Submitted to *Machine Learning* for publication.

# Malicious Omissions and Errors in Answers to Membership Queries

Dana Angluin* Mārtiņš Kriķis* Robert H. Sloan† György Turán‡

May 1, 1995

**Abstract**

We consider two issues in polynomial-time exact learning of concepts using membership and equivalence queries: (1) errors or omissions in answers to membership queries, and (2) learning finite variants of concepts drawn from a learnable class.

To study (1), we introduce two new kinds of membership queries: limited membership queries and malicious membership queries. Each is allowed to give incorrect responses on a maliciously chosen set of strings in the domain. Instead of answering correctly about a string, a limited membership query may give a special "I don't know" answer, while a malicious membership query may give the wrong answer. The table-size of the set of strings that receive such incorrect answers is bounded by a new parameter $L$. Equivalence queries are answered correctly, and learning algorithms are allowed time polynomial in the usual parameters and $L$. Any class of concepts learnable in polynomial time

*Partially supported by NSF grant CCR-9213881. Department of Computer Science, Yale University, P.O. Box 208285, New Haven, CT 06520. E-mail: angluin@cs.yale.edu and krikis@cs.yale.edu

†Partially supported by NSF grants CCR-9108753 and CCR-9314258. Dept. of Electrical Eng. and Computer Science, 851 S. Morgan St. Rm 1120, University of Illinois at Chicago, Chicago, IL 60607. E-mail: sloan@eecs.uic.edu

‡Partially supported by NSF grant CCR-9208170, Esprit BRA ILP, Project 6020, and OTKA T014228. Dept. of Math., Stat., and Comp. Sci., 851 S. Morgan St. Rm 322, University of Illinois at Chicago, Chicago, IL 60607, Automata Theory Research Group Hungarian Academy of Sciences, Szeged. E-mail: U11557@uicvm.uic.edu

using equivalence and malicious membership queries is learnable in polynomial time using equivalence and limited membership queries; the converse is an open problem. For the classes of monotone monomials and monotone $k$-term DNF formulas, we present polynomial-time learning algorithms using limited membership queries alone. We present polynomial-time learning algorithms for the class of monotone DNF formulas using equivalence and limited membership queries, and using equivalence and malicious membership queries.

To study (2), we consider classes of concepts that are polynomially closed under finite exceptions and a natural operation to add exception tables to a class of concepts. Applying this operation, we obtain the class of monotone DNF formulas with finite exceptions. We give a polynomial-time algorithm to learn the class of monotone DNF formulas with finite exceptions using equivalence and membership queries. We also give a general transformation showing that any class of concepts that is polynomially closed under finite exceptions and is learnable in polynomial time using standard membership and equivalence queries is also polynomial-time learnable using malicious membership and equivalence queries. Corollaries include the polynomial-time learnability of the following classes using malicious membership and equivalence queries: deterministic finite acceptors (DFA's), boolean decision trees, and monotone DNF formulas with finite exceptions.

# 1 Introduction

There is an impressive and growing number of polynomial-time algorithms, many of them quite beautiful and ingenious, to learn various interesting classes of concepts using equivalence and membership queries. To apply such algorithms in practice, researchers need to overcome a number of problems.

One significant issue is the problem of omissions and errors in answers to queries. Previous learning algorithms in the equivalence and membership query model are guaranteed to perform well assuming that queries are answered correctly, but there is often no guarantee that the performance of the algorithm will "degrade gracefully" if that assumption is not exactly satisfied.

Lang and Baum [19] report that this problem derailed their attempt to apply Baum's algorithm for learning neural nets from examples and membership queries [9] to the problem of recognizing hand-written digits. The attempt failed because the membership questions posed by the algorithm were too difficult for people to answer reliably. The algorithm typically asked membership queries on, say, a random-looking blur midway between a "5" and a "7," and the humans being queried gave very inconsistent responses. Studies in cognitive psychology indicate that this is the norm; people are typically quite inconsistent in deciding where the precise boundary of a concept lies. (See, e.g., [1].)

### 1.1 Omissions and Limited Membership Queries

This motivated us to introduce the limited membership query. A limited membership query may be answered either correctly, or with an omission, that is, a special value signifying "I don't know." The answers are persistent, that is, repeated queries about the same example are given the same answer. The choice of the set of strings on which to give answers of "I don't know" is assumed to be made by a malicious adversary. We introduce a new parameter $L$ to quantify the "amount" of omission — it is a bound on the table-size of the set of strings on which the adversary answers "I don't know." (The table-size of a set of strings is the number of strings in the set plus the sum of their lengths.) A polynomial-time learning algorithm is permitted time polynomial in the usual parameters and $L$.

For this model, we define a hypothesis to be "nonstrictly" correct if it agrees with the target concept on all examples except possibly ones for which a limited membership query was answered "I don't know." Thus, domain elements answered with "I don't know" are allowed to be classified arbitrarily by the final hypothesis of a learning algorithm. This corresponds to the intuition that since a person could not be sure of the classification of a "blur between 5 and 7", it does not matter how the final hypothesis classifies it.

We give a polynomial-time learning algorithm using just limited membership queries for the class of monotone monomials, and a lower bound on the query complexity of this problem. We also give a polynomial-time learning algorithm in this model for the class of $k$-term monotone DNF formulas.

We also consider combining limited membership queries with equivalence queries. We assume that the answers to equivalence queries remain correct, that is, any counterexample given is truly a counterexample to the hypothesis of the learning algorithm. In the nonstrict model, the equivalence query is answered "yes" if the hypothesis is nonstrictly correct; otherwise a counterexample must be returned from among examples not answered with "I don't know." In the strict model, equivalence queries remain as usual, that is, an equivalence query is answered with "yes" if the hypothesis is exactly equivalent to the target concept; otherwise an arbitrary counterexample is returned (including possibly an example previously answered with "I don't know.")

We show that the same classes of concepts can be learned in polynomial-time using limited membership queries and equivalence queries in the nonstrict and strict models. We give a polynomial-time algorithm to learn monotone DNF formulas using limited membership queries and equivalence queries in the nonstrict model.

## 1.2 Malicious Membership Queries and Errors

In the case of limited membership queries, the answers that are not omissions are guaranteed to be correct. We also consider the situation in which the answers to membership queries may be wrong. In a malicious membership query, the answer given may be correct, or it may be an error. As in the case of limited membership queries, we use the parameter $L$ to bound the table-size of the set of strings whose malicious membership queries are answered erroneously, the choice of that set of strings is assumed to be made by a malicious adversary, and the answers to queries are persistent. We assume that equivalence queries remain correct, which corresponds to the strict model introduced above. That is, the final hypothesis of the learning algorithm must be exactly equivalent to the target concept.

We give a polynomial-time algorithm to learn monotone DNF formulas using malicious membership queries and equivalence queries.

