On Learning Exclusive-Or of Terms

Nader H. Bshouty*
Department of Computer Science
University of Calgary
Calgary, Alberta, Canada
bshouty@cpsc.ucalgary.ca

Eyal Kushilevitz†
Department of Computer Science
Technion Institute of Technology
Haifa, Israel
eyalk@csa.cs.technion.ac.il

Abstract

Consider the class of read-$O(\log n)$ Exclusive-Or of terms formulae (ENF). That is, formulae which are Exclusive-Or of, not necessarily monotone, terms where each variable appears in the formula at most $O(\log n)$ times. We show that this class is learnable in polynomial time from membership and equivalence queries with an output hypothesis that is a polynomial size Exclusive-Or of terms.

This, in particular, solves the problem of learning read-$O(\log n)$ disjoint DNF from membership and equivalence queries with output hypothesis that is a depth 2 formula.

1 Introduction

Schapire and Sellie [SS93] show that the class of multivariate polynomial over $GF[2]$ is learnable from membership and equivalence queries with output hypothesis that is multivariate polynomial. Such polynomials can be thought of as the Exclusive-or of monotone terms. They pose the question of whether the class of Exclusive-or of not necessarily monotone terms is also learnable from membership and equivalence

*http://www.cpsc.ucalgary.ca/∼bshouty/home.html
†http://www.cs.technion.ac.il/∼eyalk

queries.[1] Proving the learnability of this class implies the learnability of disjoint DNF (these are DNF formula where each assignment can satisfy at most one term). Aizenstein and Pitt [AP92] show that when restricting the formulae to *read-k* disjoint DNF (i.e., each variable appears at most k times in the formula), for constant k, the class becomes learnable with output hypothesis that is polynomial size DNF. This was improved by Blum et al. [BK+94] that showed the learnability of read-k disjoint DNF, for $k = O(\log n/\log\log n)$.[2] Blum et al. also asked whether this result can be extended to read-$O(\log n)$ disjoint DNF. It is also important to note that Blum et al. proved that there are read-2 disjoint DNF formulae which do not have a small CNF representation. Hence, this class is not learnable using the monotone theory of [B93]. The class of disjoint DNF was very recently shown to be learnable from membership and equivalence queries in [BCV96] in polynomial time with output hypothesis that is a polynomial size circuit.

In this paper we show that the class of read-$O(\log n)$ Exclusive-or of terms (read-$O(\log n)$-ENF) is learnable from membership and equivalence queries with output hypothesis that is Exclusive-or of terms. This, in particular, gives an exact learning algorithm for the class of read-$O(\log n)$ disjoint DNF with output hypothesis that is depth 2 circuit.

Our approach is a randomized version of the divide and conquer technique developed in [B95a] combined with the multivariate interpolation technique developed in [B95b, BM95]. We can informally describe our approach as follows: We run an algorithm that can learn formulae consisting of a single term. If this algorithm fails then a variable will be found that shows that the formula is not a single term (a "troublemaker"). We then randomly and uniformly choose one of three ways (based on the abovementioned techniques) to eliminate this troublemaker (where two out of these three ways are guaranteed to make progress). Our analysis shows that this simple algorithm runs in polynomial time for the above classes.

Our paper is organized as follows. In Section 2 we give the learning model. In Section 3 we give some definitions and preliminary results and in Section 4 we present the algorithm and prove its correctness and efficiency.

[1] The PAC-learnability of such functions, in the case where the number of terms is some constant k was considered in [BS90, FS90].

[2] In fact both the results of [AP92] and [BK+94] deal with more general formulae which are "almost disjoint".

2 The Learning Model

In what follows we define the *exact learning* model, using membership queries and equivalence queries, as defined by Angluin [A88]. In this model, there is a function f called the *target function*, which is a member of a class of function C defined over the variable set $\{x_1, \ldots, x_n\}$. The learning algorithm can ask two types of queries about the target function:

- *membership query* in which the learning algorithm supplies an assignment a to the variables in $\{x_1, \cdots, x_n\}$ as an input to a *membership oracle*, and receives in return the value of $f(a)$. (We denote such a query by $MQ(a)$.)
- *equivalence query* in which the learning algorithm supplies a function h as an input to an *equivalence oracle*, and the reply of the oracle is either "yes", meaning that h is equivalent to f, or a *counterexample*, which is an assignment b such that $h(b) \neq f(b)$. (We denote such a query by $EQ(h)$.)

The goal of the learning algorithm is to halt after polynomial time in the number of variables, n, and the size of the target formula and output a function h that is logically equivalent to f.

