Learning k-term DNF formulas using queries and counterexamples

Dana Angluin *
Yale University

August 1987

Abstract

We consider the class of propositional formulas over n variables in disjunctive normal form with at most k terms, the k-term DNF formulas. We show that for each fixed $k \geq 0$, there is an algorithm to learn any k-term DNF formula using equivalence and membership queries that runs in time bounded by a polynomial in n. Dual results hold for k-clause CNF formulas.

1 Introduction

We consider the problem of learning a propositional formula using queries to a teacher. The class of k-term DNF formulas is the class of propositional formulas over n variables in disjunctive normal form with at most k terms. Dually, the class of k-clause CNF formulas is the class of CNF formulas over n variables with at most k clauses.

Pitt and Valiant [2,3] consider the problem of learning k-term DNF formulas in the stochastic setting introduced by Valiant [4]. They show that for each $k \geq 2$ the class of k-term DNF formulas cannot be probably approximately identified in time polynomial in n unless RP = NP. RP is the class of sets recognizable in random polynomial time, and NP is the class of sets recognizable in nondeterministic polynomial time. RP is a subclass of NP, but it is unknown whether the containment is strict. Many researchers suspect that RP and NP are unequal, so this is interpreted as a negative result about learning k-term DNF formulas.

A straightforward adaptation of Pitt and Valiant's proof shows that for $k \geq 2$, the k-term DNF formulas cannot be exactly identified in time polynomial in n using equivalence queries if P is not equal to NP. (See [1] for relationships between types of queries and Valiant's stochastic setting.) P is the class of sets recognizable in deterministic polynomial time; it is a subclass of RP, and the question of strict containment is open.

The main result in this note is that if membership queries are available in addition to equivalence queries, there is an algorithm that exactly identifies any k-term DNF formula in time polynomial in n.

---

*Supported by NSF grant IRI-8404226

2 Preliminaries

2.1 Propositional formulas

Let $X_1, \ldots, X_n$ be propositional variables. An ***assignment*** to these variables is a function a from $X_1, \ldots, X_n$ to the set $\{0, 1\}$. An assignment is extended to all propositional formulas over the variables $X_1, \ldots, X_n$ in the usual way. An assignment may be represented compactly as an n-bit vector, with the i-th bit representing the value of $a(X_i)$.

A *literal* is one of the propositional variables or the negation of one of the propositional variables, *e.g.*, X_3 or $\neg X_6$. For each i, the complement of X_i is $\neg X_i$ and the complement of $\neg X_i$ is X_i. X_i and $\neg X_i$ are called a ***complementary pair.***

A ***term*** is the conjunction of a collection of literals that does not contain a complementary pair. For example, $X_1 \neg X_3 X_5$ is a term.

A ***disjunctive normal form formula*** ϕ is the disjunction of a finite sequence of terms. For example,

$$X_1 \neg X_3 X_5 + X_2 \neg X_3 \neg X_4$$

is a disjunctive normal form formula with two terms. By convention, the empty disjunction, denoted $\perp$, is assigned 0 by every assignment. Thus, $\perp$ is the everywhere false function. We use the abbreviation DNF for "disjunctive normal form" hereafter.

Two formulas ϕ and ϕ' over the variables $X_1, \ldots, X_n$ are ***equivalent*** if and only if for every assignment a to these variables, $a(\phi) = a(\phi')$.

2.2 Special notation

If a is an assignment and L is a literal, let $a[L]$ be the assignment obtained from a by forcing the literal L to be assigned the value 0. That is, if L is X_i then $a[L](X_i) = 0$ and $a[L](X_j) = a(X_j)$ for all $j \neq i$. If L is $\neg X_i$ then $a[L](X_i) = 1$ and $a[L](X_j) = a(X_j)$ for all $j \neq i$.

If $C = \{L_{i_1}, \ldots, L_{i_h}\}$ is a set of literals that does not contain a complementary pair, then we define $a[C]$ to be the assignment obtained from a by forcing the values on each of the literals in C to be 0. Then

$$a[C] = a[L_{i_1}] \ldots [L_{i_h}].$$

If ϕ is a DNF formula and a is any assignment satisfying ϕ, then the ***sensitive set*** of a with respect to ϕ is the set of literals L such that $a[L](\phi) = 0$. We denote the sensitive set of a with respect to ϕ by $S(a)$.

