Fast Learning of k-Term DNF Formulas with Queries

Avrim Blum*
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Steven Rudich
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

November 7, 1991

Abstract

This paper presents an algorithm that uses equivalence and membership queries to learn the class of k-term DNF formulas in time $O(n \cdot k^{O(k)})$, where n is the number of input variables. This algorithm allows one to learn DNF of $O(\frac{\log n}{\log \log n})$ terms in polynomial time, and is the first polynomial-time procedure known to learn general DNF with more than a constant number of terms in a worst-case model. The algorithm presented is randomized, and the time bounds are expected time bounds with the expectation taken over the coin tosses of the procedure. The algorithm is an *exact learning* algorithm, but one where the equivalence query hypotheses and the final output are general (not necessarily k-term) DNF formulas.

For the special case of 2-term DNF formulas, we give a deterministic version of our algorithm that uses at most $4n + 2$ total membership and equivalence queries.

1 Introduction

The question of the learnability of disjunctive-normal-form (DNF) formulas under standard models is one of the main open problems in machine learning theory. DNF formulas having a constant number of terms, along with those where each term has size at most a constant, are basic subclasses known to be learnable in polynomial-time in the (extended) equivalence-query or PAC prediction models. The standard strategy for learning k-term DNF formulas uses a hypothesis class of k-CNF formulas (CNF with each clause of size at most k) and runs in time $O(n^k)$, where n is the number of input variables [8]. It thus can learn k-term DNF formulas only for constant values of k. A different technique for learning k-term DNF using a representation of general DNF is known as well [5], but also requires time $O(n^k)$. With the added ability of the learner to make membership queries—that is, to ask for classification of examples of its own choosing—a wider collection of DNF subclasses are known learnable, such as monotone DNF [9] and read-twice DNF (each variable appears at most twice) [1][6]. However, the only previous advance known for learning k-term DNF formulas with queries is an algorithm of Angluin [2] whose running time is worse, $O(n^{k^2})$, but actually finds the original k-term DNF.

This paper presents an algorithm to learn k-term DNF formulas that uses equivalence and membership queries and runs in an expected time, over the coin tosses of the algorithm, of $O(n \cdot k^{O(k)})$. Thus for constant values of k the running time is *linear* in n (rather than n^k), and more generally the algorithm can learn k-term DNF in polynomial time for k as large as $O(\frac{\log n}{\log \log n})$. This is the first polynomial-time procedure known to learn general DNF of more than a constant

*Supported by an NSF Postdoctoral Fellowship. Email: avrim@theory.cs.cmu.edu

number of terms in a standard model. The algorithm is an *exact learning* algorithm, but uses hypotheses that are general (not necessarily k-term) DNF. For the special case of 2-term DNF, a simpler, deterministic version of the algorithm is given that uses at most $4n + 2$ equivalence and membership queries, and where the equivalence query hypotheses are in fact 2-term DNF formulas.

Angluin and Kharitonov [4] have shown that membership queries do not help in learning DNF formulas containing an unbounded number of terms. They prove this by demonstrating a general class of DNF formulas where the learner cannot produce a negative example it has not yet seen, even if it knows the target concept. While the inability of the learner to produce "good" queries holds even if all terms of the DNF are small (size at most 3), the arguments break down when the *number* of terms is restricted. In particular, they require the number of terms to be polynomially related to the number of variables. We show that when the number of terms is $O(\frac{\log n}{\log\log n})$, then membership queries *can* be usefully employed. The power of membership queries in the intermediate range, say for DNF of polylog(n) terms, remains an open problem.

1.1 Definitions

Let V be a set of n variables $X_1, \ldots, X_n$ and define a *literal* to be either a variable or a negation of a variable. An example $\vec{x}$ is a boolean vector over $\{0,1\}^n$, and we write $\vec{x}[i]$ to denote the ith bit of $\vec{x}$. We will also think of examples as boolean functions over the literals with $\vec{x}(X_i) = \vec{x}[i]$ and $\vec{x}(\overline{X}_i) = 1 - \vec{x}[i]$. A *term* is a conjunction of literals—that is, an example satisfies a term T if and only if it satisfies all literals in T.[1] A k-term DNF formula is a disjunction of at most k terms. An example $\vec{x}$ is a *positive example* of a function F if $\vec{x}$ satisfies F ($F(\vec{x}) = 1$) and a *negative example* otherwise ($F(\vec{x}) = 0$).