3 Preliminaries

In this section we give some notations that are used in the paper. Then, we recall Angluin's algorithm for learning monotone DNF.

Let $B = \{0, 1\}$ with the order $0 < 1$. For a vector $a \in B^n$, let $a[i]$ denote the ith entry of a. The order on B induces a partial order on B^n: for $a, b \in B^n$ define $a \leq b$ if $a[i] \leq b[i]$ for all i. We say that a and b are *incomparable* if $a \not< b$ and $b \not< a$. For a vector (assignment) $a \in B$ we write $\lfloor a \rfloor$ for the set of the largest vectors that are less than a. That is, $b \in \lfloor a \rfloor$ if there exists i where $b[i] = 0, a[i] = 1$ and for every $j \neq i$, $a[j] = b[j]$. For two assignments a and b the vector $c = a + b$ is defined by $c[i] = a[i] \oplus b[i]$ where $\oplus$ denotes the exclusive-or of bits. The assignment (vector) $\mathbf{1}$ satisfies $\mathbf{1}[i] = 1$ for all i. For an assignment a we define $\bar{a} = a + \mathbf{1}$, that is, the assignment such that $(\bar{a})[i] = a[i] \oplus 1$ for all i.

A *boolean function* is a function $f : B^n \to B$. We represent boolean functions by *formulae* that use the variables $x_1, \ldots, x_n$ with the operations $\vee$ (disjunction), $\wedge$ (conjunction), and $+$ (exclusive-or). A *literal* is either x_i or $\bar{x}_i = x_i + 1$. A *term*

is a conjunction of literals. A *monotone term* is a conjunction of variables. For an assignment $b \in B^n$ we define T_b the conjunction of all x_i where $b[i] = 1$. That is

$$T_b = \bigwedge_{b[i]=1} x_i.$$

If $b = 0$, the zero assignment, then $T_b = 1$. Notice that $T_b(a) = 1$ if and only if $a \geq b$. In many cases we will say "a is above b" to mean $a \geq b$. A *DNF* is a disjunction of terms. A *monotone DNF* is a disjunction of monotone terms. Every monotone DNF has a unique representation $T_{b_1} \vee \cdots \vee T_{b_r}$ where b_i and b_j are incomparable for every $1 \leq i < j \leq r$. The integer r is called the *size* of the monotone DNF. The vectors b_i are called the minterms of f. The *DNF size* of a boolean function is the minimum of the number of terms over all the DNFs (monotone and non-monotone) that are equivalent to f. A *disjoint DNF* is a DNF where the conjunction of any two terms is 0. An ENF is an exclusive-or of terms. The *ENF size* of a boolean function f, denoted $|f|$, is the minimum number of terms over all the ENFs that are equivalent to f. An ENF is *read-k* if each variable appears in at most k terms.

For a boolean function $f : B^n \to B$, and $\sigma \in B$, we define $f|_{x_i \leftarrow \sigma}$ to be the projection of the function f by assigning σ to x_i. That is, $f|_{x_i \leftarrow \sigma}$ is a function of $n-1$ variables defined by $f|_{x_i \leftarrow \sigma}(x_1, \ldots, x_{i-1}, x_{i+1}, \ldots, x_n) = f(x_1, \ldots, x_{i-1}, \sigma, x_{i+1}, \ldots, x_n)$. For two boolean functions f_1 and f_2 we write $f_1 \Rightarrow f_2$ if $f_1(a) \leq f_2(a)$ for all $a \in B^n$.

Angluin [A88] gives an algorithm for learning monotone DNF. The algorithm learns the minterms of the DNF. At any stage of the algorithm, if $\{b_1, \ldots, b_i\}$ are the minterms learned so far, the algorithm proceeds by asking the equivalence query $EQ(T_{b_1} \vee \cdots \vee T_{b_i})$. If the answer of the equivalence query is "YES" then the algorithm outputs $T_{b_1} \vee \cdots \vee T_{b_i}$. Otherwise, the equivalence query returns a counterexample a. Since at each stage of the algorithm $T_{b_1} \vee \cdots \vee T_{b_i} \Rightarrow f$ the counterexample returned by the equivalence oracle satisfies $f(a) = 1$ and $T_{b_1}(a) \vee \cdots \vee T_{b_i}(a) = 0$. Hence $T_{b_1}(a) = \cdots = T_{b_i}(a) = 0$ and therefore a is not above any of the b_i. But since $f(a) = 1$, a must be above some other (new) minterm of f. The algorithm then "walks" down in the lattice B^n (by flipping ones to zeros in a) as long as $MQ(a) = 1$. When no element $b \in \lfloor a \rfloor$ satisfies $f(b) = 1$ then T_a is a new minterm for f. We add T_a to our collection of minterms and repeat the above procedure again.

