Tst		
APDT	100	99.8	3	3	100	100	5	3	99.8	99.3	7	4
OC1	100	99.6	13	5	99.9	98.6	18	6	99.92	99.3	10	4
CART-LC	99.9	99.6	7	4	99	97.2	13	6	99.84	99.4	10	4

TABLE III
AVERAGE RESULTS (OVER FIVE RUNS) FOR PROBLEMS 1–3

Classifier	Problem 1				Problem 2				Problem 3			
	% Accuracy		N	Depth	% Accuracy		N	Depth	% Accuracy		N	Depth
	Trn	Tst			Trn	Tst			Trn	Tst		
APDT	99.92	99.52	4.8	3.4	99.96	99.84	5	3.4	99.14	98.52	7.6	4.2
OC1	99.35	98.8	11.6	5	98.5	97.56	9.4	4.8	99.85	99.16	10.4	4.4
CART-LC	99.76	99.28	7.4	4	98.94	96.04	13	5.6	99.75	99.0	13.8	5.4
CART-AP	100	99.8	4	3	98.7	97.2	14	6	99.88	97.1	54	8
ID3	100	100	3	3	97.4	97.4	22	10	97.2	97.2	53	12

1) by dividing each attribute by its maximum value from the training set.

C. Discussion

Out of three synthetic problems we studied, the first two had piecewise linear separating surfaces, and Problem 3 had a nonlinear separating boundary. Compared with the other tree-growing methods, it was observed that the best and average tree learned by APDT is smaller and more accurate. This justifies the motivation we had given earlier regarding degree of separability being more relevant than degree of