It is not difficult to see that any concept class learnable in polynomial time using malicious membership queries and equivalence queries is learnable in polynomial time using limited membership queries and equivalence queries, but the converse direction is an interesting open question. We exhibit a class of concepts for which the query complexity for equivalence and malicious membership queries is not bounded by any polynomial in the query complexity for equivalence and limited membership queries, but both complexities are exponential in the number of strings that receive incorrect answers.

## 1.3 Finite Exceptions

A related issue is the assumption that the target concept is drawn from a particular class of concepts, for example, monotone DNF formulas. Even if the target concept is "nearly" a monotone DNF formula, there is typically no guarantee that the learning algorithm will do anything reasonable. We approach this question by considering finite variants of the concepts in a given class, using the table-size of the set of exceptions as a measure of "how different" the target concept is from one in the specified class.

We define what it means for a concept class to be polynomially closed under finite exceptions. Some concept classes, for example, DFA's and decision trees, are polynomially closed under finite exceptions, while others, like monotone DNF formulas, are not. For the latter, we define a natural operation of adding exception tables to the concept class to make it polynomially closed under exceptions. We give a polynomial-time learning algorithm for the resulting class of monotone DNF formulas with finite exceptions, using equivalence queries and standard membership queries.

We give a general transformation that shows that any class of concepts that is polynomially closed under exceptions and polynomial-time learnable using equivalence queries and standard membership queries is also polynomial-time learnable using equivalence queries and malicious membership queries. Corollaries include polynomial-time learning algorithms using equivalence queries and malicious membership queries for the concept classes of DFA's, decision trees, and monotone DNF formulas with finite exceptions.

The notion of a finite variant of a concept, that is, a concept with a finite set of exceptions, is a unifying theme between the models of learning with equivalence and malicious membership queries and of learning a concept class with finite exceptions. Our model of errors in membership queries can be viewed as combining an equivalence oracle for the target concept and a membership oracle for a finite variant of the target concept. In the case of learning a concept class with finite exceptions, the equivalence and membership oracles present the same finite variant of a concept in the base class. In both cases, the goal is to identify exactly the concept presented by the equivalence oracle.

## 1.4 Related Work

There is a considerable body of literature on errors in examples in the PAC model, starting with the first error-tolerant algorithm in the PAC model, given by Valiant [24]. In this case the goal is PAC-identification of the target concept, despite the corruption of the examples by one or another kind of error, for example, random or malicious misclassification errors, random or malicious attribute errors, or malicious errors (in which both attributes and classification may be arbitrarily changed.)

There has been not as much work on omissions and errors in learning models in which membership queries are available, and the issues are not as well understood. One relevant distinction is whether the omissions of errors in answers to membership queries are persistent or not. They are *persistent* if repeated queries to the same domain element always return the same answer. In general, the case of persistent omissions or errors is more difficult, since non-persistent omissions or errors can yield extra information, and can always be made persistent simply by caching and using the first answer for each domain point queried.

## 1.5 Non-persistent Errors in Queries

Sakakibara defines one model of non-persistent errors, in which each answer to a query may be wrong with some probability, and repeated queries constitute independent events [22]. He gives a general technique of repeating each query sufficiently often to establish the correct answer with high probability. This yields a uniform transformation of existing query algorithms. The method also works for both of Bultman's models [12]. This could be a reasonable model of a situation in which the answers to queries were being transmitted through a medium subject to random independent errors; then the technique of repeating the query is eminently sensible.

A related model is considered by Dean et al. for the case of a robot learning a finite-state map of its environment using faulty sensors and reliable effectors [13]. This model assumes that observation errors are independent as long as there is a nonempty action sequence separating the observations. This means that there is no simple way to "repeat the same query", since a nonempty action sequence may take the robot to another state, and no reset operation is available. A polynomial-time learning algorithm is given for the situation in which the environment has a known distinguishing sequence. It achieves exact identification with high probability.

## 1.6 Persistent Errors in Membership Queries

The method of "repeating the query" is insufficient for the more difficult case of persistent omissions or errors in membership queries. In this case, we must exploit the error-correcting properties of groups of "related" queries. In an explicit and very interesting application of the ideas of error-correcting algorithms, Ron and Rubinfeld use the criterion of PAC-identification with respect to the uniform distribution, and give a polynomial-time randomized algorithm using membership queries to learn DFA's with high rates of random persistent errors in the answers to the membership queries [21].

Algorithms that use membership queries to estimate probabilities (in the spirit of the statistical queries defined by Kearns [17]) are generally not too sensitive to small rates of random persistent errors in the answers to queries. For example, Goldman, Kearns, and Schapire give polynomial-time algorithms for exactly learning read-once majority formulas and read-once positive NAND formulas of depth $O(\log n)$ with high probability using membership queries with high rates of persistent random noise or modest rates of persistent malicious noise [15]. As another example, Kushilevitz and Mansour's algorithm that uses membership queries and exactly learns logarithmic-depth decision trees with high probability in polynomial time seems likely to be robust under nontrivial rates of persistent random noise in the answers to queries [18].

However, learning algorithms for other classes of concepts using equivalence and membership queries may depend more strongly on the correctness of the answers to individual queries; in these cases, there is no guarantee of a learning algorithm for the class that can tolerate omissions or errors in the answers to membership queries.

One model that addressed these questions was introduced by Angluin and Slonim: equivalence queries are assumed to be answered correctly, while membership queries are either answered correctly or with "I don't know" and the answers are persistent. The "I don't know" answers are determined by independent coin flips the first time each query is made [7]. They give a polynomial-time algorithm to learn monotone DNF formulas with high probability in this setting. They also show that a variant of this algorithm can deal with one-sided errors, assuming that no negative point is classified as positive. Goldman and Mathias also consider this model [16]. Our current models and results differ in that the omissions and errors are chosen by a malicious adversary instead of a random process, and the rate of incorrect answers that can be tolerated is consequently much lower.

Frazier et al. [14] have introduced a model of omissions in answers to membership queries, called learning from a consistently ignorant teacher. The basic idea is to require that if the teacher gives answers to certain queries that would imply a particular answer to another query, the teacher cannot answer the latter query with "I don't know." For example, in the domain of monotone DNF formulas, if the teacher classifies a particular point as positive, then the teacher cannot answer "I don't know" about any of the ancestors of the point. The goal of the learner is to learn exactly the ternary classification of points into positive, negative, and "I don't know" that is presented by the teacher. Efficient learning algorithms are given in this model for monomials with at least one positive example, concepts represented as the agreement of a constant number of monotone DNF formulas, $k$-term DNF formulas, DFA's, or decision trees, and concepts represented by an agreement of boxes with samplable intersection. Compared to our model, this model has a different measure of the representational complexity of a concept with omissions, which allows a much higher rate of omissions to be compactly represented. It also differs in requiring the learner to reproduce exactly the "I don't know" labels of the teacher, whereas in our (nonstrict) model of omissions such examples can be classified arbitrarily.

