On Learning Decision Trees with Large Output Domains

Nader H. Bshouty **Christino Tamon** **David K. Wilson**
Department of Computer Science
The University of Calgary
Calgary, Alberta, Canada T2N 1N4
e-mail:{bshouty, tamon, wilsond}@cpsc.ucalgary.ca

Abstract

For two disjoint sets of variables, X and Y, and a class of functions C, we define $DT(X,Y,C)$ to be the class of all decision trees over X whose leaves are functions from C over Y. We study the learnability of $DT(X,Y,C)$ using membership and equivalence queries. Boolean decision trees, $DT(X,\emptyset,\{0,1\})$, were shown to be exactly learnable in [Bs93] but does this imply the learnability of decision trees that have non-boolean leaves? A simple encoding of all possible leaf values will work provided that the size of C is *reasonable*. Our investigation involves several cases where simple encoding is not feasible, i.e., when $|C|$ is large.

We show how to learn decision trees whose leaves are learnable concepts belonging to a class C, $DT(X,Y,C)$, when the separation between the variables X and Y is known. A simple algorithm for decision trees whose leaves are constants, $DT(X,\emptyset,C)$, is also presented.

Each case above requires at least s separate executions of the algorithm from [Bs93] where s is the number of distinct leaves of the tree but we show that if C is a bounded lattice, $DT(X,\emptyset,C)$ is learnable using only one execution of this algorithm.

1 Introduction

Rooted binary trees, or decision trees, provide a natural representation both visually and conceptually for functions that classify examples of a concept. Each node holds a yes/no question concerning an example attribute and examples are classified by moving through the tree from root to leaf. Boolean decision trees hold the values 1 and 0 at their leaves indicating positive and negative membership in the given concept respectively. In terms of decision making, the nodes and edges of a boolean decision tree specify the preliminary questions necessary to arrive at a response and the leaves represent the responses themselves.

In the recent past, considerable effort has been devoted to finding learning algorithms for decision trees [Bl92,EH89,H92a,H92b,H93,KM91,R87]. Boolean decision trees were shown to be learnable in [Bs93] using a technique called the monotone theory which is based on Angluin's algorithm for learning monotone DNF formulas [A88]. Using this result we investigate some situations where the leaves of the decision trees are non-boolean.

We now give a general definition for decision trees having various types of leaves. For two disjoint sets of variables, X and Y, and a class of concepts C, we define $DT(X,Y,C)$ to be the class of all decision trees over X whose leaves are functions from C over Y. We will study the learnability of this class using membership and equivalence queries. Notice that $DT(X,\emptyset,C)$ refers to decision trees with leaves that are constant functions or just simply leaves that are members of C and $DT(\emptyset,Y,C)$ is just the class of functions from C over Y. If the learner has prior knowledge of the difference between X and Y we say that X and Y are *distinguishable.*

Allowing the leaves of the decision tree to hold non-boolean values may be thought of as representing a non-trivial action or response to a situation requiring a decision. Consider the following examples.

- An engineer wants to predict the effects of a particular force on a certain type of metal. Before this can be done a series of questions regarding the material's history (eg. manufacturing technique) must be answered to determine what function must be applied to accurately predict the result of the force.
- A physician is trying to determine an appropriate prescription for an ailing patient. After a series of personal questions for the patient the doctor may be in a position to recommend different medicines to alleviate the patient's problem. This prescription may include a schedule indicating specific dosages of certain medicines at different times.

Both of these situations have a common separation that is seen in our definition for decision trees. First, there is a set of preliminary questions that are dependent upon the subject itself. Second, the completion of these questions classifies the problem to the point where an appropriate set of instructions or calculations may be applied. We are interested in learning these types of functions but we must first show that a boolean encoding coupled with an application of the [Bs93] algorithm is not always practical for this type of problem.

Consider the problem of learning decision trees with leaves that are constant values, $DT(X, \emptyset, C)$. One approach would be to encode the set C in a boolean space such as $C \subseteq \{0,1\}^m$ where $|C| \leq 2^m$. This effectively reforms our problem as m boolean problems since $f : \{0,1\}^n \to \{0,1\}^m$ can be broken into m functions of the form $f_i : \{0,1\}^n \to \{0,1\}$ such that $f(x) = (f_1(x), f_2(x), \cdots, f_m(x))$. Such a technique is appropriate as long as the size of C is not extremely *large*. Size is obviously a factor as the variable set Y grows also. We study some situations where it is not feasible to use such an encoding and present the following results.

Theorem 1: *Decision trees with leaves from an exactly learnable concept class C, $DT(X, Y, C)$, are learnable when X and Y can be distinguished.*

Theorem 2: *Decision trees of size s with constant valued leaves over the variables $X = \{x_1, \ldots, x_n\}$, $DT(X, \emptyset, C)$, are learnable using $O(s^3)$ equivalence queries and $O(s^3n^2)$ membership queries.*

For the proofs of theorems 1 and 2 we introduce a representation for the hypothesis based on decision lists [R87]. The advantage of this is that it guarantees that during the learning process the hypothesis will evaluate to only one leaf value for each input eliminating the need to define the interaction of leaf values if an input evaluates to more than one possible output. Notice this is not an issue for the target decision trees as they are disjoint, that is, each input leads to only one leaf value in the tree.

Another issue arises with respect to our learning representation as we break the target function into separate functions and then learn each of these. This requires the simulation of membership and equivalence queries for these new functions using the oracles based on the target and we show how to do this in all cases. The equivalence query simulation is especially important because we must demonstrate that every counterexample can be used to progress the learning process.

The algorithm described for the proof of theorem 1 maintains a complete history of all values of examples seen along with other values obtained by using combinations of different X and Y values already seen. At certain points during the running of this algorithm it may be necessary to completely restart the entire learning process but we maintain the same values in the history table to ensure we don't make the same mistakes repeatedly.

One more result is presented showing a case where the efficiency of learning $DT(X, \emptyset, C)$ is improved when C is a bounded lattice.

Theorem 3: *When C forms a bounded lattice, $DT(X, \emptyset, C)$ is learnable using $O(s^2)$ equivalence queries and $O(s^2 n^2)$ membership queries where s is the number of leaves of the tree.*

This theorem follows from a generalization of the monotone theory from [Bs93] to functions whose range forms any bounded lattice.

For the algorithms of theorems 1 and 2 the hypothesis to the equivalence query and the output hypothesis are decision lists whose nodes are depth-three circuits and leafs are the hypothesis representation of class C. For the algorithm of theorem 3 the hypothesis to the equivalence query and the output hypothesis are depth-three circuits over the variable X.

The paper is organized as follows. After defining the learning model and hypothesis representation in section 2, we present the proof for theorem 2 in section 3. This provides a relatively simple introduction for the proof of theorem 1 covered in section 4. The last section describes the generalization of the monotone theory which gives us theorem 3.

2 Preliminaries

2.1 The Learning Model

We use the exact learning model as introduced by Angluin [A88] and Littlestone [L88]. A target concept, f, exists that is a member of a class of concepts $C \subseteq 2^{\{0,1\}^n}$. The learning algorithm is allowed access to certain queries that are answered by oracles with knowledge of the target concept. Those queries allowed here are as follows.

- *Membership Query*, $MQ_f(x)$: The learning algorithm supplies an element $x \in \{0,1\}^n$ as the input to a membership oracle and receives an answer $f(x)$.
- *Equivalence Query*, $EQ_f(h)$: The learning algorithm supplies a concept hypothesis h as the input to the equivalence oracle. The reply of the oracle is either *"Yes"* signifying that h is equivalent to f, or a *counterexample*, which is an element $b \in \{0,1\}^n$ such that $f(b) \neq h(b)$.

A concept class is said to be *learnable* if any concept $f \in C$ is learnable requiring time and a number of queries polynomial in n and the minimal representation size of f. For a concept representation class H, the class of concepts C is learnable from H if C is learnable when the hypothesis to the equivalence query is from H.

The goal of the learning algorithm is to output a representation, f', that is equivalent to f using polynomial time and as few queries as possible.

As we will be dealing with the simulation of queries for other functions, we will subscript the query name with the function in question to avoid confusion.

2.2 Function Representation

In this subsection we will address the issue of hypothesis representation. The size of a decision tree is the number of leaves of the tree. For each $f \in DT(X, Y, C)$ let T_f denote a minimal size decision tree for f. Let $lv(f)$ be the set of functions that appear as leaves in T_f. Each such function, $c \in lv(f)$, induces a boolean decision tree obtained from T_f by replacing each leaf of T_f with a 1 if and only if the function at that leaf is c. An example of such a tree can be found in figure 1.