We say two terms are *disjoint* if there is no literal l such that either l appears in both or l appears in one and $\overline{l}$ in the other (if $l = \overline{X}_i$, then define $\overline{l}$ to be X_i). For a term T, let *lits*(T) be the set of literals in T, and for a set of literals S, let *term*(S) be the conjunction of all elements of S. For convenience, we will often identify T with *lits*(T). Also, let $V(T)$ be the set of variables X_i such that $X_i \in T$ or $\overline{X}_i \in T$. Finally, if F_1, F_2 are two boolean functions, we will write "F_1 implies F_2" or "$F_1 \Rightarrow F_2$" to mean that if $F_1(x) = 1$ then $F_2(x) = 1$.

We will often want to look at examples obtained by changing the value of a given example on some specified set of variables. Towards this end, let *flip*($\vec{x}, i$) be the example $\vec{y}$ where $\vec{y}[i] = 1 - \vec{x}[i]$ and $\vec{y}[j] = \vec{x}[j]$ for all $j \neq i$. More generally, for I a set of indices, let *flip*($\vec{x}, I$) be the example $\vec{y}$ such that $\vec{y}[i] = 1 - \vec{x}[i]$ if $i \in I$ and $\vec{y}[i] = \vec{x}[i]$ otherwise. For convenience, we also extend *flip* to act on literals or sets of literals: if $l \in \{X_i, \overline{X}_i\}$, let *flip*($\vec{x}, l$) = *flip*($\vec{x}, i$), and similarly define *flip*($\vec{x}, S$) for a set of literals S.

We assume throughout this paper that the target function F is *reduced* in that there is no term T_i in F that can be removed without changing the function, and there is no literal that can be removed from any T_i in F without changing the function.

The learning model we use is the standard extended-equivalence query and membership query model. In this model, a learner may either propose a polynomial-time hypothesis H (an *equivalence query*) and receive a counterexample $\vec{x}$ such that $H(\vec{x}) \neq F(\vec{x})$ if one exists, or else propose an example (a *membership query*) and receive the value of F on that example. The learner succeeds when it makes an equivalence query hypothesis logically equivalent to F. This model is the same as the "mistake-bound with membership queries" model, in which equivalence queries are replaced by an adversarial source of examples. A learner here may select either to make a membership query or else to predict the classification of an example of the adversary's choosing, and the learner is charged for each membership query and each mistake in prediction [7]. Learnability in these

[1] An empty term is considered to be identically true.

models implies learnability in the "PAC prediction with membership queries" model in which an adversarial source is replaced by an adversarial distribution. We note that the equivalence query model sometimes connotes a need for hypotheses to be of the same form as the target concept, i.e. k-term DNF formulas in our case, but our equivalence queries will be more general DNF formulas.

1.2 High-Level Structure of the Algorithm

Say the target concept F is the DNF formula $T_1 + T_2 + \ldots + T_k$. The algorithm we present is divided into two main parts. The first part, given a positive example $\vec{x}$, uses $O(n)$ queries and with probability $1/2^{O(k)}$ produces a positive example that satisfies only one term T_i of F — in fact, one of the terms satisfied by $\vec{x}$. The second part, given an example that satisfies exactly one term T_i of F, uses $O(n \cdot 2^{O(k)})$ queries and produces $O(k^{O(k)})$ terms such that with probability at least $1/2$, one of them is the term T_i. The probabilities here are taken over the coin tosses of the algorithm.

So, given a positive example $\vec{x}$, after $2^{O(k)}$ iterations of the combined procedure ($O(n \cdot 2^{O(k)})$ membership queries) we have with constant probability at most $O(k^{O(k)})$ terms at least one of which is a term in F satisfied by $\vec{x}$. With an extra multiplicative factor of $O(\log(k/\delta))$, we can improve our probability of success to $1 - \delta/k$. Now we add all the terms found into a hypothesis DNF and ask for an equivalence query. A negative counterexample allows us to throw out "bad" terms from the hypothesis: terms that do not imply F. A positive counterexample must satisfy some $T_i \in F$, and can only satisfy terms $T_i \in F$ we have not yet found. Thus, when we run again through the above procedure, we place a new term of F into our hypothesis with high probability. So, this procedure uses an expected $O(k^{O(k)})$ equivalence queries, mostly as negative examples to remove the "bad" terms, and $O(n \cdot 2^{O(k)})$ membership queries. The expected time bound is $O(n \cdot k^{O(k)})$.

For very small values of k, the procedure can be made simpler. In fact for $k = 2$, we give a deterministic version that requires at most $4n + 2$ total membership and equivalence queries.

2 Preliminaries

The following is a simple fact about DNF formulas.