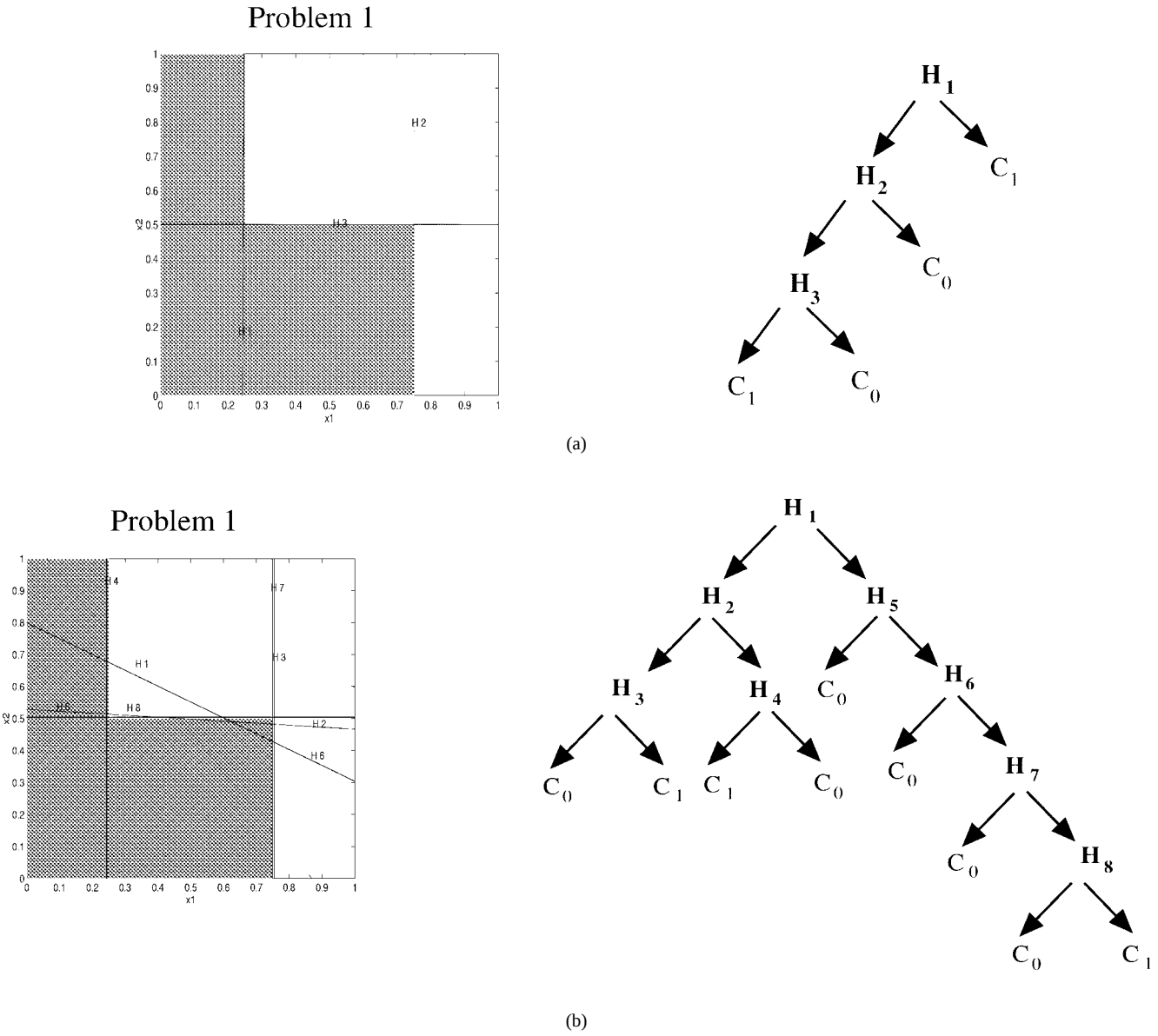


Fig. 3. Best tree learned for Problem 1 (a) APDT algorithm and (b) OC1 algorithm.

TABLE IV
AVERAGE RESULTS (OVER FIVE RUNS) FOR ML PROBLEMS

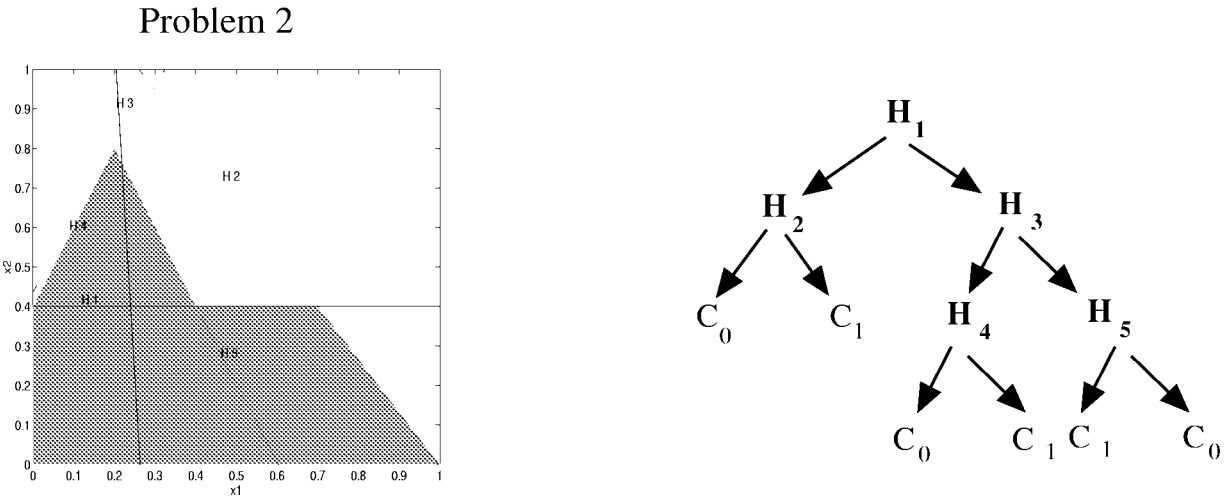
Classifier	IRIS				WINE			
	% Accuracy		N	Depth	% Accuracy		N	Depth
	Trn	Tst			Trn	Tst		
APDT	99.66	100	3.4	2.4	100	97.58	3	2
OC1	97.5	100	2.0	2.0	96.33	96.14	3.0	2.8
CART-LC	95.83	97.17	2.0	2.0	93.0	95.79	3.8	3

purity, especially in problems with piecewise-linear separating boundaries. It should be noted that in the APDT algorithm, we did not employ any pruning, while the results presented for all the other algorithms were those obtained after pruning the learned tree. Also, it is clear from the tables that axis-parallel trees are a poor choice for representing general classifiers. The APDT algorithm takes approximately twice the time taken by OC1 or CART-LC in all problems.