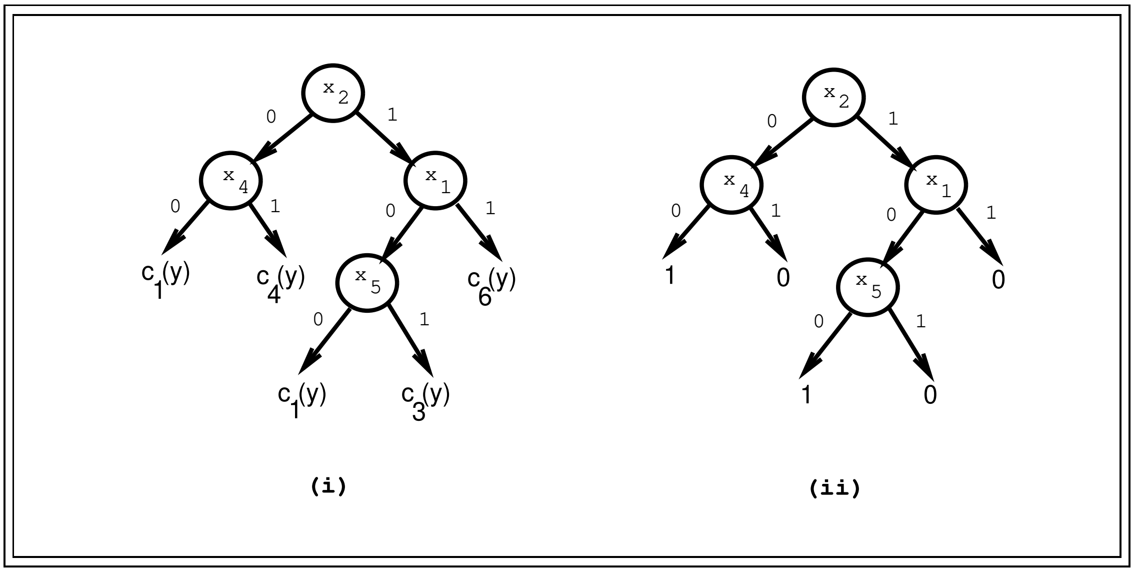


Figure 1: An example of an induced boolean decision tree on c_1.

It is easy to see that the boolean decision tree induced by $c \in lv(f)$ computes the boolean function $f_c(x) = I[f(x,y) \equiv c(y)]$ over X, where the notation $I[\text{statement}]$ equals 1 if the statement is true and 0 otherwise. Since $f(x,y)$ is a decision tree over X, each assignment for x leads to a unique leaf and therefore for every assignment x_0 there exists exactly one $c \in lv(f)$ such that $f_c(x_0) = 1$. This means if we define $c(y) + 0 = c(y)$, $c(y) \times 1 = c(y)$ and $c(y) \times 0 = 0$ for each $y \in Y$ then each $f \in DT(X,Y,C)$ admits the form $f(x,y) = \sum_{c \in lv(f)} c(y) f_c(x)$. For the purposes of learning the following decision list representation will be more useful. Ordering the set $lv(f)$ in some arbitrary way, say $lv(f) = \{c_1, c_2, \ldots c_t\}$, gives the following equivalent form of the above.

$f(x,y) =$ if $f_{c_1}(x) = 1$ then $c_1(y)$
else if $f_{c_2}(x) = 1$ then $c_2(y)$
else ...
else if $f_{c_t}(x) = 1$ then $c_t(y)$.

For brevity we will write

$$[f_{c_1}(x)c_1(y), f_{c_2}(x)c_2(y), \ldots, f_{c_t}(x)c_t(y)]$$

to mean the above decision list. Note that the size of the decision list matches the decision tree size of T_f.

3 Learning Decision Trees with Constant Valued Leaves

This section is devoted to the proof of theorem 2. We begin with a discussion about the hypothesis representation. We then describe the learning algorithm inductively showing first how to initialize the learning process and then how to proceed at any stage of the algorithm.

Theorem 2: *Decision trees of size s with constant valued leaves over the variables $X = \{x_1, \ldots, x_n\}$, $DT(X, \emptyset, C)$, are learnable using $O(s^3)$ equivalence queries and $O(s^3 n^2)$ membership queries.*

Proof We can view the target function f from $DT(X, \emptyset, C)$ being represented as

$$f(x) = [f_{c_1}(x)c_1, f_{c_2}(x)c_2, \ldots, f_{c_t}(x)c_t] \text{ where } t = |lv(f)| \leq s.$$

Since Y is empty, each c_i is just a member of the set C.

Each f_{c_i} can be seen as a boolean decision tree that indicates which leaves of the target hold the value c_i. We will learn the target function by learning each decision tree for all $f_c, c \in lv(f)$ using the decision tree algorithm from [Bs93]. For convenience, we will refer to this algorithm as $\mathcal{DTA}$.

We begin the algorithm by asking $MQ_f(0)$ where 0 is the all zero vector. The answer will be c_{i_1} which is one of the elements of $lv(f)$. Then we use the equivalence oracle to ask whether the function is the constant function c_{i_1}. If the answer is yes then we are done but if not we will receive an assignment x_0 that satisfies $f(x_0) = c_{i_2} \neq c_{i_1}$.

At some stage of the algorithm the learner will have seen t different leaf values $L = \{c_{i_1}, c_{i_2}, \ldots, c_{i_t}\}$. It assumes that these are the only leaf values in the target and proceeds to try and learn the decision trees $f_{c_{i_j}}$ for all $1 \leq j \leq t$. The learning continues as long as no counterexample contrary to this assumption is seen. If there is such a counterexample then it will be a new value $c \notin L$. This will be added to the set L and the learning continues under the previous assumption. The following will describe the above ideas in greater detail explaining the generation of decision tree algorithms and query simulation.

Membership queries for f_c for $c \in L$ are easily simulated for any vector $x_0 \in \{0,1\}^n$ as follows:

$$MQ_{f_c}(x_0) \leftarrow I[MQ_f(x_0) = c],$$

that is, we ask membership query $MQ_f(x_0)$ and if the answer is c then we return 1 and if not we return 0.

The simulation of equivalence queries is not as obvious. Assume that at some point during the learning we have t copies of $\mathcal{DTA}$ running for the values of L. We will let each copy of $\mathcal{DTA}$ run until it requests an equivalence query. If the jth $\mathcal{DTA}$ asks the equivalence query $EQ_{f_{c_{i_j}}}(h_j)$ then we ask the equivalence query $EQ_f(h)$ where

$$h(x) = [h_1(x)c_{i_1}, h_2(x)c_{i_2}, \ldots, h_t(x)c_{i_t}, 1c_*].$$

Notice the special value c_* which is any arbitrary leaf value not in the set $\{c_1, c_2, \ldots, c_t\}$. The purpose of this value is to mark the end of the decision list and the reason for doing this is to have a well defined hypothesis.

We claim that if we receive a counterexample, a, then it is either a counterexample for one of the existing decision tree algorithms or it leads to a new leaf value not currently in L. In the former case we continue running the copy of $\mathcal{DTA}$ that this is a counterexample for. In the latter case we add the new leaf value to the set L and initiate a new $\mathcal{DTA}$ for this value. This is an important claim because it ensures progress towards the learning of f for each counterexample returned until $h \equiv f$. We defend this claim by describing the three possible cases that can occur given the counterexample a. Since a is a counterexample we have $c_i = h(a) \neq f(a) = c_j$.

- Case 1: $h(a) = c_i$ for some $c_i \in L$.

  Since $f_{c_j}(a) = 1$ and since f_c for $c \in L$ are disjoint we must have $f_{c_i}(a) = 0$. Now $h_{c_i}(a) = 1$ which implies that a is a counterexample for h_{c_i}. We may get another counterexample for another algorithm in this case also. We check $MQ_f(a)$ and see if the value returned is a constant value we have already seen. If it is, we check to see whether the existing tree for this constant classifies this example correctly. If it does not then we can use it as a counterexample to this algorithm. If the value of the membership query returns a value not already seen then we can start a new algorithm for this value. If this new value is c_* then we must select a new c_*.

- Case 2: $h(a) = c_*$ and $f(a) = c_j$ where $c_j \in L$.

  This again means we have a counterexample for the algorithm learning f_{c_j} because $0 = h_{c_j}(a) \neq f_{c_j}(a) = 1$.

- Case 3: $h(a) = c_*$ and $f(a) = c$ for some $c \notin L$.

  This means we must initiate another algorithm to learn f_c.

Once all of the values in $lv(f)$ have been seen at least once then only cases 1 and 2 can occur which implies that, excluding the special value, the number of terms in h never exceeds the number of terms in f. The placement of the new terms in the hypothesis will not hinder the learning process and is therefore arbitrary.

Letting s represent the size of the target decision tree we know from [Bs93] that each copy of $DT\mathcal{A}$ will run in polynomial time using $O(s^2)$ equivalence queries and $O(s^2n^2)$ membership queries. At most we will have s copies of $DT\mathcal{A}$ to learn f so we conclude that the algorithm we have just described will run in polynomial time using $O(s^3)$ equivalence queries and $O(s^3n^2)$ membership queries. ∎

4 Learning Decision Trees With Leaves That Are Functions

The proof for theorem 1 is presented in this section. This section proceeds in a similar fashion to the previous one. The main difference being the need for additional algorithms to learn the leaf functions. Again the learning algorithm is described in an inductive manner.

Theorem 1: *Decision trees with leaves from an exactly learnable concept class C, DT(X,Y,C), are learnable where X and Y are disjoint variable sets that can be distinguished.*

Proof We will let $\mathcal{A}$ represent the learning algorithm for the concept class C and, as before, we will let $DT\mathcal{A}$ represent the boolean decision tree algorithm from [Bs93]. The target formula f can be represented in decision list form as follows

$$f(x, y) = [f_{c_1}(x)c_1(y), f_{c_2}(x)c_2(y), \ldots, f_{c_t}(x)c_t(y)]$$

where $lv(f) = \{c_1, \ldots, c_t\}$.