Fact 1 *If F is k-term DNF formula that is not identically* true*, then there exists a set of $r \leq k$ literals such that any example satisfying those literals is negative for F.*

Proof: Since F is a k-term DNF, its negation $\neg F$ is a k-term CNF. Thus, $\neg F$ can be written as a k-DNF by multiplying out the CNF formula. Formula $\neg F$ must contain at least one term since $F \neq$ true, and that term provides the set of at most k literals desired. (Any example $\vec{x}$ satisfying that term will satisfy $\neg F$, so $F(\vec{x})$ will be 0). ■

The following theorem is an extension of this fact.

Theorem 1 *If T is a term that does* not *imply the k-term DNF F, then there exists a set of of $r \leq k$ literals disjoint from T such that any example satisfying both T and that set of literals must be negative for F.*

Proof: Let $\hat{F}$ be the DNF formula F where variables $X_i \in T$ are replaced by 1, and variables X_i such that $\overline{X}_i \in T$ are replaced by 0. That is, $\hat{F}$ is defined only on variables disjoint from T and is a "projection" of F fixing T onto those variables. We know $\hat{F}$ is not identically true since T does not imply F. Thus, by Fact 1, there are $r \leq k$ literals disjoint from T such that any example over

$V - V(T)$ satisfying those literals is negative for $\hat{F}$. Thus, by definition of $\hat{F}$, any example over V satisfying both T and that set of literals is negative for F. ■

Theorem 1 implies that one can use random examples satisfying a term T to test whether that term implies F. Specifically, we get the following corollary.

Corollary 2 *If T is a term that does not imply a k-term DNF formula F, then a random example satisfying T has a probability at least $1/2^k$ of being a negative example for F.*

In the other direction, we have the following simple lemma.

Lemma 3 *If $\vec{x}$ is a negative example for a k-term DNF formula, then there exist at most k indices i such that $flip(\vec{x}, i)$ is positive.*

Proof: Let $\vec{x}$ be a negative example and suppose T is a term in the DNF formula and i is an index such that $flip(\vec{x}, i)$ satisfies T. Since $\vec{x}$ does not satisfy T and $flip(\vec{x}, i)$ does, it must be that $\vec{x}$ fails to satisfy exactly one literal $l \in \{X_i, \overline{X}_i\}$ in T. So, there can be no other index $j \neq i$ such that $flip(\vec{x}, j)$ satisfies T as well. Since there are only k terms in the DNF, this implies there can be at most k indices i such that $flip(\vec{x}, i)$ is positive. ■

3 2-Term DNF

Before describing the general algorithm for k-term DNF, we describe first a simpler and deterministic version that learns 2-term DNF formulas using $O(n)$ equivalence and membership queries. In fact, the equivalence queries used by this procedure can be constructed to all be 2-term DNF formulas, and thus fall into the framework of Angluin's original, more restrictive model [3].

Let T_1 and T_2 be the two terms that make up the target concept F. One simple fact to note is that if at some point we know one of the terms, say T_1, then with only n additional equivalence queries, we can learn F. The idea is just to use the standard "list-and-cross-off" strategy, keeping the hypothesis more specific than F so we never receive negative counterexamples. Specifically, let procedure **list-and-cross-off** be the following: hypothesize $H = T_1 + T_2'$ where $T_2' = X_1\overline{X}_1X_2\overline{X}_2 \cdots X_n\overline{X}_n$, and on each positive counterexample, remove from T_2 those literals not satisfied. The first counterexample will remove n literals from T_2' and each subsequent one will remove at least one literal. So, we need only describe a procedure to learn one of the two terms.

Let $\vec{x}$ be a positive example (say the result of an equivalence query to an identically false hypothesis). Let $\hat{T}$ be the conjunction of all literals l satisfied by $\vec{x}$ such that $flip(\vec{x}, l)$ is negative. We can find $\hat{T}$ using n membership queries, and note that each literal in $\hat{T}$ must be contained in every term satisfied by $\vec{x}$. We now consider three cases for example $\vec{x}$ and provide an algorithm for each one. We do not know apriori to which case $\vec{x}$ belongs, but in the end we can just try the cases sequentially, at most increasing the time by a factor of 3.

The first case [Case A] is that $\vec{x}$ satisfies exactly one term of F, say T_1, and there is no literal l in T_1 such that $flip(\vec{x}, l)$ satisfies T_2. In this case, $T_1 = \hat{T}$ since each $flip(\vec{x}, l)$ for $l \in T_1$ is negative, so we are done.