# 2 Preliminaries

## 2.1 Concepts and Concept Classes

Our definitions for concepts and concept classes are a bit non-standard. We have explicitly introduced the domains of concepts in order to try to unify the treatment of fixed-length and variable-length domains. We take $\Sigma$ and $\Gamma$ to be two finite alphabets. Examples are represented by finite strings over $\Sigma$ and concepts are represented by finite strings over $\Gamma$.

A *concept* consists of a pair $(X, f)$, where $X \subseteq \Sigma^*$, and $f$ maps $X$ to $\{0,1\}$. The set $X$ is the *domain* of the concept. The *positive examples* of $(X, f)$ are those $w \in X$ such that $f(w) = 1$, and the *negative examples* of $(X, f)$ are those $w \in X$ such that $f(w) = 0$. Note that strings not in the domain of the concept are neither positive nor negative examples of it.

A *concept class* is a triple $(R, Dom, \mu)$, where $R$ is a subset of $\Gamma^*$, $Dom$ is a map from $R$ to subsets of $\Sigma^*$, and for each $r \in R$, $\mu(r)$ is a function from $Dom(r)$ to $\{0,1\}$. $R$ is the set of legal representations of concepts. For each $r \in R$, the *concept represented by* $r$ is $(Dom(r), \mu(r))$.

A concept $(X, f)$ is *represented by* a concept class $(R, Dom, \mu)$ if and only if for some $r \in R$, $(X, f)$ is the concept represented by $r$. The *size* of a concept $(X, f)$ represented by $(R, Dom, \mu)$ is defined to be the length of the shortest string $r \in R$ such that $r$ represents $(X, f)$. The size of $(X, f)$ is denoted by $|(X, f)|$; note that it depends on the concept class chosen.

The concept classes we consider in this paper are the boolean formulas and syntactically restricted subclasses of them, boolean decision trees, and DFA's. The representations are more or less standard, except each concept representation specifies the relevant domain. For DFA's, the domain of every concept is the set $\Sigma^*$ itself. For boolean formulas and decision trees, we assume that $\Sigma = \{0,1\}$, and each concept representation specifies a domain of the form $\{0,1\}^n$.

## 2.2 Exceptions

For each finite set $S$ of strings from $\Sigma^*$, we define its *table-size*, denoted $||S||$, as the sum of the lengths of the strings in $S$ and the number of strings in $S$. Note that $||S|| = 0$ if and only if $S = \emptyset$. For a concept $(X, f)$ and a finite set $S \subseteq X$, we define *the concept* $(X, f)$ *with exceptions* $S$, denoted $\mathit{xcpt}((X, f), S)$, as the concept $(X, f')$ where $f'(w) = f(w)$ for strings in $X - S$, and $f'(w) = 1 - f(w)$ for strings in $S$. (Thus $f$ and $f'$ have the same domain, and are equal except on the set of strings $S$, which is a subset of their common domain.) It is useful to note that $S$ is partitioned by $(X, f)$ into a set of *positive exceptions* $S_+$ and a set of *negative exceptions* $S_-$, defined by $S_+ \stackrel{\text{def}}{=} S - \{w \in X \mid f(w) = 1\}$ and $S_- \stackrel{\text{def}}{=} S \cap \{w \in X \mid f(w) = 1\}$.

A concept class $(R, \mathit{Dom}, \mu)$ is *closed under finite exceptions* provided that for every concept $(X, f)$ represented by $(R, \mathit{Dom}, \mu)$ and every finite set $S \subseteq X$, the concept $\mathit{xcpt}((X, f), S)$ is also represented by $(R, \mathit{Dom}, \mu)$. If, in addition, there is a fixed polynomial of two arguments such that the concept $\mathit{xcpt}((X, f), S)$ is of size bounded by this polynomial in the size of $(X, f)$ and $||S||$, we say that $(R, \mathit{Dom}, \mu)$ is *polynomially closed under finite exceptions.*

This definition differs from a similar earlier definition [10] in that we do not require the existence of a polynomial-time algorithm that produces the new concept given the old concept and a list of exceptions. However, for the classes that we consider there are such algorithms.

We define a natural operation of adding finite exception tables to a class of concepts to produce another class of concepts that "embeds" the first and is polynomially closed under finite exceptions.

We assume $\Sigma \subseteq \Gamma$ and $|\Gamma| \geq 2$. We define a simple encoding $e$ that takes a string $r$ from $\Gamma^*$ and a finite set of strings $S \subseteq \Sigma^*$ and produces a string $r'$ in $\Gamma^*$ from which $r$

and the elements of $S$ can easily be recovered, and is such that $|r'| = 2(1+|r|+||S||)$. The details of the encoding are as follows.

Assume that 0 and 1 are distinct symbols in $\Gamma$. We define

$$e_b(b_1 b_2 \ldots b_j) \stackrel{\text{def}}{=} bbbb_1 bb_2 \ldots bb_j,$$

for $b \in \{0,1\}$ and $b_1, b_2, \ldots, b_j \in \Gamma$. Note that $|e_b(w)| = 2(1+|w|)$ for every string $w \in \Gamma^*$. We then define the encoding of $r$ and $S$ as

$$r' = e(r,S) \stackrel{\text{def}}{=} e_0(r)e_1(s_1)e_0(s_2)\ldots e_{k \bmod 2}(s_k),$$

where $s_1, s_2, \ldots, s_k$ are the strings in $S$.

Given a concept class $(R, Dom, \mu)$, we define the *class obtained from it by adding exception tables* to be $(R', Dom', \mu')$, where $R'$ is the set of all strings of the form $e(r,S)$ such that $r \in R$ and $S$ is a finite subset of $Dom(r)$, and for each $r' \in R'$, the concept represented by $r' = e(r,S)$ is the concept represented by $r$ with exceptions $S$, that is, $(Dom'(r'), \mu'(r')) = xcpt((Dom(r), \mu(r)), S)$.

For example, adding exception tables to the monotone DNF formulas produces a concept class which we term *monotone DNF formulas with finite exceptions*. More detailed discussion of classes obtained by adding exception tables and of polynomial closure under finite exceptions can be found in Section 5.

## 2.3 Queries

Information about the target concept is available to the algorithm as the answers to two types of queries: equivalence queries and membership queries.

In an equivalence query, the algorithm gives as input a concept $r' \in R$ with the same domain as the target, and the answer depends on whether $\mu(r) = \mu(r')$. If so, the answer is "yes", and the learning algorithm has succeeded in its goal of exact identification of the target concept. Otherwise, the answer is a *counterexample*, any string $w \in Dom(r)$ on which the functions $\mu(r)$ and $\mu(r')$ differ. We denote an equivalence query on a hypothesis $h$ by $\mathrm{EQ}(h)$.