2.3 Queries

Let ϕ be a DNF formula ϕ over the variables $X_1, \ldots, X_n$ with k or fewer terms. The goal is to learn a formula equivalent to ϕ. The variables $X_1, \ldots, X_n$ are known, as is the value of k. Further information about ϕ comes from two types of queries: equivalence queries and membership queries. (The paper [1] contains a discussion of types of queries.)

A ***membership query*** proposes an assignment a to the variables $X_1, \ldots, X_n$. The reply is ***yes*** if $a(\phi) = 1$ and ***no*** if $a(\phi) = 0$.

An *equivalence query* proposes a DNF formula ϕ' over the variables $X_1, \ldots, X_n$ with at most k terms. The reply is *yes* if ϕ' is equivalent to ϕ. The reply is *no* if ϕ' is not equivalent to ϕ, and in this case the reply also contains an assignment a such that $a(\phi) \neq a(\phi')$. The assignment a is called a *counterexample* because it witnesses the inequivalence of ϕ' to the unknown formula ϕ.

The goal of the learning algorithm is to find a DNF formula over the variables $X_1, \ldots, X_n$ with at most k terms that is equivalent to ϕ. An algorithm that accomplishes this is said to perform *exact identification*. One strategy is to enumerate k-term DNF formulas and use equivalence queries to find one equivalent to ϕ, but in general this strategy will not run in time polynomial in n.

3 The case $k = 1$

There is a simple algorithm in case ϕ consists of at most one term, which we describe here to help motivate the general algorithm.

The learning algorithm first tests the constant false function, $\perp$, using an equivalence query. If the reply is *yes*, the algorithm outputs $\perp$ and halts. If the reply is *no*, then a counterexample a is also returned. Then we know that ϕ consists of one term t and $a(t) = 1$.

Then the algorithm determines the sensitive set, $S(a)$, of a with respect to ϕ by using membership queries. In particular, for each literal L such that $a(L) = 1$, the algorithm does a membership query to determine whether $a[L](\phi) = 0$. It then outputs the term t' that is the conjunction of all the literals in $S(a)$.

We claim that t' is equal to t. Consider any literal L in t. When this literal is assigned the value 0, the term t and therefore the formula ϕ are also assigned the value 0. Hence every literal in t is in $S(a)$.

Conversely, suppose L is a literal not in t. If the complement of L is also not in t, then the value of $a[L]$ is equal to the value of a on all the literals in t, so $a[L](\phi) = 1$. If the complement of L is in t, then the value of a is already 0 on L, so $a[L]$ is equal to a, and $a[L](\phi) = 1$. In either case, L is not in $S(a)$. Thus, the literals in t are precisely those in t'.

This algorithm makes a maximum of one equivalence query and n membership queries, and clearly runs in time polynomial in n. If we try simply to extend this strategy when $k > 1$, we are stymied by the fact that when we change the assignment to a literal it may affect several terms, obscuring the effect of the change. However, a slightly more complex generalization works.

4 Main result

This section is devoted to a proof of the following.

Theorem 1 *Let $k \geq 0$. There is an algorithm that exactly identifies any k-term DNF formula over n variables using equivalence and membership queries that runs in time polynomial in n.*

We first develop a little more machinery.

4.1 Nonredundancy

Let

$$\phi = t_1 + \ldots + t_k$$

be a k-term DNF formula over the n variables $X_1, \ldots, X_n$. We say ϕ is ***redundant*** if for some i, the formula ϕ' obtained from ϕ by removing t_i is equivalent to ϕ. If ϕ is not redundant, it is ***nonredundant***.

Lemma 2 *Suppose the formula ϕ is nonredundant. Then for each i, there exists an assignment a_i to $X_1, \ldots, X_n$ such that $a_i(t_i) = 1$ but for each $j \neq i$, $a_i(t_j) = 0$. That is, a_i satisfies term t_i and none of the rest of the terms.*

Suppose that for some i, for every assignment a such that $a(t_i) = 1$ there is some $j \neq i$ such that $a(t_j) = 1$. Then let ϕ' be obtained from ϕ by removing term t_i. Clearly, for every assignment a, if $a(\phi') = 1$ then $a(\phi) = 1$.

Conversely, if a is any assignment such that $a(\phi) = 1$, then for some j, $a(t_j) = 1$. If $j \neq i$ then $a(\phi') = 1$. If $j = i$, then there is some $h \neq i$ such that $a(t_h) = 1$, so also $a(\phi') = 1$. Hence for every assignment a, $a(\phi) = a(\phi')$. Thus the two formulas are equivalent and ϕ is redundant, contrary to hypothesis. This proves Lemma 2.