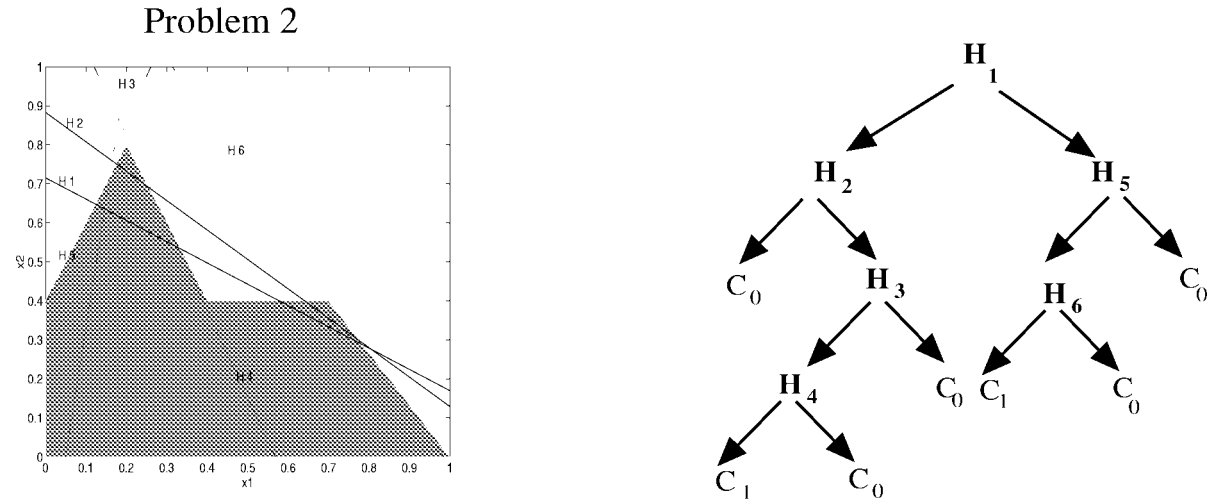
It is empirically observed that on different runs, the APDT algorithm learned trees that do not vary much in terms of size and performance, unlike methods such as CART-LC (compare the best versus average performance). This may be due to the fact that we use the Alopex algorithm for finding the best split at each node, and that this algorithm is known to be a very robust, general-purpose optimization technique [6].

Among all the other algorithms that we tried, OC1 gave the best results in terms of both accuracy and size of the tree learned. Hence, in Figs. 3 and 4 we compare the best tree learned by the APDT and OC1 for two synthetic problems[3]. The (a) pictures in Figs. 3 and 4 show the tree learned by the APDT, and the (b) pictures show the tree learned by OC1. In

[3]The best tree is in terms of accuracy on the test sample.



(a)



(b)

Fig. 4. Best tree learned for Problem 2 (a) APDT algorithm and (b) OC1 algorithm.

the figure, each split rule in the decision tree is indicated by the name of the corresponding hyperplane, and it is shown on left side of each figure. For Problem 2, the best tree learned by OC1 has 18 internal nodes, and it is complicated. Therefore, in Fig. 4 we give the smallest tree learned by OC1, the accuracy of which is much lower (98.5% on the training set and 97.4% on the test set). Figs. 3 and 4 show that the APDT learns exact boundaries of piecewise linear separators where they exist, and in general, the trees learned are more compact and accurate. This is the reason the APDT algorithm did not need pruning for these problems. But for more complicated problems, we may have to use pruning techniques to avoid overfitting problems.

IV. Decision-Tree Pruning as a Boolean-Vector Learning Problem

Most top-down, decision-tree induction methods, being greedy algorithms, generally suffer from the problem of overfitting. Pruning is a well-known method to handle this [3]. The objective of pruning is to restructure a learned decision tree so that its accuracy on test data is improved. Most standard pruning techniques achieve this by replacing selected subtrees with leaf nodes [3], [12]. In this section, we present a new class of pruning techniques that are capable of effecting a more radical restructuring of the learned tree.

In classifying a pattern using a decision tree, we follow a path from the root to a leaf node such that the pattern satisfies all the conditions in the nodes along the path. Hence, in an oblique decision tree, each such path would represent an intersection of half spaces, and thus in a decision tree, each class would be represented as a union of such convex regions in the feature space. We can use this fact to represent a decision tree as a special kind of 3-layer feedforward network.