A *label* for a counterexample $v = \mathrm{EQ}(r')$ is $h^*(v)$ or, equivalently, $(1-h(v))$, where $h^*$ is the target concept and $h = \mu(r')$ is the concept that equivalence query was made with. The counterexamples that have label 1 are called *positive counterexamples* and the ones that have label 0 are called *negative counterexamples*.

In a membership query, the learning algorithm gives as input a string $w \in Dom(r)$, and the answer is either 0, 1, or $\perp$. If the answer is equal to the value of $\mu(r)$ on $w$, then the answer is *correct*. If the answer is equal to $\perp$, we say that the answer is an *omission* or a *"Don't know"*. If the answer is 0 or 1 but not equal to the value of $\mu(r)$ on $w$, then the answer is an *error* or a *lie*.

In a *standard membership query*, denoted MQ, all the answers are required to be correct. In a *limited membership query*, denoted LMQ, each answer is required to be correct or an omission. In a *malicious membership query*, denoted MMQ, each answer is required to be correct or an error (no omissions). Note that an answer of 0 or 1 to a limited membership query is necessarily correct, but this is not true for answers to malicious membership queries.

The answers to malicious and limited membership queries are also restricted as follows.

1. They are *persistent*, that is, different membership queries with the same input string $w$ receive the same answer. Note that non-persistent queries may reveal some information, namely, if one query on a string $w$ returns a different answer than another query on the same string, the algorithm knows that one of them is wrong. Every algorithm designed to work with persistent queries can be made to work with non-persistent ones by caching the queries and always using the first answer for subsequent queries of the same string.

2. In addition, we bound the quantity of errors (or omissions) permitted in answers to malicious (resp., limited) membership queries. One natural quantity to bound would be the number of different strings whose membership queries can be answered incorrectly, and this works well in fixed-length domains. However, in variable-length domains, we wish to account for the lengths of the strings as well as their number.

   Therefore, in general the algorithm is given a bound $L$ on the table-size, $||S||$, of the set $S$ of strings whose malicious (resp., limited) membership queries are answered erroneously (resp., with a $\perp$) during a single run. In the case of a fixed-length domain, $\{0,1\}^n$, we may instead give a bound $\ell$ on the number of different strings whose MMQ's (resp., LMQ's) are answered incorrectly (resp., with a $\perp$). Note that $L = \ell(n+1)$ is a bound on the table-size in this case.

   Note that when $L = 0$ or $\ell = 0$ there can be no errors in the answers to MMQ's (or LMQ's) and we have the usual model of standard membership queries as a special case.

We assume that an on-line adversary controls the choice of counterexamples in answers to equivalence queries and the choice of which elements of the domain will be answered with errors (or $\perp$'s) in malicious (or limited) membership queries. When the learning algorithms we consider are deterministic, the adversary may be viewed as choosing in advance the set of strings for which it will give incorrect answers to membership queries, as well as all the counterexamples it will give to equivalence queries.

In the setting of standard or malicious membership queries, the goal of a learning algorithm is exact identification of a target concept $(Dom(r), \mu(r))$ chosen from a known concept class $(R, Dom, \mu)$. We assume that the domain $Dom(r)$ of the target concept is also known to the learning algorithm (which for boolean formulas and decision trees means $n$, the number of attributes, is known).

An algorithm *exactly learns* $(R, Dom, \mu)$ *in polynomial time using equivalence and malicious membership queries* if and only if there exists a polynomial $p(s, n, L)$ such that for every $r \in R$, every nonnegative integer $L$, and every on-line adversary answering equivalence and malicious membership queries about $r$, the algorithm halts and outputs a concept $r' \in R$ with the same domain as $r$ and such that $\mu(r') = \mu(r)$, and the running time of the algorithm is bounded from above by $p(s, n, L)$. Here $s$ is the size of the target concept, $n$ is the maximum length of any counterexample given so far, and $L$ is a bound on the table-size of the strings for which MMQ's are answered incorrectly.

In the setting of limited membership queries we consider two different models of learning — the *strict* and the *nonstrict* model. The latter is used when studying learning from limited membership queries alone. If equivalence queries are also available, both models make sense and are considered. In the nonstrict model, equivalence queries may not give a counterexample from the set of strings that have been classified as $\perp$ by the LMQ, and it is assumed that correct identification is achieved once the learner outputs a concept that agrees with the target concept on all strings in the domain not counting those answered $\perp$ by the LMQ. This corresponds to our view that $\perp$ points form the borderline of the concept and that the classification of them is irrelevant or meaningless. In the strict model, however, counterexamples can be arbitrary and we require exact identification of the target concept, as with standard or malicious membership queries.

The definitions are extended in the usual way to cover randomized learning algorithms and their expected running times, and also extended equivalence queries, in which the inputs to equivalence queries and the final result of the algorithm are allowed to come from a concept class different from (usually larger than) the concept class from which the target is drawn.

It is straightforward to transform any algorithm that uses malicious membership queries into one that uses limited membership queries. Every $\perp$ answer can be replaced by a 0 or a 1 arbitrarily and given to the learner. Therefore learning with malicious membership queries is at least as hard as learning with limited membership queries in the strict model. The same applies to learning from equivalence and malicious membership queries and learning from equivalence and limited membership queries in the strict model. Furthermore, in Subsection 6.3 we show that the strict and the nonstrict models of learning from equivalence and limited membership queries are in fact polynomial-time equivalent.

## 2.4 Monotone DNF Formulas

We assume a set of propositional variables $V$ and denote its elements by $x_1, x_2, \ldots, x_n$, where $n$ is the cardinality of $V$. A monotone DNF formula over $V$ is a DNF formula over $V$ where no literal is negated. The domain of such a formula is $\{0,1\}^n$. For example, for $n = 20$,

$$x_1x_4 \vee x_2x_{17}x_3 \vee x_9x_5x_{12}x_3 \vee x_8$$

is a monotone DNF formula (with domain $\{0,1\}^{20}$) and a possible target concept in Sections 3 or 4. We will assume that the target formula $h^*$ has been minimized. (Incidentally, there is an efficient algorithm to minimize the number of terms of a monotone DNF formula.) For a monotone DNF formula $f$, let $\#(f)$ denote the number of terms in $f$. In the above example, $\#(f) = 4$.