4.2 Discriminants

A discriminant of a DNF formula gives us a way of focusing on a single term of the formula. Let $k \geq 1$ and define the index set

$$I_k = \{(i,j) : 1 \leq i \neq j \leq k\}.$$

Note that I_k is empty if $k = 1$. Let ϕ be a DNF formula with k terms. A ***discriminant*** for ϕ is an indexed collection of literals L_{ij} for $(i,j) \in I_k$ such that

1. For every $(i,j) \in I_k$, L_{ij} is a literal that is in t_i and not in t_j.

2. If t_i and t_j contain a complementary pair of literals, then L_{ij} and L_{ji} are a complementary pair of literals.

3. For each $i = 1, \ldots, k$, the set $\{L_{ji} : j \neq i\}$ does not contain a complementary pair.

For each i, let L_{i*} denote the set of literals $\{L_{ij} : j \neq i\}$. Analogously, let L_{*i} denote the set of literals $\{L_{ji} : j \neq i\}$. Then L_{i*} is a subset of the literals of t_i, and condition (3) above states that L_{*i} does not contain a complementary pair. By convention, the empty collection of literals is a discriminant for a DNF formula with one term.

Lemma 3 *If ϕ is a nonredundant DNF formula with $k \geq 1$ terms then a discriminant exists for ϕ.*

Since ϕ is not redundant, for each $i = 1, \ldots, k$ there exists an assignment a_i that satisfies t_i but satisfies no other term of ϕ, by Lemma 2. We describe how to construct a discriminant for ϕ.

For each pair $(i,j) \in I_k$, if t_i and t_j contain a complementary pair of literals, say X_h and $\neg X_h$, then let L_{ij} be the member of this pair that appears in t_i and L_{ji} be the member of this pair that appears in t_j. Note that $a_j(L_{ji}) = 1$ because $a_j(t_j) = 1$, so $a_j(L_{ij}) = 0$.

Otherwise, let L_{ij} be any literal in the term t_i such that $a_j(L_{ij}) = 0$. There must be at least one such, since $a_j(t_i) = 0$. The literal L_{ij} cannot appear in t_j since $a_j(t_j) = 1$.

Finally, note that for each $i = 1, \ldots, k$, the set L_{*i} contains only literals that are assigned the value 0 by a_i, and so cannot contain a complementary pair. This proves Lemma 3.

The key lemma on the use of a discriminant is the following.

Lemma 4 *Let $k \geq 1$. Let $\phi = t_1 + \ldots + t_k$ be a nonredundant DNF formula and let $L_{ij}, (i,j) \in I_k$ be a discriminant for ϕ. For any $i = 1, \ldots, k$, let a be any assignment such that $a[L_{*i}](t_i) = 1$. Then the literals in t_i are precisely those in $L_{i*} \cup S(a[L_{*i}])$.*

By the definition of a discriminant, L_{*i} does not contain a complementary pair of literals, so the assignment $a[L_{*i}]$ is well defined. By hypothesis, $a[L_{*i}]$ assigns 1 to t_i. If $j \neq i$ then t_j contains a literal from L_{*i}, so $a[L_{*i}](t_j) = 0$. In particular, $a[L_{*i}](\phi) = 1$, so the sensitive set of $a[L_{*i}]$ with respect to ϕ is well defined.

Let L be any literal in $L_{i*} \cup S(a[L_{*i}])$. Clearly if L is in L_{i*} then it is in t_i, so assume L is in $S(a[L_{*i}])$. This means that $a[L_{*i}][L](\phi) = 0$, so $a[L_{*i}][L](t_i) = 0$. But $a[L_{*i}](t_i) = 1$, so L must be in t_i. Thus the literals in $L_{i*} \cup S(a[L_{*i}])$ are a subset of the literals of t_i.

Conversely, suppose L is a literal in t_i. If L is in L_{i*} then L is in the union of L_{i*} and $S(a[L_{*i}])$, so assume L is not in L_{i*}. We prove that L is in $S(a[L_{*i}])$ by showing that $a[L_{*i}][L](\phi) = 0$.

Since L is in t_i, $a[L_{*i}][L](t_i) = 0$. Suppose $j \neq i$. We know $a[L_{*i}](t_j) = 0$, and we need to show $a[L_{*i}][L](t_j) = 0$. We consider three cases, as follows.