The general structure of such a network is shown in Fig. 5. Each node in the first layer of the network (not counting the input layer) represents one hyperplane. To fix the notation, let M denote the number of first-layer units, and let the ith unit represent a hyperplane H_i parametrized by $\boldsymbol{W}_i = [w_{i0}, \cdots, w_{in}] \in \Re^{(n+1)}, 1 \leq i \leq M$ (we are assuming an n-dimensional feature space). On an input pattern $X = [x_1, x_2, \cdots, x_n] \in \Re^n$, the output of the ith unit is $y_i = 1$ if

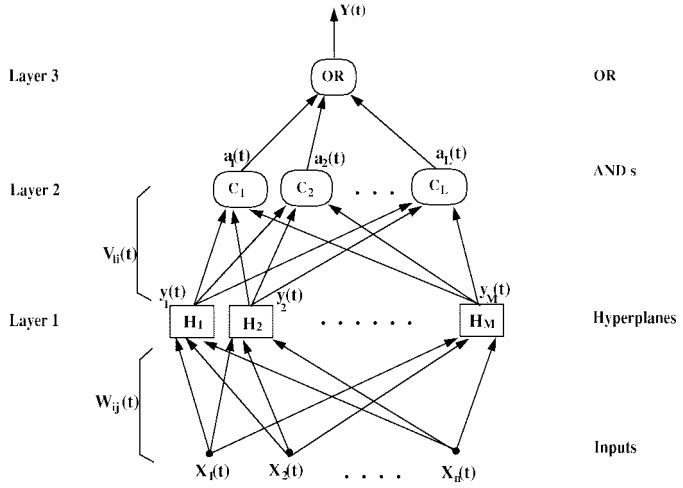


Fig. 5. Three-layer feedforward network representation.

$\sum_{j=1}^{n} w_{ij}x_j + w_{i0} > 0$ and $y_i = 0$ otherwise. Let there be L units in the second layer, each of which implements a kind of AND function. The ℓth second-layer unit is connected to all first-layer units through a weight vector $\boldsymbol{V}_\ell = [v_{\ell 1} v_{\ell 2} \cdots v_{\ell M}]$ where $v_{\ell i} \in \{0,1\} \forall i, 1 \leq \ell \leq L$. The output of the ℓth second-layer unit is $a_\ell = 1$ if $y_i = 1$ for all i, such that $v_{\ell i} = 1$, otherwise $a_\ell = 0$. The third layer consists of a single OR unit, which is connected to all second-layer units. Its output, which is the output of the network, is one if the output of at least one second-layer unit is one.

It is easy to see how any decision tree can be represented as such a 3-layer network. The first layer consists of all distinct hyperplanes (split rules) at all nodes of the decision tree and also their complements. The second layer consists of as many nodes as there are leaves in the decison tree, labeled class-1. Each second-layer unit is connected, with weight one, to all hyperplanes in the first layer that appear on the path from the root to the leaf node corresponding to this second-layer unit. All the other connections from that second-layer unit to the first-layer units will have weight zero. Now it is easy to see that output of the network will be one on all patterns that would be classified as class-1 by the decision tree.

We can use this network representation for pruning a decision tree as follows. We fix all the first-layer units so that they represent all distinct hyperplanes that are present in the learned tree and also all the complements of the hyperplanes. This fixes a value of M and all $\boldsymbol{W}_i, 1 \leq i \leq M$ in the first layer. We can fix L (the number of second-layer units) heuristically based on the pruning expected. The value of L decides the maximum possible number of paths ending in class-1 leaves in the final pruned tree. Now pruning is viewed as learning the boolean vectors $\boldsymbol{V}_\ell, 1 \leq \ell \leq L$. Any specific value for each $\boldsymbol{V}_\ell$ would represent one specific way of restructuring the learned tree (using only the hyperplanes originally learned). The goodness of any choice can be assessed using a training set of patterns. Learning optimal $\boldsymbol{V}_\ell$ vectors amounts to learning a way of choosing and combining the hyperplanes in the original tree to improve the accuracy of the learned tree. Such a pruning technique can effect a more radical restructuring of the original tree than that of simply replacing selected subtrees by leaves.

Even though there are a finite number of possible $\boldsymbol{V}_\ell$ vectors (as each $v_{\ell i} \in \{0,1\} \forall i, 1 \leq \ell \leq L$), an exhaustive search would be very expensive computationally for most problems. In what follows, we briefly discuss two stochastic algorithms for learning these Boolean vectors. The first algorithm is based on learning automata (LA) models [13], and the second is based on genetic algorithm (GA) models [14].