We view the domain $\{0,1\}^n$ of monotone DNF formulas (with or without exceptions) as a lattice, with componentwise "or" and "and" as the lattice operations. The top element is the vector of all 1's, and the bottom element is the vector of all 0's. The elements are partially ordered by $\leq$, where $v \leq w$ if and only if $v[i] \leq w[i]$ for all $1 \leq i \leq n$. Often we refer to the examples as points of the hypercube $\{0,1\}^n$. For a point $v$, all points $w$ such that $w \leq v$ are called the *descendants* of $v$. Those descendants that can be obtained by changing exactly one coordinate of $v$ from a 1 to a 0 are called the *children* of $v$. The *ancestors* and the *parents* are defined similarly.

For convenience, we use a representation of monotone DNF formulas in which each term is represented by the minimum vector, in the ordering $\leq$, that satisfies the term. Thus, vector 10011 (where $n = 5$) denotes the term $x_1x_4x_5$. In this representation, if $h$ is a monotone DNF formula and $v$ is a vector in the domain, $v$ satisfies $h$ if and only if for some term $t$ of $h$, $t \leq v$. That is, a monotone DNF formula is satisfied only by the ancestors of its terms. In the other direction, we say that term $t$ *covers* point $v$ if and only if $v$ satisfies $t$.

For any $n$-argument boolean function $f$, we call point $x$ a *local minimum point* of $f$ if $f(x) = 1$ but for every child $y$ of $x$ in the lattice, $f(y) = 0$. The local minimum points of a minimized DNF formula represent its terms in our representation.

For two $n$-argument boolean functions $f_1$ and $f_2$ we define the set $Err(f_1, f_2)$ to be the set of points where they differ. I.e., $Err(f_1, f_2) = \{x \mid f_1(x) \neq f_2(x)\}$. The cardinality of $Err(f_1, f_2)$ is called the distance between $f_1$ and $f_2$ and is denoted by $d(f_1, f_2)$.

# 3 Limited Membership Queries

In this section, we present results concerning learning with the help of limited membership queries. We start with the description of a subroutine which is repeatedly used in all algorithms for this model.

## 3.1 Delimiting A Term

Algorithm Delimit takes a positive point and finds a set of candidates for a term in the target concept covering the point. This algorithm plays the role of the algorithms called "Reduce" in other works on learning monotone DNF [7]. We choose a different name because those algorithms output a single monomial, whereas Algorithm Delimit finds a set of points that must include a correct term.

Consider, for example, the situation in Figure 1 where the target concept is $x_1 x_2$, and we start with the known positive point $p = 1111$. Boldface $+$'s, $-$'s and ?s indicate responses 1, 0, and $\perp$, respectively, of the LMQ oracle. With complete information, we would begin by querying each child of $p$, updating $p$ to be the first positive child found. This process would be repeated until eventually we had $p = 1100$. After determining by membership queries that every child of $p$ is negative, we could stop.

Because Algorithm Delimit can make only limited membership queries, it may

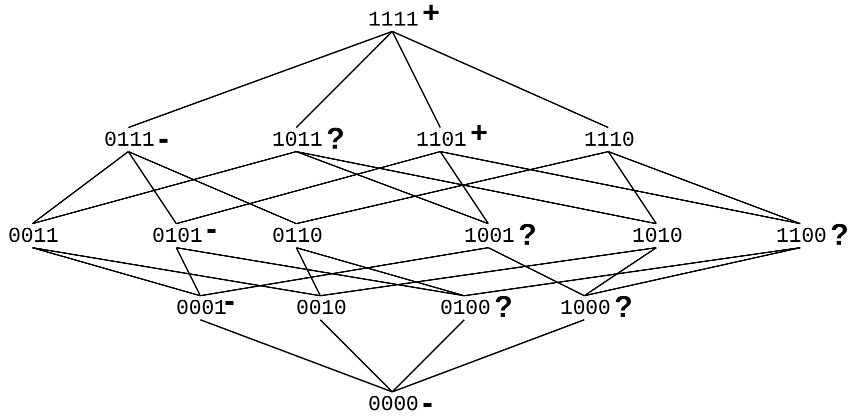


Figure 1: Example run of Algorithm DELIMIT for target concept $x_1x_2$.

DELIMIT($p$)
{
    $P = DK = \emptyset$
    $\mathit{offbits} = \underbrace{00\ldots0}_{n}$
    $\mathit{root} = $ DOWN($p$)
    UP()
    $P = P \cup \{\mathit{root}\}$
    $T = DK \cup \{\mathit{root}\}$
    **Return** $(T, P)$
}

Figure 2: Algorithm DELIMIT.

encounter the difficulty that some positive point $p$ has children that are all "Don't know," or a mix of "No" and "Don't know." For instance, in the example in Figure 1, all queries of children of 1101 return 0 or $\perp$. In this case, Algorithm DELIMIT continues by querying all the children of *all* the "Don't know" points. Should it ever get another 1 response to a limited membership query, it replaces $p$ by that point. What we have just described is the subroutine DOWN of DELIMIT, which invokes itself recursively upon finding a new positive point.

Detailed code of algorithm DELIMIT and its subroutines is given in Figures 2, 3 and 5. In our C-like metacode we often use a **For** loop over a changing set of points; for example, "**For** (each $b \in A$)". By this we mean that in each iteration of the loop the current point (i.e., $b$ in this example) is marked, and that only unmarked points can be considered in the next iterations. Furthermore, the loop condition is checked every time, that is, we check for an existence of some unmarked point in the possibly changing set. If one exists, we mark it and do the body of the loop which may add new (unmarked) points to the set or delete some existing points (either marked or unmarked). Points that were deleted before marking need not be processed.

```
Down(p)
{
    DK = ∅
    C = {c : c is a child of p  &&  c ≥ offbits}
    While (C ≠ ∅)
    {
        a = some maximal element of C
        Delete a from C
        Switch (LMQ(a))
        {
            Case 1:
                Return Down(a)
            Case 0:
                If (a is a child of p)
                    offbits = offbits | (a ⊕ p)
                Break
            Case ⊥:
                DK = DK ∪ {a}
                C = C ∪ {c : c is a child of a  &&  c ≥ offbits}
        }
    }
    Return p
}
```

Figure 3: Subroutine Down.

The subroutine Down uses a global variable *offbits* to improve efficiency. If a query to a child $a$ of a known positive point $p$ gives a 0 response, then we know that in any descendant of $p$, switching off the bit position that distinguishes $a$ from $p$ will lead to a negative point, because this point will be a descendant of $a$. Therefore, variable *offbits* keeps track of those bit positions that must be 1, allowing subroutine Down to save some queries. The line that updates *offbits* is merely a combination of the *bitwise or*, $|$, and the *bitwise exclusive or*, $\oplus$. The latter finds the bit position that was switched off, and the former marks that bit as 1 in *offbits*, so that it can never be switched off again.