The second case [Case B] is that $\vec{x}$ satisfies only one term T_1 of F, but that there *does* exist a literal $l_i \in T_1$ such that $flip(\vec{x}, l_i)$ satisfies T_2. Note that this implies $\overline{l_i}$ is a literal of T_2 and that this is the only literal of T_2 not satisfied by $\vec{x}$. So, there can be only one such literal l_i and we have $T_1 = \hat{T}l_i$. Our goal now is to determine, or at least limit the possibilities for, literal l_i. First, one

last fact to note is that T_2 must contain some literal l_j not in T_1, since otherwise literal l_i could be removed from T_1 without changing the function F. That is, any example that satisfied all of term T_1 with l_i removed would necessarily satisfy either T_1 or T_2.

Let W be the set of literals satisfied by $\vec{x}$ but not in $\hat{T}$, and let $\vec{y} = \mathit{flip}(\vec{x}, W)$. Example $\vec{y}$ is negative since it doesn't satisfy either l_i or l_j. So, if we let S be the set of literals satisfied by $\vec{x}$ such that $\mathit{flip}(\vec{y}, l)$ is positive (using n membership queries here), then S has size at most 2 by Lemma 3. Also, we know that $l_i \in S$ since $\mathit{flip}(\vec{y}, l_i)$ satisfies T_1. So, if $S = \{l, l'\}$, we know that either $T_1 = \hat{T}l$ or $T_1 = \hat{T}l'$, and we can just try the list-and-cross-off procedure on both hypotheses sequentially.

The final case [Case C] is that $\vec{x}$ satisfies both T_1 and T_2. So, $F = \hat{T}(T_1' + T_2')$ where T_1' and T_2' are disjoint and both nonempty. Now, we just "walk" down $\vec{x}$, flipping variables not in $\hat{T}$ until we get a negative example. More formally, for each $i = 1, 2, \ldots, n$ do the following. If neither X_i nor $\overline{X}_i$ is in $\hat{T}$, then query $\vec{y} = \mathit{flip}(\vec{x}, i)$: if $\vec{y}$ is positive, let $\vec{x} := \vec{y}$ and continue on the new $\vec{x}$, and if $\vec{y}$ is negative, break out of the loop and halt. At some point, the above procedure will halt with $\vec{y}$ negative since T_1' and T_2' are nonempty. The positive example $\vec{x}$ produced will be one of Case A since the last variable flipped was only in one of the two terms (using that T_1' and T_2' are disjoint).

A slightly more efficient way of combining the above three cases that uses at most $4n + 2$ total equivalence and membership queries is described as algorithm **Learn-2-term-DNF** below.

Algorithm Learn-2-term-DNF:

1. *Get an initial positive example $\vec{x}$.* **[1 equivalence query]**
   (Can use equivalence query of $H = X_1\overline{X}_1 + X_1\overline{X}_1$.)
2. *Query example $\mathit{flip}(\vec{x}, i)$ for each $i \in \{1, \ldots, n\}$.* **[n membership queries]**
   Let $\hat{T}$ be the conjunction of all literals $l_i \in \{X_i, \overline{X}_i\}$ satisfied by $\vec{x}$ such that $\mathit{flip}(\vec{x}, i)$ is negative.
3. *Query example $\vec{y} = \mathit{flip}(\vec{x}, V - V(\hat{T}))$; this is the example obtained by taking $\vec{x}$ and flipping the assignment of $\vec{x}$ to all variables not in $\hat{T}$.* **[1 membership query]**
4. *Query example $\mathit{flip}(\vec{y}, i)$ for each $i \in \{1, \ldots, n\}$, and let $S = \{l_i \in \{X_i, \overline{X}_i\} \mid \vec{x}$ satisfies l_i and $\mathit{flip}(\vec{y}, i)$ is positive$\}$.* **[n membership queries]**

   - If $\vec{y}$ is positive [Case A]: *We know $\hat{T}$ is one of the two terms of F, so run procedure* **list-and-cross-off** *to find the other.* **[n equivalence queries]**

     Total: $3n + 2$ queries.
   - If $\vec{y}$ is negative and S is empty [Case C]: *Let $\vec{x}_0 = \vec{x}$. For each literal $l \notin \hat{T}$ satisfied by $\vec{x}_0$, query $\vec{z} = \mathit{flip}(\vec{x}, l)$: if $\vec{z}$ is positive then let $\vec{x} := \vec{z}$ (so the next iteration will use the new example $\vec{x}$) and if $\vec{z}$ is negative then put l into $\hat{T}$. Now, $\hat{T}$ is one of the two terms of F so run procedure* **list-and-cross-off** *to find the other.*

     [n memb. and n equiv. queries]

     Total: $4n + 2$ queries.
   - If $\vec{y}$ is negative and S is non-empty [Case B or C]: *Run procedure* **list-and-cross-off** *on term $\hat{T}l$ for each $l \in S$. We know one such term is a term of F and that S has size at most 2, so this will succeed.* **[$2n$ equivalence queries]**

     Total: $4n + 2$ queries.