A. LA Pruning

The LA pruning algorithm is given in Table V. At each iteration t, the algorithm updates probabilities $q_{\ell i}(t), 1 \leq \ell \leq L, 1 \leq i \leq M$. All of these probabilities are initialized to 0.5. At each iteration t, the algorithms makes a random choice of Boolean vectors $\boldsymbol{V}_\ell(t)$ based on these probabilities. That is, Prob$[v_{\ell i}(t) = 1] = q_{\ell i}(t), \forall \ell, i$. The resulting network is used to classify the next sample pattern, and the correctness or otherwise of this classification is used to update the probabilities. The parameter λ controls the stepsize used in the updating, and it should be sufficiently small. This algorithm is a special case of a general LA algorithm [13]. The general algorithm is known to converge to parameter values that are local maxima of expected value of β calculated in step 7 of the algorithm. Hence, one can expect this pruning technique to result in compact trees with good accuracy.

B. GA Pruning

Since our pruning algorithm amounts to learning parameter values, all of which are Boolean, the problem is well suited for application of genetic algorithms [14]. In GA pruning, we used the classification accuracy on the sample achieved with any specific set of parameter values as the fitness function value for that parameter setting. We used standard crossover and mutation operators.

C. Simulation Results

In this section, we present some empirical results with our pruning technique and discuss the utility of the 3-layer network structure for pruning decision trees. We also show that both our pruning algorithms can learn a good tree even if the sample used is corrupted by classification noise.

To test the pruning technique, we have selected Problem 2, discussed in Section III-A. We have taken the initial trees as learned by OC1 and CART-LC algorithms (without any pruning) and compared the results obtained using our pruning algorithms with those obtained using the standard CC pruning technique.[4]. These results are shown in Table VI. For each pruning technique, the table shows the percentage accuracy (on both training and test sets) and the size of the learned tree in terms of number of internal nodes N. From the table, it is easy to see that the pruning techniques that use our three-layer feedforward network structure consistently outperform the standard CC pruning.

[4] As mentioned in Section III-A, the APDT algorithm learned the separating boundary to a close approximation in all the synthetic problems and that is the reason we are not testing the pruning techniques on trees grown using the APDT algorithm.

TABLE V
LA PRUNING ALGORITHM

LA Pruning Algorithm

1. Initialize all the probabilities $q_{\ell i}(0)$ and set w_{ij} parameters to learnt hyperplanes.

2. Choose $v_{\ell i}(t) \in \{0,1\}$ at random based on $q_{\ell i}(t), 1 \leq \ell \leq L, 1 \leq i \leq M$.

3. Get the next pattern $(X(t), c(t)) \in \mathcal{S}$, from the training set.

4. Calculate output for each unit in first layer using

$$y_i(t) = \begin{cases} 1, & \text{if } \sum_{j=0}^{n} w_{ij} x_j(t) > 0 \\ 0, & \text{otherwise} \end{cases} \tag{6}$$

5. Calculate the output of second layer units using

$$a_\ell(t) = \begin{cases} 1, & \text{if } y_i(t) = 1 \text{ for all } i \text{ such that } v_{\ell i}(t) = 1, \\ 0, & \text{otherwise} \end{cases} \tag{7}$$

6. The final output of the network, $Y(t)$, is

$$Y(t) = \begin{cases} 1, & \text{when any of the second layer unit outputs 1} \\ 0, & \text{otherwise} \end{cases} \tag{8}$$

7. If ($Y(t) = c(t)$) then set $\beta(t) = 1$ else $\beta(t) = 0$

8. Update the probabilities as follows :

- for $\ell = 1, \cdots, L,\ i = 1, \cdots, M$

$$q_{\ell i}(t+1) = \begin{cases} q_{\ell i}(t) + \lambda\beta(t)(1 - q_{\ell i}(t)); & \text{if } v_{\ell i}(t) = 1 \\ q_{\ell i}(t)(1 - \lambda\beta(t)); & \text{otherwise} \end{cases} \tag{9}$$

where $0 < \lambda < 1$ is a step size parameter.

9. if($\mathbf{q}(t)$ has converged) then stop else set $t = t + 1$ and goto 2.

Table VII shows results (percentage accuracy and size in terms of number of nodes N) obtained using different pruning techniques for Problem 2 when the training set is corrupted with unbiased classification noise (the percentage of noise added is shown in column 2 in Table VII). From the results in Table VII, it is easy to see that even after adding 15% of classification noise, the final pruned tree is very accurate (note that the percentage accuracy of the final tree in the table is calculated using uncorrupted samples). The last two columns in Table VII compare the time taken by our pruning techniques (t_{lrn}) with that needed for an exhaustive search over all possible Boolean vectors (t_{exh})[5].