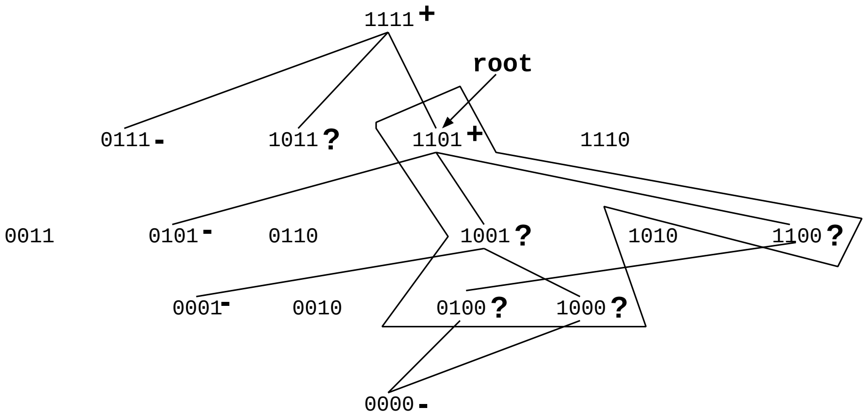


Figure 4: The set of candidate terms obtained by subroutine DOWN.

Eventually DOWN is called on some positive point $p$, where there is a set $DK$ of descendants of $p$ (possibly empty) such that the limited membership oracle responded $\perp$ for every point in $DK$, and all other descendants of $p$ are known to be negative. Then DOWN returns and point $p$ from then on is called the *root*. Since *root* is positive, and we know that every descendant of *root* not in set $DK$ is negative, a term of the monotone DNF must lie somewhere in the set $DK \cup \{root\}$. This set is outlined in Figure 4.

The next major part of DELIMIT is the subroutine UP. Any point corresponding to a term of the target concept must have only positive ancestors. Subroutine UP ensures that no point in $DK$ has any ancestor with LMQ 0. Thus, DELIMIT gets a thinned set of possible terms $T$. On the example of Figures 1 and 4, 0100 is deleted from $DK$ because LMQ(0101) = 0. If LMQ(1010) = 0, then 1000 will also be deleted. If, in addition, LMQ(1110) = 0 holds as well, then 0100, 1000, and 1100 will all be deleted.

As a useful byproduct of subroutine UP, we get a set $P$, which contains the minimal ancestors with LMQ 1 of the points in $DK$. If there are no points in $DK$, $P$ contains the *root* only.

We summarize our discussion in the following lemma.

```
Up()
{
    For (each a ∈ DK)
    {
        A = {parents of a}
        For (each b ∈ A)
            If (LMQ(b) == 0)
            {
                Delete all descendants of b from DK
                Break
            }
            Else If (LMQ(b) == 1)
                P = P ∪ {b}
            Else
                A = A ∪ {parents of b}
    }

}
```

Figure 5: Subroutine Up.

**Lemma 1** *Let $T$ and $P$ be the outputs of running Algorithm* Delimit *on a positive point $p$ when the target concept is a monotone DNF. Then*

1. *$T$ contains a term of the target concept covering $p$.*
2. *For every $t \in T$, every point covered by $t$ has* LMQ *1 or $\perp$.*
3. *For any $v$ such that* $\mathrm{LMQ}(v) = 1$, *and for any $t \in T$ that covers $v$, $t$ covers some point in $P$ that is a descendant of $v$.*

**Proof:** Part 1 follows from the discussion above. Part 2 holds since every ancestor of the *root* is positive (since *root* itself is), and since the code in Up ensures this condition for every other point in $T$. Part 3 holds because every point in $T$ covers *root* and thus all of its ancestors, and, for other points in $T$, because $P$ contains their minimal ancestors with LMQ 1. ■

We are now ready to analyze the running time of Delimit.

**Lemma 2** *For any monotone DNF target concept, Algorithm* DELIMIT *makes at most $n\ell + n$ limited membership queries, where $\ell$ is the number of $\perp$ responses received. In general, any sequence of $s$ calls to Algorithm* DELIMIT *for the same target concept makes at most $n(\ell + s)$ limited membership queries.*

**Proof:** Every point queried by DELIMIT($p$) is either a child of $p$, or a child of a previously queried point with LMQ 1, or else is within Hamming distance 1 of a $\perp$ point.

For the last case, each time we receive a $\perp$ response on a point $v$, we could in principle need to query all the children of $v$ in DOWN and then all the parents of $v$ in UP, except that there is a parent or a child of $v$ that we must have queried before querying $v$. This leads to at most $n - 1$ queries for each $\perp$ received, plus 1 for the $\perp$ query itself. As long as the algorithm remembers the answers to all queries, this holds for any number of calls to DELIMIT on the same concept.

For the first two cases, let us first note that they only happen in subroutine DOWN. Let us call the point that DOWN was called on the *current point*, which can be either $p$ or some previously queried point $a$ with LMQ($a$) $= 1$. When DOWN makes a query on a child of the current point and receives a 0 or 1 response, the Hamming distance between the current point (old or new) and the *offbits* decreases by 1. The maximum Hamming distance between any two points can be $n$, so there can be at most $n$ queries of this sort, in each call of DELIMIT. ■

The running time of DELIMIT is polynomial in the number of queries it makes.

## 3.2 Learning Monotone Monomials from Limited Membership Queries Alone

We begin with a very simple application of Algorithm DELIMIT to learn monotone monomials from limited membership queries in the nonstrict model. This should elucidate the basic ideas at the heart of the more complicated algorithm for learning arbitrary monotone $k$-term DNF from limited membership queries that follows.

**Theorem 1** *Monotone monomials can be learned from no more than $n\ell + n + 1$ limited membership queries in the nonstrict model, where $\ell$ is the number of $\perp$ responses received.*

**Proof:** The method is to run Algorithm DELIMIT starting with the all 1's vector, and then output any term $t \in T$ that covers every point in $P$. (If either $\mathrm{LMQ}(11\cdots 1) = 0$, or $\mathrm{LMQ}(11\cdots 1) = \perp$ and the "down" phase of Algorithm DELIMIT finds no positive points, then we will output FALSE.)

First, observe that such a term $t$ must exist in $T$, since the true target monomial covers every point in $P$ and lies in $T$ by point 1 of Lemma 1. The output term $t$ covers every point with LMQ 1 by point 3 of Lemma 1, since it covers every point in $P$. And by point 2 of Lemma 1, it cannot cover any points with LMQ 0, since it is in $T$. The bound on the number of queries follows from Lemma 2, which counts all of them except for the first query to $11\cdots 1$. ■

The running time again is polynomial in the number of queries made.

Another, and computationally somewhat more efficient way of learning monotone monomials would be to run DELIMIT on the all 1's vector, and then output the monomial $m$ that is the intersection of all the points in $P$. That is, the learner's output would have a variable if and only if that variable is in every term in $P$.