4 The Learning Algorithm

In this section we give the learning algorithm, prove its correctness and then analyze its running time.

4.1 Algorithm

We first show how to learn a function f which is a single term, i.e. $f = T$. The algorithm for learning one term asks $EQ(0)$. If the answer to this equivalence query is "YES" then the term is 0. Otherwise, the equivalence oracle returns an assignment a such that $f(a) = 1$. Since $f(a) = 1$ the function g, defined by $g(x) = f(x + \bar{a})$, is monotone. To learn the single monotone term of g, we use Angluin's algorithm. That is, we start from $b = \mathbf{1}$, for which $g(b) = f(a) = 1$ and flip bits which are ones in b until no more bits can be flipped (note that membership queries about g can be easily simulated by membership queries about f, since $g(b) = f(b+\bar{a})$). Therefore, b is a minterm, $g(x) = T_b$ and $f(x) = g(x+\bar{a})$.

Next we show how to extend the above algorithm to functions with more than one term. The algorithm starts by assuming that the target function f consists of a single term. The algorithm asks for a positive example a for f (that is, $f(a) = 1$) by asking $EQ(0)$. As in the previous algorithm we find some term $g(x) = T_b$. Now we ask another equivalence query $EQ(T_b(x+\bar{a}))$. If f is a single term then the answer of the equivalence oracle will be "YES". Otherwise, we get a counterexample, c', that distinguishes between $T_b(x+\bar{a})$ and f. Then $c = c' + \bar{a}$ distinguishes between T_b and g. Take the assignment c and flip the ones in c as long as it still satisfies $g(c) \neq T_b(c)$. The resulting assignment c satisfies $g(c) \neq T_b(c)$ and, in addition, for every $d \in \lfloor c \rfloor$ we have $g(d) = T_b(d)$. For every function g and every i, g can be expressed as

$$g = g_0\bar{x}_i + g_1 x_i + g_2,$$

where g_0, g_1 and g_2 are functions of $n-1$ variables and $|g| = |g_0| + |g_1| + |g_2|$. We now show how to use the assignment c to find an index i such that at most one of g_0, g_1 and g_2 is 0. First, since $g(\mathbf{1}) = 1$ we cannot have $g = g_0\bar{x}_i$, for any x_i. There are two cases:

Case I. $g(c) = 0$ and $T_b(c) = 1$.
Since $T_b(c) = 1$ we have $c \geq b$. Also, $c \neq b$ because $g(b) = 1$ and $g(c) = 0$. Let x_i be a variable such that $c[i] = 1$ and $b[i] = 0$. The function g depends on x_i because

flipping $c[i]$ will change the value of g from 0 to 1 (i.e., at least one of g_0, g_1 is not 0). In addition, g is not of the form $x_i g'$ because $g(b) = 1$ and $b[i] = 0$.

Case II. $g(c) = 1$ and $T_b(c) = 0$.
Since $T_b(c) = 0$ we must have $c \not> b$. Let x_i be such that $b[i] = 1$ and $c[i] = 0$. The function g depends on x_i because flipping $b[i]$ will change the value of g from 1 to 0. In addition, g is not of the form $x_i g'$ because $g(c) = 1$ and $c[i] = 0$.

Therefore (in both cases) g is of the form $g = g_0\bar{x}_i + g_1 x_i + g_2$ where at most one of g_0, g_1 and g_2 is 0 and $|g| = |g_0| + |g_1| + |g_2|$. By the definition of g, for the same i, we can write

$$f = f_0\bar{x}_i + f_1 x_i + f_2,$$

where at most one of f_0, f_1 and f_2 is 0 and $|f| = |f_0| + |f_1| + |f_2|$. Now, let i be the index found by the algorithm. Denote

$$f' \equiv f|_{x_i \leftarrow 0} \equiv f_0 + f_2,$$

$$f'' \equiv f|_{x_i \leftarrow 1} \equiv f_1 + f_2,$$

and

$$f''' \equiv f|_{x_i \leftarrow 1} + f|_{x_i \leftarrow 0} \equiv f_0 + f_1.$$

We now choose (using a strategy described below) one of the following three representations for f:

1. $f = f''x_i + f'\bar{x}_i$.
2. $f = f'''x_i + f'$.
3. $f = f'' + f'''\bar{x}_i$.