4 k-Term DNF

We now describe an algorithm for learning k-term DNF formulas for $k \geq 2$. As mentioned in the introduction, this algorithm consists of two main parts. We first describe how, given a positive example satisfying only one term T_i of the target concept, to produce with constant probability $O(k^{O(k)})$ terms such that one of them is T_i. We then describe how given a positive example satisfying perhaps several terms of the target concept to produce an example that with probability $1/2^{O(k)}$ satisfies only one — in fact, one of the terms satisfied by the original positive example. These two parts are then combined as described in Section 1.2.

4.1 Using an example that satisfies exactly one term

Suppose $\vec{x}$ is an example that satisfies exactly one term of F, say term T_1. In this section we describe a procedure to use $\vec{x}$ to produce with probability at least 1/2 a "relatively small" number of terms, $k^{O(k)}$, with the property that one of those terms is T_1. This will be done using $O(n \cdot 2^{O(k)})$ membership queries, and in time $O(n \cdot k^{O(k)})$. The procedure may be repeated $\log(1/\delta)$ times for a failure probability δ.

As in the case of 2-term DNF, we begin by querying each neighbor of example $\vec{x}$ to see which variables appear to matter. Using n membership queries, we let $\hat{T}$ be the conjunction of all literals l satisfied by $\vec{x}$ such that $\mathit{flip}(\vec{x}, l)$ is a negative example. Notice that $\mathit{lits}(\hat{T}) \subseteq \mathit{lits}(T_1)$.

In the case of 2-term DNF formulas, there was at most one literal in T_1 not found in $\hat{T}$. More generally for a k-term DNF formula, there are at most $k-1$ such literals. This fact is a corollary to Lemma 3 and is given as Corollary 4 below.

Corollary 4 *Suppose $\vec{x}$ is an example satisfying exactly one term T_i of F. Then, there exist at most $k-1$ literals $l \in \mathit{lits}(T_i)$ such that $\mathit{flip}(\vec{x}, l)$ is a positive example.*

Proof: Consider the DNF formula F' produced by removing T_i from F; so, $\vec{x}$ is a negative example for F'. Since F' is a $(k-1)$-term DNF formula and $\vec{x}$ is negative for F', by Lemma 3 there are at most $k-1$ literals l satisfied by $\vec{x}$ such that $\mathit{flip}(\vec{x}, l)$ satisfies F'. But, for $l \in T_i$ we have $F(\mathit{flip}(\vec{x}, l)) = F'(\mathit{flip}(\vec{x}, l))$, so this implies that there are at most $k-1$ literals $l \in \mathit{lits}(T_i)$ such that $\mathit{flip}(\vec{x}, l)$ is a positive example for F. ■

Now, given $\hat{T}$, our goal is to build up a not-too-large set S containing the missing literals: the literals in T_1 not in $\hat{T}$. Once we get the set S, we will then try all possible subsets S of size at most $k-1$, appending each to $\hat{T}$ to form the output terms of this section.

The algorithm to create S uses the following sub-procedure, which we separate out because it will be used again in Section 4.2. Note that in this procedure, the set I is simply the set of indices i such that either X_i or $\overline{X}_i$ is in $\hat{T}$, and therefore can be created with no additional membership queries.

Procedure Random-flip($\vec{x}$):

1. *Let I be the set of all indices i such that $\mathit{flip}(\vec{x}, i)$ is a negative example. (If this is not known beforehand, then it can be obtained using n membership queries).*
2. *Output example $\vec{y}$ produced using the rule: If $i \in I$, then let $\vec{y}[i] = \vec{x}[i]$. If $i \notin I$, then let $\vec{y}[i] = 0$ or 1 randomly.*

The procedure to create S is now as follows.

Procedure Create-S($\vec{x}$):

Begin with $S = \phi$, and repeat the following two steps for $2^{2k-2}\ln(4k)$ iterations.

1. *Let $\vec{y}$ =* **Random-flip**$(\vec{x})$*, and query the example $\vec{y}$.*
2. *If $\vec{y}$ is negative, then for each literal l satisfied by $\vec{x}$ but not satisfied by $\vec{y}$, query the example flip$(\vec{y},l)$; if this is positive then put l into set S. Note that this step never queries examples flip$(\vec{y},l)$ for which $l \in \hat{T}$.*

   Also, note that querying $\vec{y}$ and its neighbors uses at most $n+1$ membership queries.