To learn this function, our algorithm will generate a copy of $\mathcal{A}$ and $DT\mathcal{A}$ for each c_i and f_{c_i} respectively where $1 \leq i \leq |lv(f)|$. The generation of each algorithm and simulation of membership and equivalence queries for each is not done as easily as in the previous section where the leaves were constants. We employ the help of a history table or matrix which will contain previously seen information in a form that will allow us to simulate queries and learn the above class.

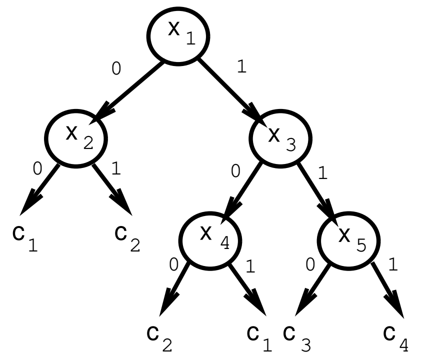
This history matrix, denoted HM, will have its rows indexed with values from Y and its columns indexed with disjoint subsets of X. Our goal is to build a table such that there will be a column for each distinct function that appears as a leaf in the target. The columns are labeled with sets containing the X values seen so far that appear to lead to the same function, that is, for each X

in the column set, the value of $f(x, y)$ will be the same for every Y value that there is a row for. Notice these x values may lead to different leaves in the target tree. Each set will be referred to by its *leader* which is any chosen member of the set.

Throughout the entire learning process the history matrix will be maintained as follows. Each time a counterexample is returned from an equivalence query, say $x'y'$, we add a row for y' and calculate its value with $MQ_f(x, y')$ for every x in each column set. If this value is the same for all x then the set remains intact. If however there is some disagreement, the set is divided into subsets which agree on this value and new columns are created for these subsets. Once this is done we add the value x' to the table by first calculating $MQ_f(x', y)$ for all the y values in the matrix. If all of these values correspond exactly to an existing column then x' is added to the set of x values that label this column. If there is no such column then a new column is added with $\{x'\}$ as its label and leader. The same procedure is also followed for every membership query that any algorithm asks on an input not yet placed in the table.

Notice that it is possible that at any time during the construction of this table that one column may actually be labeled by a set of x values that lead to different functions. This will happen when the only examples that have been seen evaluate to the same values for each of these functions. We will demonstrate that after a polynomial number of queries we can build a table that achieves our goal of having one column for each distinct function of the target. We will do this shortly but first we will show that if our goal concerning the matrix has been achieved then we can simulate any query for any copy of A or DTA allowing us to learn the target.

To illustrate the simulations, we will refer to Figure 2 which depicts a possible target tree along with a possible partial history matrix. Our discussion will be oriented towards any arbitrary hypothesis and Figure 2 will be used for specific examples. Notice that the goal of having a column for each distinct function has been achieved in Figure 2.



2^X

Y	10110 11100	11010 10011 00111	10000 01010	11111 11101
	$C_3(y_1)$	$C_1(y_1)$	$C_2(y_1)$	$C_4(y_1)$
	$C_3(y_2)$	$C_1(y_2)$	$C_2(y_2)$	$C_4(y_2)$
	$C_3(y_3)$	$C_1(y_3)$	$C_2(y_3)$	$C_4(y_3)$

Figure 2: An example target from $DT(X, Y, C)$ with a possible history matrix. Assume the leaders are the topmost element of each set.

Even though the matrix goal is complete we continue to maintain the table as before because we have no way of telling this has happened. We are however guaranteed that no additional columns will be added because each new example will have an x value that leads to one of these function which means it will agree with some column on all values of y seen so far.

This algorithm is similar to the algorithm in the previous section. We will use the same initialization procedure except this time the leaves are functions.

Assume that the current hypothesis has the following form for some $t = |lv(f)|$,

$$h(x, y) = [h_1(x)d_1(y), h_2(x)d_2(y), \ldots, h_t(x)d_t(y), 1c_*(y)]$$

where d_i is a hypothesis for c_i and h_i is a hypothesis for f_{c_i}.

Notice the special function c_* which serves a purpose similar to the special character in the hypothesis of the previous section. At this point of the learning there will be t copies of A and $DT\mathcal{A}$ running for each portion of the hypothesis. First we show how to simulate membership queries for each of these algorithms.

- Membership Queries for any f_{c_i}.

  We are trying to learn the boolean decision tree for the leaf function in question. We know that the leader of the column set, x_0, leads us to the function we want so we use this value to see if the example we want to question does also. This is done as follows.

  $$\mathrm{MQ}_{f_{c_i}}(x_1) = I[\forall y \in HM,\ \mathrm{MQ}_f(x_1, y) = \mathrm{MQ}_f(x_0, y)],$$

  that is, $f_{c_i}(x_1) = 1$ if for every y in the table $f(x_1, y) = f(x_0, y)$. This is because $f_{c_i}(x) = I[f(x, y) = c_i(y)]$ and for every $c_j(y)$, $j \neq i$ there exists a y_0 in the table such that $c_i(y_0) \neq c_j(y_0)$.

  As an example, suppose we wanted to ask a membership query for the copy of $DT\mathcal{A}$ generated by the second column of HM in Figure 2. Suppose that specifically we want the value of $MQ_{f_{c_i}}(00011)$. The leader of the column set is 11010 which leads to the function c_1 and we can see that 00011 also leads to the same function on a different leaf so they will agree on all values of Y resulting in a 1 being returned for the membership query. Had the value been 10001 for a membership query for this same algorithm, the answer would have been zero as this vector leads to a different function.

- Membership Queries for any c_i.

  Under our assumption of a column for each distinct function, the membership query for any copy of $\mathcal{A}$ is just the value of the target when the leader of the column representing this leaf function is used as the X portion of the query.

  $$\mathrm{MQ}_{c_i}(y_1) = \mathrm{MQ}_f(x_0, y_1)$$

  So say we wanted to find out what the value $c_2(y)$ was for the algorithm generated to learn this function. All we do is ask $MQ_f(10000, y)$ because this leader allows access to this specific function.

As in the previous section the simulation of the equivalence queries is not as straightforward. Similar to before we wait until each copy of an algorithm requests an equivalence query and then we use the entire hypothesis $h(x, y)$ as input to the equivalence query. In order to guarantee that the learning process progresses, we must again show that each counterexample returned will be a counterexample for at least one of the learning algorithms currently running. We proceed by analyzing all possible cases given the return of a counterexample, (x', y'), to the following equivalence query,

$$EQ_f([h_{c_1}(x)d_1(y), h_{c_2}(x)d_2(y), \ldots, h_{c_t}(x)d_t(y), 1c_*(y)]) \rightarrow (x', y').$$

We know that $f(x', y') \neq h(x', y')$. Now we find the column that corresponds to x' by checking $f(x', y)$ for all $y \in HM$. Suppose that $f(x', y) = c_{i_0}(y)$. This implies that $f_{c_j}(x') = 0$ for all $j \neq i_0$ and $f_{c_{i_0}}(x') = 1$. Now we have three cases.

- Case 1: $h_{c_k}(x') = 1$ for some $k < i_0$.

  In this case x' is a counterexample for the hypothesis h_{c_k} because $f_{c_k}(x') = 0$.

- Case 2: $h_{c_k}(x') = 0$ for all $k \leq i_0$.

  In this case x' is a counterexample for the hypothesis $h_{c_{i_0}}$ because $f_{c_{i_0}}(x') = 1$.

- Case 3: $h_{c_k}(x') = 0$ for $k < i_0$ and $h_{c_{i_0}}(x') = 1$.

  In this case y' is a counterexample for c_{i_0} because

  $$c_{i_0}(y') = f(x', y') \neq h(x', y') = d_{i_0}(y').$$

We now address the issue of showing that the history matrix can be built to the point where no more splitting or addition of a new column will occur after using a polynomial number of queries. At any point during the overall algorithm's execution it is assumed that the matrix does have a column for each distinct leaf function. We start with the assumption that there is only one such function and label this column with the all zero vector. This is the same as assuming that there is no tree portion and all we are learning is one function from C. Whenever we find it necessary to split or add a column we will start the learning process again except the start will be based on the existing history matrix. We will restart a copy of $DT\mathcal{A}$ and $\mathcal{A}$ for each column but any time we ask an equivalence query we will check to see if the hypothesis is consistent with the matrix. If it is not, then we will take our counterexample from there.

We will now show that only a polynomial number of queries are required to build a history matrix that has a column for each distinct leaf function in the target tree.

Suppose we have r columns in the table, $r < t$. Let $L_i = \{c_{i1}(y), \ldots, c_{il_i}(y)\}$ be the set of leaf functions that are consistent with column i in the table. Since each time we get a counterexample or we want to ask a membership query for column i we check all other entries in the table. If no splitting or addition of a new column happens this means that the counterexample or the answer to the membership query is valid for all c_{ij} in L_i. Therefore in column i the equivalence and membership query gives , in particular, a correct answer for c_{i1}. Since the algorithm for C runs in polynomial time we must at some point receive a counterexample that forces a splitting of the ith column or it will force a new column to be added. This only shows that the algorithms for the c's do not run more than polynomial time before a splitting or addition happens. We still need to show that the algorithms for the f_c's also do not run more than polynomial time when the number of columns is less than the size of the set $lv(f)$.