[5]The time for exhaustive search is obtained by evaluating a fixed number of Boolean vectors on the sample set and then extrapolating the time for the total number of Boolean vectors to be considered.

Fig. 6 shows typical learning curves for LA pruning and GA pruning[6]. In these graphs, the error is with respect to the corrupted training samples, which is why, asymptotically, the error rate is the same as the noise rate.

V. CONCLUSIONS

In this paper, we have considered the problem of learning accurate and compact decision trees from a sample of preclassified pattern vectors. Most of the tree induction algorithms currently available follow the top-down recursive method. As discussed in Section I, the design of any algorithm for inducing decision trees involves two issues: choosing an evaluation

[6]Each iteration in GA pruning corresponds to one generation.

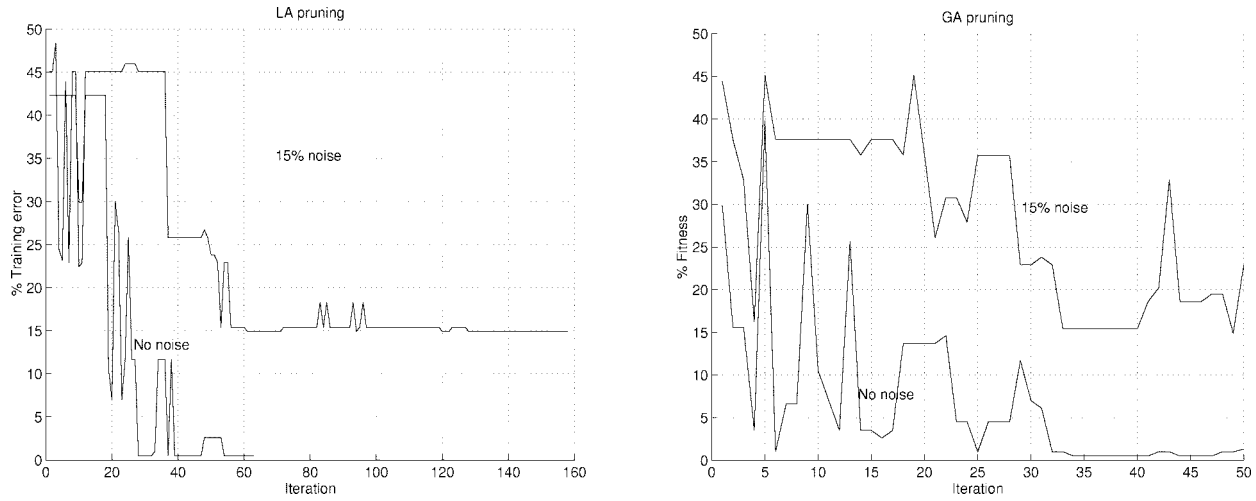


Fig. 6. Error plots for LA pruning and GA pruning.

TABLE VI
COMPARISON OF CC PRUNING, LA PRUNING, AND GA PRUNING FOR PROBLEM 2. $L = 2$ HAS BEEN CHOSEN FOR PROBLEM 2 IN LA PRUNING AND GA PRUNING, AND $\lambda = 0.003$ HAS BEEN CHOSEN IN LA PRUNING. PROBABILITY OF CROSSOVER AND MUTATION IS 0.7 AND 0.03, RESPECTIVELY, IN GA PRUNING

DT growing method	DT Pruning method	Problem 2		
		% Accuracy		N
		Trn	Tst	
OC1	NO	99.6	98.4	16
	CC	99.1	97.4	8
	LA	99.4	98.8	4
	GA	99.6	99.8	5
CART-LC	NO	100.0	99.6	8
	CC	99.6	98.4	7
	LA	99.9	99.4	3
	GA	100.0	99.8	4

TABLE VII
COMPARISON OF CC PRUNING, LA PRUNING, AND GA PRUNING ON PROBLEM 2 WITH NOISY SAMPLES

DT Pruning method	% Noise	% Accuracy		N	t_{lrn} sec	t_{exh} sec
		Trn	Tst			
NO	0	99.6	98.4	16	-	-
CC	0	99.1	97.4	8	-	-
LA	0	99.4	98.8	4	94.4	236.2×10^9
	15	99.9	98.7	10	132.8	
GA	0	99.6	99.8	5	376.5	175.9×10^{10}
	15	99.4	99.0	8	327.6	

function to quantify goodness of a split rule and choosing an optimization technique to find the best split rule at each node.