We now show that with high probability the final set S created by the above procedure will both contain all the "missing literals" and not be too large. Thus, appending to $\hat{T}$ each subset of S of size at most $k-1$ will produce not too many terms, at least one of which is T_1.

Theorem 5 *With probability at least $3/4$, procedure* **Create-S**$(\vec{x})$ *produces a set S containing each literal l in T_1 missing from $\hat{T}$.*

Proof: Let $l_1, l_2, \ldots, l_r$ for some $r \leq k-1$ be those literals in T_1 not appearing in $\hat{T}$ and let T_1' be the term T_1 with literal l_1 removed. The probability that a given example $\vec{y} =$ **Random-flip**$(\vec{x})$ satisfies each of $l_2, l_3, \ldots, l_r$ is at least $1/2^{k-2}$. Given that $\vec{y}$ satisfies those literals, $\vec{y}$ is simply a random example satisfying T_1'. (That is, the distribution of examples $\vec{y}$ restricted to those which satisfy the literals listed is the same as the uniform distribution over examples satisfying T_1'.) Since T_1' does not imply function F by our assumption that F is reduced, by Corollary 2 a random example satisfying T_1' has a probability at least $1/2^k$ of being negative. Thus, with probability at least $1/2^{2k-2}$, example $\vec{y} =$ **Random-flip**$(\vec{x})$ is a negative example having the property that flip$(\vec{y},l_1)$ is positive and so l_1 is placed into S. So, over the course of Procedure **Create-S**, we have:

$$\textbf{Prob}[l_1 \notin S] \;\leq\; (1-1/2^{2k-2})^{2^{2k-2}\ln(4k)} \;\leq\; e^{-\ln(4k)} \;=\; \tfrac{1}{4k}.$$

Thus, the probability *any* of the at most $k-1$ literals in $lits(T_1) - lits(\hat{T})$ is missing from S is at most $1/4$. ■

Theorem 6 *With probability at least $3/4$, procedure* **Create-S**$(\vec{x})$ *produces a set S of size at most $4k^2$.*

Proof: Let $T_2 \neq T_1$ be some term of F, and let t be the number of literals l in T_2 such that neither l nor $\overline{l}$ is in T_1. The probability an example $\vec{y} =$ **Random-flip**$(\vec{x})$ satisfies all but one of the literals in T_2 (so that possibly one of the examples flip$(\vec{y},l)$ considered in Step 2 of Procedure **Create-S** satisfies T_2) is at *most* $t/2^t$. So, the probability over all $2^{2k-2}\ln(4k)$ iterations that any of the examples flip$(\vec{y},l)$ queried in Step 2 satisfies T_2 is at most $[2^{2k-2}\ln(4k)]t/2^t$. Since there are only $k-1$ terms in F besides T_1, the probability that *any* of the examples flip$(\vec{y},l)$ satisfies *any* term having at least $t = 2k + 3\log(4k)$ literals disjoint from T_1 is at most:

$$\frac{(k-1)[2^{2k-2}\ln(4k)][2k+3\log(4k)]}{2^{2k+3\log(4k)}} \;<\; \frac{k\ln(4k)[2k+3\log(4k)]}{4(4k)^3} \;<\; \frac{1}{4}.$$

So, with probability at least $3/4$, the positive examples flip$(\vec{y},l)$ queried in Step 2 of **Create-S** satisfy only terms of F having fewer than $2k+3\log(4k)$ literals disjoint from T_1. Thus, with probability at least $3/4$, the size of the set S is at most $(k-1)+(k-1)[2k+3\log(4k)] = 2k^2 - (k+1) + 3(k-1)\log(4k)$, which is at most $4k^2$. ■

So, the final algorithm to produce at most $k^{O(k)}$ terms, one of which is term T_1 with probability at least $1/2$, is as follows. First, use n membership queries to create $\hat{T}$ = the conjunction of all literals l satisfied by $\vec{x}$ such that $flip(\vec{x}, l)$ is a negative example. Then, run procedure **Create-S**($\vec{x}$) to get the set S of possible missing literals. Finally, if $|S| \leq 4k^2$, produce as output terms all conjunctions of $\hat{T}$ with subsets of S of size at most $k-1$. The total number of queries used is $O(n \cdot 2^{O(k)})$ and the time and number of terms produced are $O(n \cdot k^{O(k)})$.

4.2 Producing a useful positive example

Suppose we are given a positive example $\vec{x}$ that does not satisfy some hypothesis DNF H. For instance, we might receive it as a response to an equivalence query of H. We show in this section how to use $2n$ membership queries to transform $\vec{x}$ into a new positive example $\vec{z}$ that with probability $1/2^{O(k)}$ is still predicted negative by H, but now satisfies *exactly* one term of F.