Since computing f_c is done by comparing $f(x, y)$ with the columns of the table for all Y seen so far, the function corresponding to column j will be

$$f_{L_j} = I[f(x, y) \equiv c_i(y) \text{ for some } c_i \in L_j].$$

Notice that the f_{L_j} is the decision tree resulting from the decision tree of f by replacing all leaves that are in L_j by 1 and all remaining leaves by 0. Therefore before a splitting happens to column j we will see at most a polynomial number of counterexamples for f_{L_j}.

4.1 Algorithm Complexity

We will show that the size of the table is bounded by some polynomial and therefore the algorithm will run in polynomial time. By size of the table, we refer to the ordered pairs of $X \times Y$ that it contains.

Suppose we were given an x for each distinct function in the tree, call it set X_0, along with one y value such that $f(x_i, y) \neq f(x_j, y)$ for all pairs $x_i, x_j \in X_0$. Note the size of X_0 is at most s. Since we have the information to construct a table with a column for each distinct leaf function it is clear that to learn the target function from this point would require $s(s^2n^2 + s^2)$ x values added to the table and $s(M(C) + E(C))$ y values added to the table.

We will not be given this information at the beginning of our algorithm but we have descibed how to build such a table using stages that do however make this assumption. Clearly, each stage of the algorithm requires at most this number of additions to the table and there are at most s stages so the size of the table will be at most

$$s^3((s^2n^2 + s^2)(M(C) + E(C))).$$

Since this is polynomial we conclude that the running time of the algorithm is also polynomial. ∎

5 A Generalization of the Monotone Theory

This section demonstrates the learnability of decision trees whose leaves are elements of a bounded lattice. The technique is a generalization of the ideas presented in [Bs93]. First we generalize Angluin's algorithm for learning monotone DNF boolean formulas. Next we apply the same algorithm to learn an arbitrary DNF assuming that the equivalence oracle only returns positive examples. Finally, we remove this restriction and obtain the promised learning algorithm for decision trees.

5.1 Definitions

Our functions, called *$\mathcal{L}$-functions*, are maps from the boolean n-cube $\{0,1\}^n$ into a lattice $\mathcal{L}$, $f : \{0,1\}^n \to \mathcal{L}$. Standard boolean functions are a special case where the range is $\{0,1\}$. The order present in the output helps speed up the learning of decision trees that represent $\mathcal{L}$-functions.

We begin with necessary background on ordered sets [DP]. A *partial order* $(X, \leq_X)$ is a set X with a reflexive, antisymmetric and transitive relation $\leq_X$. Two elements $x, y \in X$ are *comparable* if $x \leq_X y$ or $y \leq_X x$, and are *incomparable*, denoted by $x \parallel y$, otherwise. The notation $x <_X y$ means $x \leq_X y$ and $x \neq y$. An element x *covers* y if $x <_X y$ and there is no w such that $x <_X w <_X y$. In this case x is an *immediate descendant* of y. The set of immediate descendants of x is denoted by $\lfloor x \rfloor$.

For a subset $Q \subseteq X$, an element $x \in Q$ is a *minimal* element of Q if $x \leq_X y$ for all $y \in Q$ that are comparable to x. It is a *minimum* element if $x \leq_X y$ for all $y \in Q$. The definitions for *maximal* and *maximum* are similar. The set of *upper bounds* for a subset Q is defined as

$$Q^u = \{x \in X | (\forall q \in Q) q \leq_X x\}.$$

The set of *lower bounds* for a subset is defined as

$$Q^l = \{x \in X | (\forall q \in Q) q \geq_X x\}.$$

If the set Q^u has a minimum element, q, then q is the *supremum* of Q which is denoted by $sup\ Q$. If the set Q^l has a maximum element, q, then q is the *infimum* of Q which is denoted by $inf\ Q$.

A *lattice* $(\mathcal{L}, \leq_{\mathcal{L}})$ is a partial order where $\inf\{x, y\}$ and $\sup\{x, y\}$ exist for all $x, y \in \mathcal{L}$. The lattice $\mathcal{L}$ is *complete* if the previous fact holds for any $S \subseteq \mathcal{L}$. The two standard commutative operations on a lattice, the *meet* $\wedge$ and the *join* $\vee$, are defined through inf and sup, i.e., $x \wedge y = \inf\{x, y\}$ and $x \vee y = \sup\{x, y\}$. The minimum and maximum elements of a lattice are called the top, $\top$, and bottom, $\perp$, respectively. The following properties hold for any element x of a lattice: $\perp \wedge x = \perp$, $\perp \vee x = x$, $\top \wedge x = x$ and $\top \vee x = \top$.

We now review some standard notation for the boolean n-cube $\{0,1\}^n$. For a vector $x \in \{0,1\}^n$, we denote the i-th bit of x as $x[i]$. For two vectors x and y, $x \leq y$ if $x[i] = 1$ implies $y[i] = 1$. For $x, y \in \{0,1\}^n$, $x + y$ means the bitwise exclusive-*OR* of vectors x and y. The *Hamming weight* of $x \in \{0,1\}^n$ is the number of ones contained in vector x. The term induced by a vector $a \in \{0,1\}^n$ is defined as

$$T^a(x) = \bigwedge_{i:a[i]=1} x_i.$$

As an example, for $a = 01101$, we get $T^{01101}(x) = x_2 x_3 x_5$.

The following definition extends the notion of DNF and CNF from boolean functions to $\mathcal{L}$-functions.

Definition 1 *The formula $f = \bigvee_{i=1}^{m} (\lambda_i T_i(x))$, is called an $\mathcal{L}$-DNF where T_i is a term over n variables, $\lambda_i \in \mathcal{L}$, $x \in \{0,1\}^n$ and*

$$\lambda_i T_i(x) = \begin{cases} \lambda_i & T_i(x) = 1 \\ \perp & T_i(x) = 0 \end{cases}$$

To differentiate between different kinds of terms, the $\lambda_i T_i$'s are called $\mathcal{L}$-terms while the T_i's are simply called terms. The number of $\mathcal{L}$-terms in an $\mathcal{L}$-DNF formula is the $\mathcal{L}$-DNFsize of the formula.

Definition 2 *The formula $f = \bigwedge_{i=1}^{m} (\lambda_i \vee C_i(x))$, is called an $\mathcal{L}$-CNF where C_i is a clause over n variables, $\lambda_i \in \mathcal{L}$, $x \in \{0,1\}^n$ and*

$$\lambda_i \vee C_i(x) = \begin{cases} \lambda_i & C_i(x) = 0 \\ \top & C_i(x) = 1 \end{cases}$$

Analogous to $\mathcal{L}$-DNF's, the C_i's are called clauses and the $\lambda_i \vee C_i$'s are called $\mathcal{L}$-clauses. The $\mathcal{L}$-CNFsize of an $\mathcal{L}$-function is the number of $\mathcal{L}$-clauses contained in the $\mathcal{L}$-CNF representing it.

Just as boolean decision trees have a natural DNF and CNF representation, decision trees with leaves from $\mathcal{L}$ have a natural $\mathcal{L}$-DNF and $\mathcal{L}$-CNF form. For example the $\mathcal{L}$-DNF and $\mathcal{L}$-CNF forms of Figure 1, under the assumption that the c's are from a lattice, are respectively,

$$f_{\mathcal{L}\text{-DNF}} = c_1(\bar{x}_2 \bar{x}_4) \vee c_4(\bar{x}_2 x_4) \vee c_1(x_2 \bar{x}_1 \bar{x}_5) \vee c_3(x_2 \bar{x}_1 x_5) \vee c_6(x_2 x_1),$$

$$f_{\mathcal{L}\text{-CNF}} = [c_1 \vee (x_2 \vee x_4)] \wedge [c_4 \vee (x_2 \vee \bar{x}_4)] \wedge [c_1 \vee (\bar{x}_2 \vee x_1 \vee x_5)] \wedge [c_3 \vee (\bar{x}_2 \vee x_1 \vee \bar{x}_5)] \wedge [c_6 \vee (\bar{x}_2 \vee \bar{x}_1)].$$

Definition 3 *An $\mathcal{L}$-function f is called* monotone *when* $\forall x, y \in \{0,1\}^n$,

$$x \leq y \Longrightarrow f(x) \leq_{\mathcal{L}} f(y).$$

When dealing with monotone DNF, we will write MDNF. All previous definitions containing $\mathcal{L}$-DNF also hold for $\mathcal{L}$-MDNF.

Definition 4 *An $\mathcal{L}$-DNF is called* reduced *if it has a minimal $\mathcal{L}$-DNFsize. We also assume that each term has minimal size, i.e., as few literals as possible.*

5.2 Learning Monotone $\mathcal{L}$-DNF Formulas

This section presents a learning algorithm for $\mathcal{L}$-MDNF formulas. The algorithm is based on Angluin's algorithm for learning monotone DNF boolean formulas [A88].

We begin by showing that any $\mathcal{L}$-MDNF has a unique reduced representation and then we describe an algorithm that learns this representation exactly.