In this version, monomial $m$ covers every point in $P$, so by point 3 of Lemma 1, it must cover all points $v$ such that $\mathrm{LMQ}(v) = 1$. On the other hand, since $m$ is the intersection of positive points, it must either be the correct monotone monomial or an ancestor of the correct monotone monomial, so $m$ cannot cover any negative points.

Now we present a lower bound for this problem.

**Theorem 2** *For any integer $c$, if the limited membership oracle gives $\sum_{k=0}^{c} \binom{n}{k}$ responses of $\perp$, then any learner can be forced to use at least $\binom{n}{c+1} - 1$ limited membership queries* not counting those answered $\perp$ *to learn monotone monomials.*

**Proof:** Let the target concept be defined by a monomial of length $n - (c+1)$. It contains no point with more than $(c+1)$ 0-bits, and exactly one point with $(c+1)$ 0-bits. Now let us assume that all queries of a learning algorithm on points containing $c$ or fewer 0-bits are answered by $\perp$, and its first $\binom{n}{c+1} - 2$ queries of points with at least $(c+1)$ 0-bits are answered by 0. After that there are at least two unqueried points with $(c+1)$ 0-bits. The corresponding two monomials are both consistent with the answers given so far and they differ on points that did not receive a $\perp$ response. Hence the learning algorithm needs at least one more query. ■

**Corollary 1** *For any fixed constant $c$, if the limited membership oracle gives $O(n^c)$ responses of $\perp$, then the learner can be forced to use $\Omega(n^{c+1})$ limited membership queries to learn monotone monomials.*

## 3.3 Learning Monotone $k$-term DNF from Limited Membership Queries Alone

We are now ready to present the algorithm OnlyLMQ, that learns monotone $k$-term DNF in the nonstrict model from limited membership queries alone. The detailed code of it is given in Figures 6 and 7.

FindPos($h$)
{
  $S = \{$maximal elements $v \in \{0,1\}^n$ s.t. $h(v) == 0\}$
  **For** (each $a \in S$)
    **If** (LMQ($a$) == 1)
      **Return** $a$
    **Else If** (LMQ($a$) == $\perp$)
      $S = S \cup \{$children of $a\}$
  **Return** "no"
}

Figure 6: Subroutine FindPos.

Algorithm OnlyLMQ initializes its hypothesis $h$ to be the empty formula, and repeats the following.

By searching down from each maximal point not covered by hypothesis $h$, a point $q$ with LMQ($q$) = 1 that is not covered by $h$ is found. This is done by the subroutine FindPos of Algorithm OnlyLMQ. If no such point is found, then $h$ is correct and the algorithm outputs it and halts.

Now the point $q$ is given to Algorithm Delimit, which returns the sets $T$ and $P$. If there is a single term in $T$ that covers all the positive points in $P$ not already covered by $h$, then this term is added to $h$ and we repeat the main **For** loop by looking for another point with LMQ 1 that is not covered by the current hypothesis.

Otherwise, we have to add more than one term to $h$ to cover all the points in $P$, and we begin a search for a set of terms to add. This search may disclose more positive

```
OnlyLMQ()
{
    h = "the empty formula"
    For (ever)
    {
        q = FindPos(h)
        If (q == "no")
        {
            Output h
            Return
        }
        (T, P) = Delimit(q)
        Delimited = {q}
        Terms = T
        Pos = {r ∈ P : h(r) == 0}
        For (j = 1;    ; j++)
            If (∃t_1, t_2, ..., t_j ∈ Terms s.t. ∀p ∈ Pos  ((t_1 ∨ ... ∨ t_j)(p) == 1))
            {
                Add disjunction (t_1 ∨ ... ∨ t_j) to h
                Break
            }
            Else
            {
                NewPos = ∅
                For (each p ∈ Pos s.t. p ∉ Delimited)
                {
                    (T, P) = Delimit(p)
                    Delimited = Delimited ∪ {p}
                    NewPos = NewPos ∪ {r ∈ P : h(r) == 0}
                    Terms = Terms ∪ T
                }
                Pos = Pos ∪ NewPos
            }
    }
}
```

Figure 7: Algorithm OnlyLMQ, which learns monotone $k$-term DNF from LMQ's.

points, so this process itself may have to be iterated. We begin by initializing *Terms* to be $T$ and *Pos* to be those points in $P$ that are not covered by $h$. We remember that point $q$ has already been delimited by placing it in set *Delimited*. We then call Algorithm DELIMIT on every point in *Pos* and for each call gather together the points from $P$ in *NewPos* as well as add the terms from $T$ to *Terms*. When all points in *Pos* have been delimited, we add the points gathered in *NewPos* to *Pos* and try to cover the newly enlarged set *Pos* with any $j = 2$ terms from *Terms*. If we succeed, we put these terms in $h$; if we fail, we keep enlarging *Terms*, gathering points in *NewPos*, then enlarging *Pos*, incrementing $j$, and trying again in the same manner to cover all points in *Pos* with $j$ terms from *Terms*.

**Theorem 3** *Algorithm* ONLYLMQ *learns monotone $k$-term DNF from $O(k(n^k + n^2\ell))$ limited membership queries in the nonstrict model, where $\ell$ is the number of $\perp$ responses it receives.*

**Proof:** All the terms added to the hypothesis are at some point in a set $T$ output by Algorithm DELIMIT. Therefore, by point 2 of Lemma 1, no term that we ever add to our hypothesis can err by classifying a point with LMQ 0 as positive.

Furthermore, since point 1 of Lemma 1 guarantees that each time we call the subroutine DELIMIT from a point we get at least one term in $T$ covering that point, the algorithm must eventually succeed in covering the set *Pos* with some number of terms and break out of the "**For** ($j = 1;\quad ; j$++)" loop. Since there are only a finite number of positive points, this means that the algorithm eventually terminates with a hypothesis that correctly classifies all non-$\perp$ points.

Thus the only thing we need to show for the correctness of the algorithm is that each hypothesis of Algorithm ONLYLMQ contains at most $k$ terms. The following lemmas provide the argument.

**Lemma 3** *No term in the set Terms is ever implied by the current hypothesis $h$ inside of the "**For** $j$" loop.*

**Proof:** The hypothesis $h$ is constant during the execution of the loop. Every element of *Terms* entered *Terms* from the set $T$ generated by calling Algorithm DELIMIT with this hypothesis $h$. The first call to DELIMIT (the one made immediately before entering the "**For** $j$" loop) is made on some point $q$ such that $h(q) = 0$, and therefore no element of $T$ can be implied by $h$. All subsequent calls to DELIMIT are made on some point $r \in Pos$. The set $Pos$ is constructed so that $h(r) = 0$, so again no points in the set $T$ returned by DELIMIT can be implied by $h$. ■

**Lemma 4** *Whenever Algorithm* ONLYLMQ *attempts to cover the set of positive points Pos with a disjunction of j terms from Terms (line "***If** $(\exists t_1, t_2, \ldots, t_j \in Terms$ *s.t.* $\forall p \in Pos \ ((t_1 \vee \ldots \vee t_j)(p) == 1))$*" in the code), the set Terms actually contains at least j distinct terms of the target concept.*

**Proof:** The proof is by induction on $j$. for the base case, $j = 1$, the set $Terms = T$. Thus the base case is provided by point 1 of Lemma 1, which says that there must must be a term of the target concept in $T$ at the end of the first call to Algorithm DELIMIT.