If t_i and t_j contain no complementary pair of literals, then since L is in t_i, the complement of L is not in t_j, so $a[L_{*i}][L](t_j) = 0$.

If t_i and t_j contain exactly one complementary pair of literals, then they are L_{ij} and L_{ji}, by the definition of a discriminant. Since L is not in L_{i*}, it is not equal to L_{ij}, and the complement of L is not in t_j, so $a[L_{*i}][L](t_j) = 0$.

Finally, if t_i and t_j contain more than one pair of complementary literals, then since $a[L_{*i}](t_i) = 1$, $a[L_{*i}]$ must assign 0 to at least two distinct literals of t_j, namely, the complements of the literals in t_i. Thus, $a[L_{*i}][L]$, which changes the value of $a[L_{*i}]$ on only one variable, cannot assign 1 to all the literals of t_j, so $a[L_{*i}][L](t_j) = 0$.

Hence in each of the three cases, $a[L_{*i}][L](t_j) = 0$, and this holds for an arbitrary $j \neq i$, so $a[L_{*i}][L](\phi) = 0$, and L is in $S(a[L_{*i}])$. Thus the literals of t_i are a subset of $L_{i*} \cup S(a[L_{*i}])$. This completes the proof of Lemma 4.

4.3 The main subprocedure

The learning algorithm will enumerate possible discriminants for the unknown formula and try them out. The subprocedure *TRY* takes as input the value of $k \geq 1$ and an indexed set of literals, L_{ij} for $(i,j) \in I_k$. It assumes that this is a discriminant for ϕ and attempts to learn ϕ using it. The returned value is either a formula equivalent to ϕ or the special value *fail.*

The subprocedure *TRY*

1. Initialize T to be the empty set. Initialize J to be the set $\{1, 2, \ldots, k\}$. Then repeat steps (2) through (8) until a value is returned.

2. Let ϕ' be the disjunction of the terms in T, and use an equivalence query to test ϕ' for equivalence with ϕ. If they are equivalent, return the value ϕ'. If they are inequivalent, then let a be the counterexample returned.

3. If T already contains k terms, return the value *fail.*

4. Use membership queries to find the least $i \in J$ such that $a[L_{*i}](\phi) = 1$.

5. If there is no such value i, return the value *fail.*

6. Remove i from J.

7. Use membership queries to determine the set $S(a[L_{*i}])$.

8. Let t be the conjunction of all the literals in $L_{i*} \cup S(a[L_{*i}])$, and add the term t to T.

The following lemma says that if the subprocedure *TRY* is given a correct discriminant for ϕ, it will return a formula equivalent to ϕ.

Lemma 5 *Let $k \geq 1$. Let $\phi = t_1 + \ldots + t_k$ be a nonredundant DNF formula. Suppose L_{ij} for $(i,j) \in I_k$ is a discriminant for ϕ. Then the subprocedure TRY will return a k-term DNF formula equivalent to ϕ.*

If $k = 1$, then ϕ consists of one term, and the empty collection of literals is a discriminant for ϕ, with L_{1*} and L_{*1} both equal to the empty set. It is not difficult to verify that in this case *TRY* reduces to the algorithm described in Section 3 for $k = 1$, and therefore returns ϕ.

So, assume that $k \geq 2$. We show by induction that each iteration of steps (2) through (8) correctly discovers another term of ϕ. The induction hypothesis is that T contains a subset of the terms of ϕ, and J contains the indices of the terms of ϕ that are not in T. This is true after the initialization step, since T is empty, and J contains the numbers 1 through k.

If T contains k terms at the beginning of the iteration, it must contain all the terms of ϕ, so *TRY* will discover that ϕ' is equivalent to ϕ and return ϕ'. If T contains fewer than k terms, then since ϕ is nonredundant ϕ' cannot be equivalent to ϕ, so there will be another iteration. Since $|T| < k$, the value *fail* will not be returned in step (3).

The counterexample a must be a positive one, that is, $a(\phi') = 0$ and $a(\phi) = 1$. Let j be the least integer such that $a(t_j) = 1$. Clearly j is still in J, since t_j is not in T. Since none of the literals in L_{*j} are in t_j, $a[L_{*j}](t_j) = 1$. Thus the search in step (4) will succeed in finding some $i \in J$ such that $a[L_{*i}](\phi) = 1$. In particular, $i \leq j$.