Definition 5 *The vector $x \in \{0,1\}^n$ is a* minterm *of the monotone $\mathcal{L}$-function f if*

$$\bigvee_{y \in \lfloor x \rfloor} f(y) <_{\mathcal{L}} f(x).$$

The set of minterms of f is denoted by Min(f).

The next lemma gives *one* $\mathcal{L}$-MDNF representation for each $\mathcal{L}$-monotone function.

Lemma 5.1 *If f is a monotone $\mathcal{L}$-function then $\bigvee_{a \in Min(f)} f(a)T^a(x)$ is an $\mathcal{L}$-MDNF for f.*

Proof Let $h(x) = \bigvee_{a \in Min(f)} f(a)T^a(x)$. First we show that $\forall x_0 \in \{0,1\}^n$, $h(x_0) \leq_{\mathcal{L}} f(x_0)$. Let $x_0 \in \{0,1\}^n$ be any arbitrary vector such that $h(x_0) >_{\mathcal{L}} \perp$. If one does not exist the result is obvious, otherwise we have the following

$$\begin{aligned} h(x_0) &= \bigvee_{a \in Min(f)} f(a)T^a(x_0) \\ &= \bigvee_{\substack{a \in Min(f) \\ a \leq x_0}} f(a) \\ &\leq_{\mathcal{L}} f(x_0). \end{aligned}$$

Now using induction on the Hamming weight of x_0, it will be shown that $\forall x_0 \in \{0,1\}^n$, $f(x_0) \leq_{\mathcal{L}} h(x_0)$.

Base Case: Let the Hamming weight of x_0 be 0. Obviously $f(x_0) \leq_{\mathcal{L}} h(x_0)$.

Inductive Hypothesis: For all vectors x_0, whose Hamming weights are less than k, let $f(x_0) \leq_{\mathcal{L}} h(x_0)$.

Inductive Step: Show that $f(x_0) \leq_{\mathcal{L}} h(x_0)$ for any vector x_0 that has a Hamming weight of k. If there is no x_0 such that $f(x_0) >_{\mathcal{L}} \perp$ then the result is obvious. Otherwise there are two cases to consider.

Case 1: If $x_0 \in Min(f)$ then

$$f(x_0) \leq_{\mathcal{L}} f(x_0) \vee \bigvee_{\substack{a \in Min(f) \\ a < x_0}} f(a) = h(x_0).$$

Case 2: If $x_0 \notin Min(f)$ then

$$\begin{aligned}
f(x_0) &\leq_{\mathcal{L}} \bigvee_{y \in \lfloor x_0 \rfloor} f(y) \\
&= \bigvee_{y \in \lfloor x_0 \rfloor} h(y) \quad \text{inductive hypothesis} \\
&= \bigvee_{y \in \lfloor x_0 \rfloor} \bigvee_{\substack{a \in Min(f) \\ a \leq y}} f(a) \\
&= \bigvee_{\substack{a \in Min(f) \\ a \leq x_0}} f(a) \quad \text{since } x_0 \notin Min(f) \\
&= h(x_0).
\end{aligned}$$

So for all $x_0 \in \{0,1\}^n$, $f(x_0) = h(x_0)$. ∎

The next lemma provides a lower bound on the size of any reduced $\mathcal{L}$-MDNF for $\mathcal{L}$-monotone functions.

Lemma 5.2 *If f_r is a reduced $\mathcal{L}$-MDNF for an $\mathcal{L}$-function f then*

$$\mathcal{L}\text{-}DNFsize(f_r) \geq |Min(f)|.$$

Proof Let $g = \bigvee_{i=1}^{m} \lambda_i T^{v_i}(x)$ be an $\mathcal{L}$-MDNF for f with $m < |Min(f)|$. Because of this size restriction there must be a vector $a \in Min(f)$ such that for all $T^a \neq T^{v_i}$. Since a is a minterm and f is monotone, we have $f(a) > \bigvee_{b<a} f(b)$. Now note that

$$g(a) = \bigvee_{i=1}^{m} \lambda_i T^{v_i}(a) = \bigvee_{v_i \leq a} \lambda_i = \bigvee_{v_i < a} \lambda_i,$$

as $v_i \neq a$, for all i. But then

$$\bigvee_{b<a} g(b) = \bigvee_{b<a} \bigvee_{v_i \leq b} \lambda_i = \bigvee_{v_i < a} \lambda_i,$$

which implies $g(a) = \bigvee_{b<a} g(b)$, a contradiction. ∎

Combining Lemma 5.1 and Lemma 5.2, allows us to conclude that the size of a reduced $\mathcal{L}$-MDNF equals the number of minterms. Moreover, the $\mathcal{L}$-MDNF stated in Lemma 5.1 *is* the unique representation of this size.

Lemma 5.3 *All monotone $\mathcal{L}$-functions have a unique reduced $\mathcal{L}$-MDNF.*

Proof Let $f = \bigvee_{i=1}^{m} \lambda_i T_i$ be a reduced monotone $\mathcal{L}$-DNF. The *portrait* $\mathcal{P}(f, \leq_{\mathcal{P}})$ is a partially ordered subset of $\mathcal{L} \times \{0,1\}^n$ where $\leq_{\mathcal{P}}$ is defined as

$$(\lambda_i, T_i) \leq_{\mathcal{P}} (\lambda_j, T_j) \iff \lambda_i \leq_{\mathcal{L}} \lambda_j \text{ and } T_i \geq T_j.$$

Specifically,

$$\mathcal{P}(f, \leq_{\mathcal{P}}) = \{(\lambda_i, T_i) \mid \lambda_i T_i \text{ is an } \mathcal{L}\text{-term of } f \text{ for } 1 \leq i \leq m\}.$$

LEARN-$\mathcal{L}$-MDNF

(1) $h \leftarrow \perp$;
(2) $EQ(h) \rightarrow v$. If the answer is Yes then stop;
(3) Walk v down while $\exists w \in \lfloor v \rfloor$ so that $h(w) \not\geq_{\mathcal{L}} f(w)$;
(4) $h \leftarrow h \vee f(v)T^v$;
(5) Go to step 2;

Figure 3: An algorithm that learns monotone $\mathcal{L}$-functions.

An element $x \in \mathcal{P}(f, \leq_{\mathcal{P}})$ is at *level* k if the shortest downward path from x to a minimal element of $\mathcal{P}(f, \leq_{\mathcal{P}})$ has k edges.

Assume that a monotone $\mathcal{L}$-function f has two reduced $\mathcal{L}$-DNF representations, f_{r1} and f_{r2}. To prove $f_{r1} = f_{r2}$, it will be shown that $\mathcal{P}(f_{r1}, \leq_{\mathcal{P}}) = \mathcal{P}(f_{r2}, \leq_{\mathcal{P}})$. Define A_i and B_i to be the i^{th} level elements of $\mathcal{P}(f_{r1}, \leq_{\mathcal{P}})$ and $\mathcal{P}(f_{r2}, \leq_{\mathcal{P}})$ respectively. Let $\bigvee A_i$ and $\bigvee B_i$ represent the disjunction of the elements contained within the specified set. Using this notation f_{r1} and f_{r2} can be represented as

$$f_{r1} = \bigvee_{i=1}^{t} A_i \quad \text{and} \quad f_{r2} = \bigvee_{i=1}^{s} B_i$$

where s and t are the number of distinct levels that the elements of each set occupy.

Using induction on increasing level numbers, the equality of sets A_i and B_i shall be proven. For the induction, the element $(\perp, T_{0^n})$ is added to the portraits of both f_{r1} and f_{r2}. Define artificially $A_{-1} = B_{-1} = \{(\perp, T_{0^n})\}$.

Base case: For level $i = -1$, $A_{-1} = B_{-1}$ by definition.

Inductive Hypothesis: Assume $A_i = B_i$ for all levels $i \leq k - 1$.

Inductive step: Show that $A_k = B_k$. Assume for contradiction that $A_k \neq B_k$ and let $\lambda T \in A_k \backslash B_k$. Let g represent the disjunction of all elements from levels $i \leq k - 1$.

- *Case 1:* There exists $bT^v \in B_k$ but $b \neq a$.
  This is a contradiction since $f_{r1}(v) = g(v) \vee a \neq g(v) \vee b = f_{r2}(v)$.
- *Case 2:* There exists $bT^{v'} \in B_k$ with $v' < v$.
  This is a contradiction since $f_{r1}(v') = g(v') \neq g(v') \vee b = f_{r2}(v')$.
- *Case 3:* There is no $bT^{v'} \in B_k$ with $v' < v$.
  This is a contradiction since $f_{r1}(v) = g(v) \vee a \neq g(v) = f_{r2}(v)$.

There are no other cases so $A_k = B_k$. ∎

Now that we have proven that the reduced form of an $\mathcal{L}$-MDNF for an $\mathcal{L}$-monotone function is unique we present an algorithm that learns any function from this class exactly and outputs this unique representation. This algorithm, LEARN-$\mathcal{L}$-MDNF, is given in Figure 3, and its correctness is claimed in the next lemma.

Lemma 5.4 *LEARN $\mathcal{L}$-MDNF exactly learns any monotone $\mathcal{L}$-function, f, using s EQ's and sn MQ's where s is the $\mathcal{L}$-MDNFsize of f.*

We divide the proof of the above into two lemmas.