The algorithm uses a procedure **Random-sweep** described below. Essentially, the example **Random-sweep**($\vec{x}$) is produced by walking down the indices of $\vec{x}$, flipping with probability $1/2$ those assignments to variables that do not make the example negative.

Procedure Random-sweep($\vec{x}$):

- *Let* $\vec{x}^{(0)} = \vec{x}$.
- *For each* $i = 1, 2, \ldots, n$: *if* $flip(\vec{x}^{(i-1)}, i)$ *is negative then let* $\vec{x}^{(i)} = \vec{x}^{(i-1)}$; *otherwise, toss a fair coin and with 50% probability, let* $\vec{x}^{(i)} = flip(\vec{x}^{(i-1)}, i)$ *and with 50% probability, let* $\vec{x}^{(i)} = \vec{x}^{(i-1)}$.
- *Output* $\vec{x}^{(n)}$.

The algorithm **Get-useful-positive** to produce the desired example $\vec{z}$ is simply:

Algorithm Get-useful-positive($\vec{x}$):

- *Let* $\vec{y}$ = **Random-sweep**($\vec{x}$).
- *Output* $\vec{z}$ = **Random-flip**($\vec{y}$) *(see Section 4.1).*

The rest of this section is devoted to showing that with probability $1/2^{O(k)}$, the example produced by the above algorithm has the properties we want.

First, for convenience, we make the following definitions. For positive example $\vec{x}$, let $\mathcal{S}(x)$ be the set of terms of the target concept satisfied by $\vec{x}$. Also, let $\mathsf{some}(\vec{x})$ be the set of literals that appear in some but not all of the terms of $\mathcal{S}(\vec{x})$. That is, if $l_i \in \{X_i, \overline{X}_i\}$ is in $\mathsf{some}(\vec{x})$, then $flip(\vec{x}, i)$ is a positive example that fails to satisfy at least one term of $\mathcal{S}(\vec{x})$. Finally, for each term *not* satisfied by $\vec{x}$, select some specific literal in l_i in that term not satisfied by $\vec{x}$ (say the one of least index) and let $L(\vec{x})$ be the set of the l_i. So, $|L(\vec{x})| < k$.

We prove correctness of the algorithm for this section through the following two theorems.

Theorem 7 *If* $\vec{x}$ *is a positive example, then with probability at least* $1/2^{2k}$ *the positive example* $\vec{y}$ = **Random-sweep**($\vec{x}$) *has* $\mathsf{some}(\vec{y}) \subseteq L(\vec{x})$ *and* $\mathcal{S}(\vec{y}) \subseteq \mathcal{S}(\vec{x})$.

Theorem 8 *If* $\vec{y}$ *is a positive example and* $|\mathsf{some}(\vec{y})| \leq k$, *then with probability at least* $1/2^{3k}$: *example* $\vec{z}$ = **Random-flip**($\vec{y}$) *satisfies exactly one term, and this term belongs to* $\mathcal{S}(\vec{y})$.

Theorems 7 and 8 prove that algorithm **Get-useful-positive** has at least a $1/2^{5k}$ probability of success since $\mathsf{some}(\vec{y}) \subseteq L(\vec{x})$ implies that $|\mathsf{some}(\vec{y})| \leq k$.

Proof of Theorem 7: Let $L = L(\vec{x})$. First, since $|L| < k$, with probability at least $1/2^k$, none of the values $\vec{x}(l)$ for $l \in L$ are flipped in producing $\vec{y} = \textbf{Random-sweep}(\vec{x})$. In the following discussion, we will assume that this in fact is the case, so $\vec{y}$ satisfies no terms not in $\mathcal{S}(\vec{x})$.