In each of the three cases we recursively learn the two functions (out of f', f'', f''') needed in this case. This is done using the divide and conquer technique [B95a]. That is, to simulate a membership query for f' (resp. f'') the algorithm uses a membership query to f and set $x_i = 0$ (resp. $x_i = 1$). To simulate a membership query for f''' the algorithm uses two membership queries to f (one with $x_i = 0$ and one with $x_i = 1$). To simulate equivalence queries the algorithm waits until both recursive calls reach an equivalence query. The algorithm then combines the two hypotheses into a single hypothesis h and ask $EQ(h)$. If the answer is YES then $f \equiv h$ and we are done. Otherwise, the counterexample for h gives a counterexample

for one of the two hypotheses. For example, if we choose representation (2) and the algorithm that tries to learn f''' wants to ask about h''' and the algorithm that tries to learn f' wants to ask about h' we define $h = h'''x_i + h'$. A counterexample to h (i.e., b such that $h(b) \neq f(b)$) gives a counterexample b^* (by omitting the ith bit of b) to one of h' or h''' (that is, either $h'(b^*) \neq f'(b^*)$ or $h'''(b^*) \neq f'''(b^*)$). We then continue with the corresponding algorithm until reaching the next equivalence query.

If f is known to be an $O(\log n)$-term ENF then the strategy for learning f is just to choose uniformly *at random* one of the above three representations. If f is a read $O(\log n)$-ENF then the strategy for learning f is to choose *arbitrarily* one of the representation (2) or (3). For learning f' and f'' run the same algorithm. For learning f''' use the strategy for learning $O(\log n)$-term ENF. Notice that $f''' = f_0 + f_1$ is $O(\log n)$-term ENF because x_i appears at most $O(\log n)$ times in f.

In fact for the read $O(\log n)$-ENF we only need to find a variable on which f depends and then use representation (2) or (3). A simpler way to find a variable on which the target depends is to ask $EQ(0)$ and $EQ(1)$ to get two assignments a and b such that $f(a) = 1$ and $f(b) = 0$. Then, to flip bits in a that disagree with b until one of the bits changes the value of the function from 1 to 0. This last bit flipped is a variable that the target depends on.

4.2 Analysis

The key idea in proving the correctness of the algorithm is to show that

1. the size of the recurrence tree of the algorithm is polynomial.
2. The operation f''' is used in the tree at most $O(\log n)$ times in each path in the tree.

Condition 1 guarantees that the number of recursive calls in the algorithm is polynomial and therefore the number of equivalence queries asked by the algorithm is polynomial. Condition 2 guarantees that we can simulate membership queries at each level in the tree using polynomial number of membership queries to f. This is because to ask one membership query for f''' we need two membership queries for f. In general we need 2^d membership queries for simulating one membership query of some function in the recurrence tree where d is the number of times we used the operator $'''$ to reach this node.

First we show that our algorithm runs in expected polynomial time and expected polynomial number of queries for a target that is an exclusive-or of $O(\log n)$ terms. Let f be an exclusive-or of at most k terms and let X_k be the maximum, over all k-term ENF functions with at most n variables, of the expected depth of the recurrence of the algorithm.

We run the above algorithm to find the variable x_i with the property that the target f is of the form $f = f_0\bar{x}_i + f_1 x_i + f_2$ where at most one from f_0, f_1 and f_2 is 0 and $|f| = |f_0| + |f_1| + |f_2|$. Since at least two of f_0, f_1, f_2 is not 0, then at least two of f', f'', f''' have less than $k-1$ terms. Therefore, when we randomly choose one of the above three representations for f, at least one of the two functions we need to learn have at most $k-1$ terms and the other function will have with probability (at least) $1/3$ at most $k-1$ terms (in which case both functions have at most $k-1$ terms) and with probability (at most) $2/3$ at most k terms. Therefore,

$$X_k \leq \frac{1}{3}X_{k-1} + \frac{2}{3}X_k + 1.$$

This implies that

$$X_k \leq 3k.$$

Therefore if $k = O(\log n)$ then the expected depth of the recurrence is $O(\log n)$. Similar recurrence shows (see remark below) that the expected size is at most $2^{O(k)} = poly(n)$.[3] At each level i of the algorithm a membership query can be simulated by at most 2^i membership queries to f and an equivalence query can be simulated by a single equivalence query to f. Therefore the running time of the algorithm for a target which is $O(\log n)$ term ENF is polynomial.