Lemma 5.5 *Let $f = \bigvee_{i=1}^{m} \lambda_i T_i$ be a reduced monotone $\mathcal{L}$-MDNF. Then*

1. *$I[f(x) \neq \bot] = \bigvee_{i=1}^{m} T_i(x)$.*
2. *Suppose $h = \bigvee_{i=1}^{r} \lambda_i T_i$ and $g = \bigvee_{i=r+1}^{m} \lambda_i T_i$. Then*
   (a) *$I[h(x) \not\geq_{\mathcal{L}} f(x)] \leq I[g(x) \neq \bot]$.*
   (b) *If $x_0 \in \{0,1\}^n$ is minimal in satisfying $h(x_0) \not\geq_{\mathcal{L}} f(x_0)$, then x_0 is a minterm of g.*

Proof

(1) Notice that $f(x) \neq \bot$ if and only if one of the $T_i(x)$, $1 \leq i \leq m$, is 1. This implies the result.

(2a) Let x_0 be such that $h(x_0) \not\geq_{\mathcal{L}} f(x_0)$. Assume that $g(x_0) = \bot$. Then $f(x_0) = h(x_0) \not\geq_{\mathcal{L}} f(x)$ which is a contradiction.

(2b) Assume that x_0 is minimal in satisfying $h(x_0) \not\geq_{\mathcal{L}} f(x_0)$, i.e., $\forall y \in \lfloor x_0 \rfloor$, $h(y) \geq_{\mathcal{L}} f(y)$. Since $\forall x \in \{0,1\}^n$, $h(x) \leq_{\mathcal{L}} f(x)$, the statement $\forall y \in \lfloor x_0 \rfloor$, $h(y) \geq_{\mathcal{L}} f(y)$ implies that $\forall y \in \lfloor x_0 \rfloor$, $h(y) = f(y)$.

Assume x_0 is not a minterm of g, that is, $\bigvee_{y \in \lfloor x_0 \rfloor} g(y) \geq_{\mathcal{L}} g(x_0)$. Since $h(y) = f(y)$, for all $y \in \lfloor x_0 \rfloor$, it follows that $g(y) \leq_{\mathcal{L}} f(y) = h(y)$, $\forall y \in \lfloor x_0 \rfloor$. Therefore

$$g(x_0) \leq_{\mathcal{L}} \bigvee_{y \in \lfloor x_0 \rfloor} g(y) \leq_{\mathcal{L}} \bigvee_{y \in \lfloor x_0 \rfloor} h(y) = h(x_0)$$

and

$$f(x_0) = g(x_0) \vee h(x_0) = h(x_0),$$

which contradicts the assumption that $h(x_0) \not\geq_{\mathcal{L}} f(x_0)$ meaning x_0 is a minterm of g. Since g is monotone, lemma 5.3 implies that T^{x_0} is a term in g. ∎

Lemma 5.4(1) shows that after the first equivalence query in the algorithm we will find some minterm of the target f. Lemma 5.4(2) shows that after finding some of the minterms, the next equivalence query will lead to a new minterm that has not been found before.

Lemma 5.6 *Let $f(x) = \bigvee_{i=1}^{m} \lambda_i T_i(x)$ be the reduced monotone $\mathcal{L}$-DNF representing the target function and let $h(x)$ be the hypothesis of LEARN $\mathcal{L}$-MDNF. Then at each iteration*

1. *For some $0 \leq r \leq m$, $h(x) = \vee_{i \in R} \lambda_i T_i(x)$, where $|R| = r$ and $r < m$.*
2. *There is a $j > r$, such that $T^w = T_j$, where w is the stopping point of the walk down.*

Proof The proof is an easy induction based on the previous lemma. ∎

The complexity of the algorithm depends on the number of membership queries asked in step 3 where the walking down procedure takes place. Exactly as in Angluin's paper [A88], the walking down procedure is done by flipping each one bit to zero one at a time and checking the value of the function with a membership query. Note that if we flip a bit and do not move down then we will never try to flip that bit again. This implies that the number of membership queries for each walk down is bounded by n. Each equivalence query coupled with a walk down from the returned counterexample will add an $\mathcal{L}$-term to the hypothesis implying that we will need s equivalence queries, where s is the $\mathcal{L}$-MDNF size of the target.

5.3 Learning the Monotone of $\mathcal{L}$-DNF Formulas using Superset Queries.

This section shows that we learn *the* minimal monotone $\mathcal{L}$-function that covers the target $\mathcal{L}$-function f if the equivalence query oracle for f is restricted to only return *positive* counterexamples. This section generalizes the notion of $\mathcal{M}(f)$ defined in [Bs93] to $\mathcal{L}$-functions.

Definition 6 *The* monotone of an $\mathcal{L}$-function f*, denoted $\mathcal{M}(f)$, is the minimal monotone $\mathcal{L}$-function that satisfies*

$$\forall x \in \{0,1\}^n,\ f(x) \leq_{\mathcal{L}} \mathcal{M}(f)(x).$$

The following lemma shows that $\mathcal{M}(f)$ is unique.

Lemma 5.7 *For any $\mathcal{L}$-function f there is only one $\mathcal{M}(f)$.*

Proof Suppose there are two minimal monotone functions for f, call them g and h. Since $\forall x \in \{0,1\}^n$, $f(x) \leq_{\mathcal{L}} g(x)$ and $f(x) \leq_{\mathcal{L}} h(x)$ then $\forall x \in \{0,1\}^n$, $f(x) \leq_{\mathcal{L}} g(x) \wedge h(x)$ which is a more minimal function and a contradiction to the assumption. ∎

Lemma 5.8 *The monotone of f admits the following representation.*

$$\mathcal{M}(f)(x) = \bigvee_{y \leq x} f(y) = \bigvee_{y} f(y) T^y(x).$$

Proof Let $h(x) = \bigvee_{y \leq x} f(y)$. Notice that h is monotone and $\forall x \in \{0,1\}^n$, $f(x) \leq_{\mathcal{L}} h(x)$. Assume that there is another monotone $\mathcal{L}$-function g which satisfies $\forall x \in \{0,1\}^n$, $f(x) \leq_{\mathcal{L}} g(x) <_{\mathcal{L}} h(x)$. Then

$$h(x) = \bigvee_{y \leq x} f(y) \leq_{\mathcal{L}} \bigvee_{y \leq x} g(y) = g(x)$$

which is a contradiction so h is the unique minimal monotone that includes f. ∎

Lemma 5.9 *The following properties of $\mathcal{M}$ hold for any $\mathcal{L}$-functions g and f.*

1. *If $f \leq_{\mathcal{L}} g$ then $\mathcal{M}(f) \leq_{\mathcal{L}} \mathcal{M}(g)$.*
2. *$\mathcal{M}(f \wedge g) \leq \mathcal{M}(f) \wedge \mathcal{M}(g)$.*
3. *$\mathcal{M}(f \vee g) = \mathcal{M}(f) \vee \mathcal{M}(g)$.*
4. *If $f = \bigvee_i \lambda_i T_i$ then $\mathcal{M}(f) = \bigvee_i \lambda_i \mathcal{M}(T_i)$.*
5. *If f is an $\mathcal{L}$-monotone function then $\mathcal{M}(f) = f$.*

Proof (1,3) This is immediate from Lemma 5.7. A more general statement is proven for (2). Let f_i be a $\mathcal{L}$-function, for $i \in [n]$, and let $g = \bigwedge_i f_i$. Note that $g \leq_{\mathcal{L}} f_i$ for all $i \in \{1, 2, .., n\}$. Thus by (1), $\mathcal{M}(g) \leq_{\mathcal{L}} \mathcal{M}(f_i)$ for all i and hence $\mathcal{M}(g) \leq_{\mathcal{L}} \bigwedge_i \mathcal{M}(f_i)$. (4) By (3), $\mathcal{M}(f) = \bigvee_i \mathcal{M}(\lambda_i T_i)$ and $\mathcal{M}(\lambda_i T_i) = \lambda_i \mathcal{M}(T_i)$ by the definition of $\mathcal{M}$. (5) is obvious. ∎

The following definition will prove to be useful in illustrating the working of our main algorithm.

Definition 7 Superset Query, *SQ(h): The learning algorithm supplies a concept hypothesis h as the input to the superset oracle. The reply of the oracle is either "Yes" signifying that all satisfying assignments of h form a superset of all satisfying assignments of f, or a* counterexample, *which is an element $b \in \{0,1\}^n$ such that $f(b) >_{\mathcal{L}} h(b)$.*

Our goal is to show that if we replace the equivalence query oracle with the superset query oracle in LEARN-$\mathcal{L}$-MDNF then it will learn $\mathcal{M}(f)$. First we extend the definition of minterm for all $\mathcal{L}$-functions.

Definition 8 *The vector $x \in \{0,1\}^n$ is called a* minterm *of the $\mathcal{L}$-function f if*

$$\bigvee_{y \in \lfloor x \rfloor} f(y) \not\geq_{\mathcal{L}} f(x).$$

The set of minterms of f is denoted by $Min(f)$.

The next lemma provides a representation of $\mathcal{M}(f)$ in terms of the minterms of f.