Almost all of the decision-tree algorithms in literature evaluate split rules based on the degree of purity of children nodes resulting from the split. Thus, they prefer splits that result in children nodes that have pattern sets with predominantly skewed distributions of different classes. As we argued in Section I, since each split rule in an oblique tree can be an arbitrary hyperplane, it is better to prefer split rules that result in children nodes having linearly separable subsets of patterns.

With this motivation, in this paper we presented an evaluation function that rated split rules based on the degree of linear separability of pattern-sets that go into children nodes resulting from the split. We used a heuristic estimate of the number of nonseparable patterns in a pattern-set to characterize the degree of linear separability. We used the Perceptron algorithm for estimating the number of nonseparable patterns, which is justified by theoretical results in [5].

Having decided on the evaluation function, we need an optimization algorithm to find the best split rule. For oblique split rules, brute-force optimization is clearly impractical. Because we cannot explicitly calculate the gradient of evaluation function with respect to parameters of split rule, we cannot use any standard optimization technique. Algorithms like CART and OC1 rely on perturbing each of the parameters of the hyperplane split rules individually several times to achieve the optimization. In our method, we adopted the Alopex optimization algorithm to find the best split rule. The Alopex algorithm is well suited for our problem, as it does not need any gradient information, and it exploits the correlation between changes in parameters and changes in the objective function values.

Simulation results show that the APDT algorithm learns better trees in terms of both accuracy and size compared to other oblique decision-tree induction methods. As discussed in Section III-A, the APDT algorithm learns a very good approximation to the optimal tree in problems with piecewise linear separating boundaries. Even in problems with nonlinear class boundaries, its performance is as good as or better than that of OC1 and CART. The results also show that axis-parallel trees are much poorer at representing piecewise linear class boundary compared to oblique decision trees.

The APDT algorithm presented in this paper can handle only 2-class problems. Extending this approach to multiclass problems requires further investigation. The most obvious extension is to learn one decision tree each, to separate one class from all the rest. Thus, we need to learn n different

decision trees for an n-class problem. An alternate approach would be to modify our evaluation function so that degree of separability now means degree of separability of the most predominant class from all the rest. For example, given a pattern set of S^{t_l} for estimating the number of nonseparable patterns nsp_{t_l}, we run the Perceptron algorithm with the aim of separating the most predominant class in S^{t_l} from all the rest. More empirical studies are needed to decide on the most suitable extension to handle multiclass problems.

Apart from this issue of multiclass problems, there are many other extensions possible to the basic the APDT algorithm. Our evaluation function represents a first attempt at characterizing the notion of degree of separability. A better way to characterize this notion would certainly help improve the performance of the algorithms. In our algorithm, we have not addressed issues such as handling missing attributes, achieving dimensionality reduction through feature subset selection, etc. However, such facilities easily can be incorporated in the APDT algorithm.

Most top-down, recursive-tree induction algorithms suffer from the problem of overfitting, and pruning is a well-studied method advocated to handle this. Most pruning methods improve the generalization ability of learned trees by replacing, with leaves, subtrees that do not contribute significantly to classification accuracy. In this paper, we also have presented a general pruning technique that can effect a more radical restructuring of the tree. The main idea is to represent the decision tree as a specific, 3-layer feedforward network that allows one to pick a subset of all learned hyperplanes and combine them appropriately so as to increase the accuracy of the final tree. The pruning problem here amounts to learning Boolean vectors that constitute the weights between first and second layers of the network. We have presented two specific algorithms that can learn the correct Boolean vectors efficiently. These pruning techniques, based on our representation of the decision tree as a 3-layer network, can be applied to decision trees learned using any tree-growing algorithm. The algorithm we presented is applicable only to 2-class problems. Extension of this technique to handle multiclass problems requires further investigation.