Then *TRY* goes on to compute $S(a[L_{*i}])$ and places in T the term t that is the conjunction of the literals in L_{i*} and $S(a[L_{*i}])$, which by Lemma 4, is just the term t_i. It also removes i from J. Thus the induction hypothesis is preserved by this iteration.

Hence at each iteration *TRY* correctly discovers another term of ϕ, and must therefore halt and output a formula equal to ϕ up to rearrangement of terms after k iterations. This proves Lemma 5.

We now analyze the running time of *TRY*.

Lemma 6 *The subprocedure TRY makes at most $k+1$ equivalence queries, at most $kn + k(k+1)/2$ membership queries, and runs in time polynomial in n and k.*

There are at most k iterations of the steps (2) through (8), so there are at most $k+1$ equivalence queries. Each iteration may make at most $|J|$ membership queries to find a value of i in step (4) and at most n membership queries to determine the set $S(a[L_{*i}])$ in step (7), for a total of at most $kn + k(k+1)/2$ membership queries. A straightforward implementation clearly runs in time polynomial in n and k, proving Lemma 6.

4.4 The learning algorithm

The value of k and the variables $X_1, \ldots, X_n$ are known to the learning algorithm, and there is an unknown k-term DNF formula ϕ. The goal is to find a k-term DNF formula equivalent to ϕ by using equivalence and membership queries.

The learning algorithm

1. The learning algorithm makes an equivalence query with $\perp$. If the reply is *yes*, it outputs $\perp$ and halts.

2. Otherwise, for each $r = 1, \ldots, k$, it enumerates every indexed sequence of literals L_{ij} for $(i,j) \in I_r$ and calls *TRY* with inputs r and L_{ij} for $(i,j) \in I_r$. If *TRY* returns a formula ϕ', then the learning algorithm outputs ϕ' and halts. Otherwise, the returned value is *fail*, and the learning algorithm continues.

If the subprocedure returns a formula ϕ', then it is a DNF formula with at most k terms that is equivalent to ϕ, so the correctness of the learning algorithm is immediate.

To see that it must eventually halt, note that for some $r \geq 0$, ϕ is equivalent to a nonredundant DNF formula ϕ'' with r terms. If $r = 0$ then ϕ is equivalent to $\perp$ and the learning algorithm outputs $\perp$ and halts after the first query.

If $r \geq 1$ then since ϕ'' is nonredundant, there is a discriminant L_{ij} for $(i,j) \in I_r$, by Lemma 3. If the *TRY* is called with r and this discriminant for ϕ'' as input, it will return a DNF formula with r terms equivalent to ϕ'', by Lemma 5. Hence the learning algorithm must halt, since if it hasn't halted before, it will certainly halt after it calls *TRY* on this input.

How many collections of literals indexed by I_r are there? There are $2n$ choices of literals, and $r(r-1)$ pairs in the index set, so there are $(2n)^{r(r-1)}$ collections of literals indexed by I_r. Summing over r, we find that there are no more than $k(2n)^{k(k-1)}$ collections of literals enumerated by the learning algorithm.

Thus the learning algorithm must terminate after at most $k(2n)^{k(k-1)}$ calls to *TRY*, and each of these takes time bounded by a polynomial in n and k. Hence the total time used by the learning algorithm is bounded by a polynomial in n, with an $O(k^2)$ term in the exponent. This concludes the proof of Theorem 1.

5 An example

To illustrate the learning algorithm, consider the formula

$$\phi = X_1 \neg X_2 X_3 + X_1 X_4 + X_2 X_3.$$

This is a nonredundant formula, and one discriminant for it is

$$\begin{aligned} L_{12} &= X_3 \\ L_{21} &= X_4 \\ L_{13} &= \neg X_2 \\ L_{31} &= X_2 \\ L_{23} &= X_4 \\ L_{32} &= X_3. \end{aligned}$$

To check that this is a legal discriminant for ϕ, note that for each pair $(i,j) \in I_3$, L_{ij} is in t_i and not in t_j. Also, t_1 and t_3 contain a complementary pair of literals, $\neg X_2$ and X_2, and the values of L_{13} and L_{31} are a complementary pair. Finally, note that

$$\begin{aligned} L_{*1} &= \{X_2, X_4\} \\ L_{*2} &= \{X_3\} \\ L_{*3} &= \{\neg X_2, X_4\}. \end{aligned}$$

Thus for no i does L_{*i} contain a complementary pair of literals.

We now illustrate a run of *TRY* with inputs $k = 3$ and this discriminant for ϕ. Initially the set T of terms is empty and $J = \{1,2,3\}$.