Lemma 5.10 *Let f be any $\mathcal{L}$-function. Then*

$$\mathcal{M}(f) = \bigvee_{a \in Min(f)} f(a) T^a.$$

Proof By Lemma 5.7

$$\mathcal{M}(f)(x) = \bigvee_{y} f(y) T^y(x).$$

If $y \notin Min(f)$ then $\bigvee_{b \in \lfloor y \rfloor} f(b) \geq_{\mathcal{L}} f(y)$ and

$$\bigvee_{y} f(y) T^y(x) \leq_{\mathcal{L}} \bigvee_{y \in Min(f)} f(y) T^y(x).$$

Therefore $\mathcal{M}(f)(x) \leq \bigvee_{y \in Min(f)} f(y) T^y(x)$. The other inequality follows from

$$\bigvee_{y \in Min(f)} f(y) T^y(x) \leq_{\mathcal{L}} \bigvee_{y} f(y) T^y(x) = \mathcal{M}(f)(x).$$

∎

Next we want to show that $|Min(f)|$ is bounded from above by the $\mathcal{L}$-DNF size of f.

Lemma 5.11 *Let f be an $\mathcal{L}$-function. Then*

$$|Min(f)| \leq \mathcal{L}\text{-DNF size of } f.$$

Proof Suppose $f(x) = \bigvee_{i=1}^{s} \lambda_i T_i(x)$. Let $a \in Min(f)$. Without loss of generality we assume that $T_1(a) = \ldots = T_r(a) = 1$ and $T_{r+1}(a) = \ldots = T_s(a) = 0$. This implies that $f(a) = \lambda_1 \vee \ldots \vee \lambda_r$. Suppose that for all i, $1 \leq i \leq r$, there is a variable x_{j_i} not in T_i such that $a_{j_i} = 1$. If we show that this cannot happen then there exists a term T_{i_0} such that if x_i is not a variable of T_{i_0} then $a_i = 0$. Since $T_{i_0}(a) = 1$ this is equivalent to saying that a is a minterm of T_{i_0}. Therefore every minterm of f is a minterm of one of the T_i's which implies that the number of minterms of f cannot be larger than the number of terms in f. Now we show that the above assumption leads to a contradiction.

Let b^{j_i} be the assignment a when we flip the entry j_i from 1 to 0. Obviously, $T_i(b^{j_i}) = 1$, and therefore $f(b^{j_i}) \geq \lambda_i$. This implies that

$$f(a) = \lambda_1 \vee \ldots \vee \lambda_r \leq \bigvee_{i} f(b^{j_i}) \leq \bigvee_{b \in \lfloor a \rfloor} f(b),$$

which contradicts the fact that $a \in Min(f)$. ∎

We are ready to prove the main lemma of this section.

Lemma 5.12 *If LEARN-$\mathcal{L}$-MDNF uses superset queries instead of equivalence queries, then it learns $\mathcal{M}(f)$. The number of superset queries required is s and the number of membership queries required is $n^2 s$ where s is $\mathcal{L}$-DNFsize(f).*

Proof Notice that in the algorithm the hypothesis is of the form $\bigvee_{v\in V} f(v)T^v$, for some set of assignments V. Therefore the hypothesis of the algorithm is always monotone and by Lemma 5.7 we have

$$h \leq_{\mathcal{L}} \mathcal{M}(f).$$

When the algorithm stops, we will have a hypothesis h_0 such that $h_0 \geq_{\mathcal{L}} f$. Since $\mathcal{M}(f) \geq_{\mathcal{L}} h_0 \geq_{\mathcal{L}} f$, h_0 is monotone, and $\mathcal{M}(f)$ is the minimal monotone that is greater than f, we have $h_0 = \mathcal{M}(f)$.

Let $f(x) = \bigvee_{i=1}^{s} \lambda_i T_i(x)$. We will show that the hypothesis is of the form $h(x) = \bigvee_{j=1}^{r} \lambda_{i_j} \mathcal{M}(T_{i_j})(x)$. We will show that if the superset query oracle returns a counterexample then after running step 3 we add to h the term $\lambda_{i_{r+1}} \mathcal{M}(T_{i_{r+1}})$, for $i_{r+1} \notin \{i_1, \ldots, i_r\}$. Since by Lemma 5.8, $\mathcal{M}(f) = \bigvee_i \lambda_i \mathcal{M}(T_i)$, the algorithm will stop and output $\mathcal{M}(f)$ after at most s superset queries.

Assume, without loss of generality, that $h(x) = \bigvee_{i=1}^{r} \lambda_i \mathcal{M}(T_i)(x)$. Suppose in step 3 the algorithm stops at $v = v_0$. Then by the condition in the algorithm we have

$$h(v_0) \not\geq_{\mathcal{L}} f(v_0) \quad \text{and} \quad \forall w \in \lfloor v_0 \rfloor h(w) \geq_{\mathcal{L}} f(w).$$

Again without loss of generality let $\mathcal{M}(T_1)(v_0) = \cdots = \mathcal{M}(T_l)(v_0) = 1$ and $\mathcal{M}(T_{l+1})(v_0) = \cdots = \mathcal{M}(T_r)(v_0) = 0$. The latter implies that $T_{l+1}(v_0) = \cdots = T_r(v_0) = 0$. Suppose that $T_i(v_0) = \delta_i$, for $i \in \{1, \ldots, l\}$. Then $h(v_0) = \lambda_1 \vee \cdots \vee \lambda_l$ and $f(v_0) = \lambda_1\delta_1 \vee \cdots \vee \lambda_l\delta_l \vee \lambda_{r+1}T_{r+1}(v_0) \vee \cdots \vee \lambda_s T_s(v_0)$. Since $h(v_0) \not\geq_{\mathcal{L}} f(v_0)$ we must have $T_j(v_0) = 1$, for some $j > r$, and $\lambda_j \not\leq_l f \bigvee_{i=1}^{l} \lambda_i$. Because for all $w \in \lfloor v_0 \rfloor$, $h(w) \geq_{\mathcal{L}} f(w)$, we must have for every $w \in \lfloor v_0 \rfloor$, $T_j(w) = 0$. Therefore v_0 is a minterm of T_j and the term that is added to the hypothesis is $\lambda_j \mathcal{M}(t_j)$. ∎

5.4 Learning $\mathcal{L}$-Functions

This section proves that subclasses of $\mathcal{L}$-functions with small $\mathcal{L}$-monotone basis are learnable. In particular, decision trees whose leaves form a bounded lattice are learnable. The next definition extends the notion of an $\mathcal{L}$-monotone basis.

Definition 9 *A subset $A \subset \{0,1\}^n$ is called an $\mathcal{L}$-*monotone basis *($\mathcal{LM}$-basis) of f if there exists an $\mathcal{L}$-CNF representation of f such that for every clause, $\lambda_i \vee C_i$, in this $\mathcal{L}$-CNF, there is an $a_j \in A$ such that $C_i(a_j) = 0$.*

As in [Bs93] we consider the shifted boolean cube $(\{0,1\}^n, \leq_a)$, where $a \in \{0,1\}^n$ and the relation $\leq_a$ is defined as follows.

$$x \leq_a y \iff x + a \leq y + a.$$

The function f is a-monotone if $f(x+a)$ is monotone. We can now define $\mathcal{M}_a(f)$ in a similar manner as $\mathcal{M}(f)$ except $\{0,1\}^n$ is ordered with respect to $\leq_a$. Moreover we have the following relation between $\mathcal{M}_a$ and $\mathcal{M}$.

Lemma 5.13 *$\mathcal{M}_a(f)(x) = \mathcal{M}(f(x+a))(x+a)$.*

Proof Let $h(x) = f(x+a)$. Then we have

$$\begin{aligned} \mathcal{M}_a(f)(x) &= \bigvee_{y \leq_a x} f(y) = \bigvee_{y+a \leq x+a} f((y+a)+a) \\ &= \bigvee_{z \leq x+a} h(z) = \mathcal{M}(h)(x+a) \\ &= \mathcal{M}(f(x+a))(x+a). \end{aligned}$$

∎

The following lemma generalizes the main representation theorem of the monotone theory in [Bs93].

Lemma 5.14 *If A is an $\mathcal{L}$-monotone basis of f then $f(x) = \bigwedge_{a \in A} \mathcal{M}_a(f)(x)$.*

Proof First note that $f(x) \leq_{\mathcal{L}} \mathcal{M}_a(f)(x)$ for all $x \in \{0,1\}^n$ and $a \in A$. This implies that $f(x) \leq_{\mathcal{L}} \bigwedge_{a \in A} \mathcal{M}_a(f)(x)$. For the converse direction let $g = \bigwedge_{i=1}^m (\lambda_i \vee C_i)(x)$ such that for every C_i there exists $a \in A$ where $C_i(a) = 0$. Then

$$\begin{aligned} \bigwedge_{a \in A} \mathcal{M}_a(f) &= \bigwedge_{a \in A} \mathcal{M}_a(g)(x) \\ &= \bigwedge_{a \in A} \mathcal{M}_a \left(\bigwedge_{i=1}^m (\lambda_i \vee C_i) \right)(x) \\ &\leq_{\mathcal{L}} \bigwedge_{a \in A} \bigwedge_{i=1}^m \mathcal{M}_a (\lambda_i \vee C_i)(x), \quad \text{by Lemma 5.8(2)} \\ &= \bigwedge_{i=1}^m \bigwedge_{a \in A} \mathcal{M}_a (\lambda_i \vee C_i)(x) \\ &= \bigwedge_{i=1}^m \bigwedge_{a \in A} (\lambda_i \vee \mathcal{M}_a(C_i))(x) \\ &\leq_{\mathcal{L}} \bigwedge_{i=1}^m (\lambda_i \vee \mathcal{M}_{a_j}(C_i))(x), \quad \text{where } C_i(a_j) = 0 \\ &= \bigwedge_{i=1}^m (\lambda_i \vee C_i)(x) \\ &= f(x). \end{aligned}$$