The pruning algorithms that we proposed are quite robust with respect to classification noise in the given sample of pattern vectors. This is demonstrated through our simulation results presented in Section IV-C. However, many of the tree-induction algorithms available are not very noise tolerant. One of the possible ways to design noise-tolerant tree-induction algorithms is to utilize our 3-layer network structure itself to learn the decision tree. That is, instead of fixing the hyperplanes of the first level nodes by the initially learned decision tree, as is done in the algorithm in Section IV, we could simultaneously learn the parameters of the hyperplanes in the first layer as well as the Boolean vectors that constitute the weights between the first and second layer. It is possible to use some reinforcement learning algorithms based on LA models for learning hyperplanes [13]. From our prelimenary empirical investigations, it appears that this is a difficult problem. However, we believe our network structure merits further attention from the point of view of designing robust, decision-tree learning algorithms.

REFERENCES

[1] R. R. Safavin and D. Landgrebe, "A survey of decision tree classifier methodology," *IEEE Trans. Syst., Man, Cybern.*, vol. 21, pp. 660–674, May 1991.

[2] J. R. Quinlan, "Learning efficient classification procedures and their applications to chess end games," in R. S. Michalski, J. G. Carbonell, and T. M. Mitchell, Eds., *Machine Learning: An Artificial Intelligence Approach*, vol. 1. Palo Alto, CA: Tiogo, 1983.

[3] L. Brieman, J. H. Friedman, A. Ghnene, and J. Stone, *Classification and Regression Trees*. Belmont, CA: Wadworth, 1984.

[4] S. K. Murthy, S. Kasif, and S. Salzberg, "A system for induction of oblique decision trees," *J. Artif. Intell. Res.*, vol. 2, no. 1, pp. 1–32, 1994.

[5] V. P. Roychowdhury, K.-Y. Siu, and T. Kailath, "Classification of linearly nonseparable patterns by linear threshold elements," *IEEE Trans. Neural Networks*, vol. 6, pp. 318–331, Mar. 1995.

[6] K. P. Unnikrishnaan and K. P. Venugopal, "Alopex: A correlation-based learning algorithm for feed-forward and recurrent neural networks," *Neural Comput.*, vol. 6, no. 4, pp. 469–490, 1994.

[7] P. M. Murphy and D. W. Aha, "UCI repository of machine learning databases," Machine-Readable Data Repository, Dept. Inform. Comput. Sci., Univ. California, Irvine, 1994.

[8] J. R. Quinlan, "Simplifying decision trees," *Int. J. Man-Mach. Studies*, vol. 27, no. 3, pp. 221–234, 1987.

[9] U. M. Fayyad and K. B. Irani, "On the handling of continuous valued attributes in decision tree generation," *Mach. Learn.*, vol. 8, no. 1, pp. 87–102, 1992.

[10] P. E. Utgoff, "An improved algorithm for incremental induction of decision trees," in *Proc. 11th Int. Conf. Machine Learning*, pp. 318–325, 1994.

[11] R. O. Duda and P. E. Hart, *Pattern Classification and Scene Analysis*. New York: Wiley, 1973.

[12] J. R. Quinlan, "Probabilistic decision trees," in *Machine Learning: An Artificial Intelligence Approach*, vol. 3., Y. Kodratoff and R. S. Michalski, Eds. San Mateo, CA: Morgan Kaufmann, 1990.

[13] V. V. Phansalkar and M. A. L. Thathachar, "Global convergence of feedforward networks of learning automata," in *Proc. Int. Joint Conf. Neural Networks*, Baltimore, MD, June 1992.

[14] D. Goldberg, *Genetic Algorithms in Search, Optimization and Machine Learning*. Reading, MA: Addison-Wesley, 1989.

Shesha Shah received the B.E. degree in computer science and engineering from the University of Baroda, India, in 1992 and the M.Sc. degree in electrical engineering from Indian Institute of Science, Bangalore, in 1996.

Since 1996, she has been a Research Student at the Indian Institute of Science, Bangalore, India. Her research interests include pattern recognition, computational neuroscience, and image processing.

P. S. Sastry (S'82–M'85–SM'97) received the B.Sc. (Hons.) degree in physics from the Indian Institute of Technology, Kharagpur, India, in 1978 and the B.E. degree in electrical communication engineering and the Ph.D. degree from the Indian Institute of Science, Bangalore, India, in 1981 and 1985, respectively.

He is currently an Associate Professor in the Department of Electrical Engineering, Indian Institute of Science, Bangalore. He has held visiting positions at the University of Massachusetts, Amherst, University of Michigan, Ann Arbor, and General Motors Research Laboratories, Warren, NJ. His research interests include reinforcement learning, pattern recognition, neural networks, and computational neuroscience.