The subprocedure tests $\perp$ for equivalence to ϕ and receives the reply *no* and a counterexample, say

$$a_1 = 1010.$$

The search in step (4) begins with the least element of J, namely 1, and forms the assignment

$$a_1[L_{*1}] = 1010,$$

which is equal to a_1 because a_1 is already 0 on X_2 and X_4. Next is a (superfluous) membership query to find that

$$a_1[L_{*1}](\phi) = 1.$$

Thus the search in step (4) succeeds with $i = 1$, and J is set to $\{2,3\}$.

Membership queries are used to determine that

$$\begin{aligned} 0010(\phi) &= 0 \\ 1110(\phi) &= 1 \\ 1000(\phi) &= 0 \\ 1011(\phi) &= 1. \end{aligned}$$

Thus $S(a_1[L_{*1}]) = \{X_1, X_3\}$. Since $L_{1*} = \{\neg X_2, X_3\}$, the term

$$t = X_1 \neg X_2 X_3$$

is added to T.

The next iteration starts with an equivalence query with the formula

$$\phi' = X_1 \neg X_2 X_3.$$

The reply is *no*, and a counterexample is returned, say

$$a_2 = 0111.$$

The search in step (4) takes the least element of J, now 2, and forms

$$a_2[L_{*2}] = 0101.$$

A membership query shows that $a_2[L_{*2}](\phi) = 0$, so the search moves to the next element of J, namely 3. A membership query with

$$a_2[L_{*3}] = 0110$$

shows that $a_2[L_{*3}](\phi) = 1$, so the search succeeds with $i = 3$. Then 3 is removed from J, leaving $J = \{2\}$.

Then membership queries are done to determine that

$$\begin{aligned} 1110(\phi) &= 1 \\ 0010(\phi) &= 0 \\ 0100(\phi) &= 0 \\ 0111(\phi) &= 1. \end{aligned}$$

Thus $S(a_2[L_{*3}]) = \{X_2, X_3\}$. Since $L_{3*} = \{X_2, X_3\}$, the term

$$t = X_2 X_3$$

is added to T, which now is $\{X_1 \neg X_2 X_3, X_2 X_3\}$.

The next iteration does an equivalence query with the formula

$$\phi' = X_1 \neg X_2 X_3 + X_2 X_3$$

and the reply is *no* and a counterexample is returned, say

$$a_3 = 1101.$$

The search in step (4) begins with the least (and only) element of J, now 2, and calculates

$$a_3[L_{*2}] = 1101.$$

A (superfluous) membership query determines that $a_3[L_{*2}](\phi) = 1$, so the search succeeds with 2. Then 2 is removed from J, which is now empty.

Membership queries are used to determine

$$\begin{aligned} 0101(\phi) &= 0 \\ 1001(\phi) &= 1 \\ 1111(\phi) &= 1 \\ 1100(\phi) &= 0. \end{aligned}$$

Thus, $S(a_3[L_{*2}]) = \{X_1, X_4\}$, and $L_{2*} = \{X_4\}$, so the term

$$t = X_1 X_4$$

is added to T.

At the next iteration, the equivalence query is with the formula

$$\phi' = X_1 \neg X_2 X_3 + X_2 X_3 + X_1 X_4,$$

to which the reply is *yes*, and this is the formula returned by *TRY*.

6 Remarks

By logical duality, there is an algorithm to learn k-clause CNF formulas over n variables using equivalence and membership queries that runs in time bounded by a polynomial in n.

The algorithm described above contains numerous redundancies that should be eliminated if it is to be implemented. (Which is not out of the question for small k.)

The paper [1] claimed that the k-term DNF formulas are exactly identifiable in time polynomial in n using equivalence and subset queries, although the proof given did not support the claim. The result in this note supersedes that claim.

References

[1] D. Angluin. *Types of queries for concept learning.* Technical Report, Yale University Computer Science Dept., TR-479, 1986.

[2] M. Kearns, M. Li, L. Pitt, and L. Valiant. On the learnability of boolean formulae. In *Proceedings of the Nineteenth Annual ACM Symposium on Theory of Computing*, pages 285–295, ACM, 1987.

[3] L. Pitt and L. Valiant. *Computational limitations on learning from examples.* Technical Report, Harvard University, Center for Research in Computing Technology, TR-05-86, 1986.

[4] L. G. Valiant. A theory of the learnable. *C. ACM*, 27:1134–1142, 1984.