Note that we have used the fact if $C(a) = 0$ then $\mathcal{M}_a(C) = C$ which follows from [Bs93]. ∎

The algorithm $\mathcal{L}$-Λ learns $\mathcal{L}$-functions f with $\mathcal{LM}$-basis $A = \{a_1, \ldots, a_t\}$. This algorithm simulates the running of t copies of LEARN-$\mathcal{L}$-MDNF, one for each $f(x + a_i)$. In step 2 in the algorithm we ask equivalence query with $\bigwedge_{i=1}^t h_i$, where h_i is the hypothesis of the i-th algorithm. Since $h_i \leq_{\mathcal{L}} \mathcal{M}_{a_i}(f)$, we have

$$\bigwedge_{i=1}^t h_i \leq_{\mathcal{L}} \bigwedge_{i=1}^t \mathcal{M}_{a_i}(f) = f.$$

Therefore the counterexample v always satisfies $\bigwedge_{i=1}^t h_i(v) \not\geq_{\mathcal{L}} f(v)$. This implies that there exists at least one h_i that satisfies $h_i(v) \not\geq_{\mathcal{L}} f(v)$. This shows that after each equivalence query the counterexample will be a superset query counterexample for at least one of the algorithms. This implies the following lemma.

Algorithm : $\mathcal{L}$-Λ
/* $A = \{a_1, a_2, \ldots, a_t\}$ is an $\mathcal{M}$-basis for C and $f \in C$. */

(1) $h_i \leftarrow 0$ for all $i \in \{1, 2, \ldots, t\}$;
(2) $v \leftarrow EQ(\bigwedge_{i=1}^{t} h_i)$; (3) $I = \{i : h_i(v) \not\geq_{\mathcal{L}} f(v)\}$;
(4) For each $i \in I$ do
    (a) $v_i \leftarrow v$;
    (b) Walk v_i down while $\exists w \in \lfloor v_i \rfloor_{a_i}$ so that $h_i(w) \not\geq_{\mathcal{L}} f(w)$;
    (c) $h_i \leftarrow h_i \vee f(v_i) T^{v_i}(x + a_i)$;
(5) Go to step 2;

Figure 4: An algorithm to learn $\mathcal{L}$-functions when the monotone basis is known.

Lemma 5.15 *Let C be a class with $\mathcal{LM}$-basis A. Then $\mathcal{L}$-Λ exactly learns any $f \in C$ using $s|A|$ equivalence queries and $s|A|n^2$ membership queries, where s is the $\mathcal{L}$-DNFsize of f.*

Next we extend the CDNF algorithm in [Bs93]. Recall that this algorithm simultaneously searches for a basis while learning the target. A CDNF is a boolean function whose CNF size is polynomial in its DNF size. The term $\mathcal{L}$-CDNF has the obvious similar definition.

Theorem 5.1 *$\mathcal{L}$-CDNF exactly learns any $f \in C$ using s equivalence queries and sn^2 membership queries where s is the product of the $\mathcal{L}$-DNF and the $\mathcal{L}$-CNF sizes of f. In particular, decision trees of size s are learnable from s^2 equivalence queries and s^2n^2 membership queries.*

The above theorem will follow from the following lemma which states that each counterexample that satisfies $H(a) \geq_{\mathcal{L}} f(a)$ yields a new *basis* point of C.

Lemma 5.16 *Let $f = \bigwedge_{i=1}^{s}(\lambda_i \vee C_i)$ be a minimal size $\mathcal{L}$-CNF of f. Also let $A = \{a_1, a_2, \ldots, a_t\} \subset \{0,1\}^n$ that satisfies the following property for some fixed $r < s$.*

For every C_i, $i \leq r$, there exists $a_{j(i)} \in A$ such that $a_{j(i)}$ falsifies C_i.

If h_i, $i \in \{1, 2, \ldots, t\}$, are $\mathcal{L}$-functions which satisfy $h_i \leq \mathcal{M}_{a_i}(f)$, then any assignment $a \in \{0,1\}^n$ that satisfies $f(a) < \bigwedge_{i=1}^{t} h_i(a)$, must falsify some C_j, $j > r$.

Proof Suppose that $C_i(a) = 0$ for all $i > r$. Then

$$f(a) = \bigwedge_{i=1}^{r}(\lambda_i \vee C_i(a)).$$

Now for every i we have

$$\begin{aligned} h_{j_i}(a) &\leq_{\mathcal{L}} \mathcal{M}_{a_{j_i}}(f)(a) \\ &\leq_{\mathcal{L}} \mathcal{M}_{a_{j_i}}(\lambda_i \vee C_i)(a) \\ &= \lambda_i \vee C_i(a). \end{aligned}$$

Algorithm : $\mathcal{L}$-CDNF

(1) $t \leftarrow 0$;
(2) $v \leftarrow EQ(\top)$; If the answer is Yes then stop;
(3) $t \leftarrow t+1$, $H_t \leftarrow \perp$, $a_t \leftarrow v$;
(4) $v \leftarrow EQ(\bigwedge_{i=1}^{t} h_i)$; If the answer is Yes then stop;
(5) $I = \{i : h_i(v) \not\geq_{\mathcal{L}} f(v)\}$;
(6) If $I = \emptyset$ then go to step 3;
(7) For each $i \in I$ do
    (a) $v_i \leftarrow v$;
    (b) Walk v_i down while $\exists w \in \lfloor v_i \rfloor_{a_i}$ so that $h_i(w) \not\geq_{\mathcal{L}} f(w)$;
    (c) $h_i \leftarrow h_i \vee f(v_i) T^{v_i}(x + a_i)$;
(8) Go to step 4;

Figure 5: An algorithm that learns $\mathcal{L}$-functions without a monotone basis as input.

Therefore

$$\begin{aligned} f(a) &<_{\mathcal{L}} \bigwedge_{i=1}^{t} h_i(a) \\ &\leq_{\mathcal{L}} \bigwedge_{i=1}^{r} h_{j_i}(a) \\ &\leq_{\mathcal{L}} \bigwedge_{i=1}^{r} (\lambda_i \vee C_i(a)) \end{aligned}$$

which implies

$$f(a) <_{\mathcal{L}} \bigwedge_{i=1}^{r} (\lambda_i \vee C_i(a)),$$

a contradiction. ∎

6 Open Problems

We offer some open problems that arise from this paper. In order of increasing difficulties, we ask if:

1. Is it possible to learn $DT(X, Y, MDNF)$ when the sets X and Y are not distinguishable?

2. Is it possible to learn any $DT(X, Y, C)$ when the sets X and Y are not distinguishable?

3. Are there any other structures S, other than decision trees, that *preserves* learnability, i.e., $S(X, Y, C)$ is learnable whenever C is a learnable class over Y?

References

[A88] Dana Angluin. Queries and Concept Learning. *Machine Learning*, 2(4):319–342, 1988.

[AHK93] Dana Angluin, Lisa Hellerstein and Marek Karpinski. Learning Read-Once Formulas with Queries. *Journal of ACM*, 40(1):185-210, 1993

[Bl92] Avrim Blum Rank-r Decision Trees are a Subclass of r-Decision Lists. *Information Processing Letters*, 43, pages 183-185, 1992.

[Bs93] Nader H. Bshouty. Exact Learning via the Monotone Theory. In *Proceeding of the 34th Symposium on Foundations of Computer Science.* pages 302–311, November 1993.

[DP] B.A. Davey and H.A. Priestley. *Introduction to Lattices and Order. Cambridge University Press*, 1990.

[EH89] Andrzejz Ehrenfeucht and David Haussler. Learning Decision Trees from Random Examples. *Information and Computation*, 82, pages 231-246, 1989.

[H92a] Thomas R. Hancock. Learning 2μDNF Formulas and $k\mu$ Decision Trees. In *Proceedings of the Fourth Annual Workshop on Computational Learning Theory,* August 1991.

[H92b] Thomas R. Hancock. The Complexity of Learning Formulas and Decision Trees that have Restricted Reads. TR-15-92, Harvard University. (Thesis)

[H93] Thomas R. Hancock. Learning $k\mu$ Decision Trees on the Uniform Distribution. In *Proceedings of the Sixth Annual Workshop on Computational Learning Theory,* pages 352-360 July, 1993.

[KM93] Eyal Kushilevitz and Yishay Mansour. Learning Decision Trees using the Fourier Spectrum. *SIAM J. Computing*, 22(6):1331–1348, 1993.

[L88] Nick Littlestone. Learning Quickly when Irrelevant Attributes Abound: A New Linear-Threshold Algorithm. *Machine Learning,*2, pages 285-318, 1988.

[R87] Ronald L. Rivest Learning Decision Lists. *Machine Learning,* 2, pages 229-246, 1987.

[Val84] Leslie G. Valiant. A Theory of the Learnable. *Communications of the ACM*, 27(11):1134–1142, November 1984.