Let $l_{i_0} \in \{X_{i_0}, \overline{X}_{i_0}\}$ be the literal of least index in $\mathsf{some}(\vec{x}) - L$. When the random sweep procedure on $\vec{x}$ reaches index i_0, with probability $1/2$, the value of $\vec{x}[i_0]$ is flipped. So, some term in $\mathcal{S}(\vec{x})$ is not satisfied by $\vec{x}^{(i_0)}$ (using the examples $\vec{x}^{(i)}$ as defined in the procedure). Now, let $l_{i_1} \in \{X_{i_1}, \overline{X}_{i_1}\}$ be the least index literal in $\mathsf{some}(\vec{x}^{(i_0)}) - L$. Clearly, $i_1 > i_0$ so with independent probability $1/2$, the value $\vec{x}[i_1]$ is flipped as well and example $\vec{x}^{(i_1)}$ satisfies at least one fewer term than $\vec{x}^{(i_0)}$. We can continue this process, at each stage having a probability $1/2$ that the next $\vec{x}^{(i_j)}$ satisfies at least one fewer term, until $\mathsf{some}(\vec{x}^{(i_j)}) \subseteq L$. This must occur for some $j < k$. At this point (using our initial assumption that none of the bits of $\vec{x}$ corresponding to literals of L get flipped) the examples $\vec{x}^{(i_j+t)}$ will continue satisfying exactly the same terms for the rest of the sweep. Thus, if no $\vec{x}[i]$ for X_i or $\overline{X}_i$ in L is touched and if each $\vec{x}[i_j]$ is flipped, then $\mathsf{some}(\vec{y}) \subseteq L$. This occurs with probability at least $1/2^{2k}$. ■

Proof of Theorem 8: Let T_j be a term satisfied by $\vec{y}$, and let I be the set of indices such that $\mathit{flip}(\vec{y}, i)$ is negative (see step 1 of procedure **Random-flip**). If $l_i \in \{X_i, \overline{X}_i\}$ is a literal of T_j but $i \notin I$, then it must be that either $l_i \in \mathsf{some}(\vec{y})$ (so that $\mathit{flip}(\vec{y}, i)$ satisfies some term in $\mathcal{S}(\vec{y})$) or else $\mathit{flip}(\vec{y}, i)$ satisfies some term *not* in $\mathcal{S}(\vec{y})$. There are at most k literals $l_i \in \mathsf{some}(\vec{y})$ by assumption. Also, by Lemma 3 there are at most k indices i such that $\mathit{flip}(\vec{y}, i)$ satisfies a new term not in $\mathcal{S}(\vec{y})$ (use Lemma 3 on the DNF in which terms of $\mathcal{S}(\vec{y})$ are removed from F). So, $|I| \geq |T_j| - 2k$, which implies that in procedure **Random-flip**$(\vec{y})$ there is a probability at least $1/2^{2k}$ that none of the values $\vec{y}(l)$ for literals $l \in T_j$ are flipped. Now, we know there exists a set B of $r \leq k-1$ literals outside of T_j such that some assignment to those literals forces off all terms of F besides T_j. Specifically, let $\vec{w}$ be a positive example satisfying only term T_j in F (which must exist since F is reduced), and just pick one unsatisfied literal in each of the unsatisfied terms of F. Since the assignment to each literal in B is chosen randomly in determining $\vec{z} = \textbf{Random-flip}(\vec{y})$, with probability at least $(1/2^k) \cdot (1/2^{2k}) = 1/2^{3k}$ we have both that $\vec{z}(l) = \vec{y}(l)$ for all $l \in T_j$ (so T_j is satisfied by $\vec{z}$) and that the set B is assigned so that all other terms in F are not satisfied by $\vec{z}$.
■

Acknowledgments: We would like to thank Prasad Chalasani and Merrick Furst for a collection of helpful discussions.

References

[1] H. Aizenstein and L. Pitt. Exact learning of read-twice DNF formulas. In *Proceedings of the 32nd Annual Symposium on Foundations of Computer Science*, pages 170–179, San Juan, October 1991.

[2] D. Angluin. Learning k-term DNF formulas using queries and counterexamples. Technical Report YALEU/DCS/RR-559, Yale University Department of Computer Science, 1987.

[3] D. Angluin. Queries and concept learning. *Machine Learning*, 2(4):319–342, 1988.

[4] D. Angluin and M. Kharitonov. When won't membership queries help? In *Proceedings of the Twenty-Third Annual ACM Symposium on Theory of Computing*, pages 444–454, New Orleans, Louisiana, May 1991.

[5] A. Blum and M. Singh. Learning functions of k terms. In *Proceedings of the Third Annual Workshop on Computational Learning Theory*, pages 144–153. Morgan Kaufmann, 1990.

[6] T. Hancock. Learning 2μ DNF formulas and $k\mu$ decision trees. In *Proceedings of the Fourth Annual Workshop on Computational Learning Theory*, pages 199–209, Santa Cruz, California, August 1991. Morgan Kaufmann.

[7] N. Littlestone. Learning quickly when irrelevant attributes abound: A new linear-threshold algorithm. *Machine Learning*, 2:285–318, 1988.

[8] L. Pitt and L. G. Valiant. Computational limitations on learning from examples. *Journal of the ACM*, 35(4):965–984, 1988.

[9] Leslie G. Valiant. A theory of the learnable. *Communications of the ACM*, 27(11):1134–1142, November 1984.