Now, when the function f is read $O(\log n)$-ENF we decompose it, say by representation (2) (the case of representation (3) is similar). In this case $f''' = f_0 + f_1$ is an $O(\log n)$ term ENF hence is learnable with the above algorithm in polynomial time. The other function that we have to learn, f', is still a read $O(\log n)$-ENF but with at most $n-1$ variables. Hence the total running time for the case of read $O(\log n)$-ENF is at most n times the running time for the case of $O(\log n)$ term ENF, which is polynomial.

Remark 1: Some small improvements on the above algorithm are possible. For example, suppose that f is a read $O(\log n)$-ENF which has m terms (think of m

[3]Note that just the fact that the expected depth is d does *not* imply that the expected size is 2^d.

as much larger than $\log n$ and much smaller than n). We could let the algorithm choose randomly between representations (2) and (3). Since at least one of f_0 and f_1 is not 0 then at least one of f' and f'' has at most $m-1$ terms. Therefore, with probability at least 1/2 we reduce the number of terms by 1. Therefore, on the average we have a running time which is at most $2m$ times the running time in the case of $O(\log n)$ term ENF. (On the other hand, if we use this improvement then we cannot use the simplified version of finding the variable x_i on which f depends, described at the end of Section 4.1.)

Remark 2: We can also analyze the size of the tree in the case of k-term ENF more carefully: Notice that since

$$|f'| + |f''| + |f'''| \leq 2|f| = 2k$$

we must have that at least one of f', f'' and f''' has size less than $2k/3$ and the other two functions are of sizes at most $k-1$ and k. Therefore, if the expectation of the maximal possible size (over all k-term ENF formulae) is $W(k)$ (where for size we count the number of leaves of the tree), then we have

$$W(k) \leq \frac{1}{3}(W(2k/3) + W(k)) + \frac{1}{3}(W(2k/3) + W(k-1)) + \frac{1}{3}(W(k) + W(k-1)).$$

This implies

$$W(k) \leq 2W(k-1) + 2W(2k/3).$$

It follows, by induction on k, that $W(k) \leq k2^k$.

References

[AP91] H. Aizenstein and L. Pitt. Exact learning of read-twice DNF formulas, "Proc. 32th Annu. IEEE Sympos. Found. Comput. Sci.". IEEE Computer Society Press, 1991, pages 170–179.

[AP92] H. Aizenstein and L. Pitt. Exact Learning of Read-k Disjoint DNF and Not-So-Disjoint DNF, "Proc. of 5th Annu. ACM Workshop on Comput. Learning Theory". 1992, pages 71-76.

[A88] D. Angluin. Queries and Concept Learning. Machine Learning, pages 319–342, 2, 4, 1988.

[BK+94] A. Blum, R. Khardon, E. Kushilevitz, L. Pitt and D. Roth. On learning read-k-satisfy-j DNF, "Proc. of 7th Annu. ACM Workshop on Comput. Learning Theory". 1994, pages 110–117.

[BS90] A. Blum, and M. Singh, Learning Functions of k Terms, "Proc. of 3rd Annu. ACM Workshop on Comput. Learning Theory". 1990, pages 144-153.

[B93] N. Bshouty, Exact Learning via the Monotone Theory, "Proc. 34th Annu. IEEE Sympos. Found. Comput. Sci.". IEEE Computer Society Press, 1993, pages 302–311.

[B95a] N. Bshouty, Simple Learning Algorithms Using Divide and Conquer, "Proc. of 8th Annu. ACM Workshop on Comput. Learning Theory". 1995, pages 447–453.

[B95b] N. Bshouty, A Note on Learning Multivariate Polynomials under the Uniform Distribution, "Proc. of 8th Annu. ACM Workshop on Comput. Learning Theory". 1995, pages 79–82.

[BCV96] F. Bergadano, D. Catalano and S. Varricchio. Learning Sat-k-DNF Formulas from Membership Queries. manuscript.

[BM95] N. Bshouty, and Y. Mansour, Simple Learning Algorithms for Decision Trees and Multivariate Polynomials, "Proc. 36th Annu. IEEE Sympos. Found. Comput. Sci.". IEEE Computer Society Press, 1995, pages 304–311.

[FS90] P. Fischer, and H. U. Simon, On Learning Ring-Sum-Expansions, "Proc. of 3rd Annu. ACM Workshop on Comput. Learning Theory". 1990, pages 130-142.

[SS93] R. E. Schapire and L. M. Sellie. Learning Sparse Multivariate Polynomials over a Field with Queries and Counterexamples. "Proc. of 6th Annu. ACM Workshop on Comput. Learning Theory". 1993, pages 17–26.