For the inductive step, suppose that we are trying to cover $Pos$ with $j + 1$ terms. Then we know that we tried and failed to cover the previous version of $Pos$ with $j$ terms. By the inductive hypothesis, this implies that the previous version of $Terms$ contained at least $j$ terms of the target concept. If the previous version of $Terms$ in fact contained more than $j$ terms of the target concept, then we are done.

Otherwise, the previous version of $Terms$ contained exactly $j$ terms of the target concept, but nevertheless failed to cover all points in $Pos$. That is, there was some point $p$ in the previous version of $Pos$ not covered by any of those particular $j$ terms. This point $p$ cannot have been delimited before the attempt to cover $Pos$ with $j$ terms, because if it had, the previous version of $Terms$ would have contained an element covering $p$. Therefore, after attempting and failing to cover $Pos$ with $j$ terms $p$ is delimited. Now by point 1 of Lemma 1, a term of the target concept covering $p$ is in the set $T$ output by this call to DELIMIT, and this point is added to $Terms$. Thus $Terms$ now contains at least $j + 1$ terms of the target concept. ■

**Lemma 5** *Let $h'$ be the hypothesis that Algorithm* ONLYLMQ *obtains when it updates its hypothesis $h$. Then $h'$ covers all points with* LMQ *1 that are covered by any term in Terms.*

**Proof:** Assume the contrary holds for $h$, $Terms$, and $h'$. Let $t \in Terms$ be a term and $x$ be a point with LMQ 1 such that such that $h'(x) = 0$ and $t(x) = 1$.

Term $t$ was added to $Terms$ by some call to subroutine DELIMIT. At that call, according to point 3 of Lemma 1, the set $P$ must have contained a point $p$ such that $t$ covered $p$ and $p$ is a descendant of $x$. Since $h'(x) = 0$, we know $h(x) = 0$, so $h(p) = 0$.

Therefore, after the call to Algorithm DELIMIT when the outputs had $t \in T$ and $p \in P$, point $p$ was added to *Pos*. Thus since $h'$ is satisfied by all points in *Pos*, it must be that $h'(p) = 1$ and thus $h'(x) = 1$, a contradiction. ∎

**Lemma 6** *Consider the hypothesis $h$ of Algorithm* ONLYLMQ *at any point in the run of it for a monotone DNF target concept $c$. There is a monotone DNF $d$ consisting of at least $\#(h)$ distinct terms from $c$ such that $\{v : d(v) = \mathrm{LMQ}(v) = 1\} \subseteq \{v : h(v) = 1\}$.*

**Proof:** The proof is by induction on the number times that $h$ is enlarged by adding new terms. The base case with an empty $h$ holds trivially. Let us now assume that the lemma holds up to the $k$-th time we add new terms to $h$, and that it does not hold the $k+1$-st time. Let us denote $h$ after these additions of terms by $h_k$ and $h_{k+1}$, respectively. That is:

(1) there are $\#(h_k)$ terms $t_1, t_2, \ldots, t_{\#(h_k)}$ in $c$ such that they cover a subset of those points with LMQ 1 that are covered by $h_k$,

(2) for any way we take $\#(h_{k+1})$ terms from $c$, there always is some point $w$ with LMQ 1 which is covered by these terms but not by $h_{k+1}$.

Furthermore, let us assume that exactly $j$ terms were added to $h_k$ when making $h_{k+1}$, that is, $\#(h_{k+1}) - \#(h_k) = j$. At the moment these $j$ terms were added, Lemma 4 guaranteed the existence of $j$ terms $t'_1, t'_2, \ldots, t'_j$ from $c$ in *Terms*. By Lemma 3 none of $t'_1, t'_2, \ldots, t'_j$ was implied by $h_k$. Now suppose we take terms $t_1, t_2, \ldots, t_{\#(h_k)}, t'_1, t'_2, \ldots, t'_j$, which are all in $c$, for a total of $\#(h_{k+1})$ terms. By our assumption there is a point $w$, such that $\mathrm{LMQ}(w) = 1$, that is covered by these terms but not by $h_{k+1}$. By the assumption, $w$ cannot be covered by any of the $t_1, t_2, \ldots, t_{\#(h_k)}$, or it would be covered by $h_k$ and hence $h_{k+1}$. Therefore, it must be the case that $w$ is covered by some of $t'_1, t'_2, \ldots, t'_j$. But then, by Lemma 5, $h_{k+1}$ must cover $w$, a contradiction. ∎

Lemma 6 implies that the hypothesis $h$ of Algorithm ONLYLMQ never contains more than $k$ terms, which is what we needed to show. What remains for the proof of Theorem 3 to be complete is the bound on the number of queries.

A monotone $k$-term DNF formula can have at most $n^k$ maximal negative points. This follows from the following observations. If $v$ is a negative point and $t$ is a term, then there must be a variable $x_i$ in $t$ such that $v[i] = 0$, or else $v$ would satisfy $t$. Thus, by setting to 0 at least one variable from each term, we can obtain exactly the negative points. Maximal negative points are a subset of the points that can be obtained by setting exactly one variable from each term to 0, which can be done in at most $n^k$ ways. Thus, each time $S$ is initialized in subroutine FINDPOS, it can have at most $n^k$ elements.

Also, FINDPOS is called at most $k + 1$ times, since each time except for the last one, it returns some positive point $q$, which is delimited and causes at least one term to be added to $h$. By Lemma 2, the number of queries made by Algorithm ONLYLMQ is at most $k(n^k + n(l + s))$, where $s$ is the number of times subroutine DELIMIT is called. For every $\perp$ there can be at most $n$ points added to *Pos*, so $s$ is bounded by $nl$. The total bound on the number of queries follows. ∎

The running time of the algorithm ONLYLMQ is clearly polynomial in the number of queries it makes.

### 3.4 Learning Monotone DNF from Equivalence and Limited Membership Queries

In this subsection, we give a very simple algorithm that learns monotone DNF with an unbounded number of terms from equivalence and limited membership queries.

**Theorem 4** *Monotone DNF formulas can be learned from $n(m+\ell)$ limited membership queries together with $m+1$ equivalence queries in the nonstrict model, or together with $m + \ell + 1$ equivalence queries in the strict model, where $\ell$ is the number of $\perp$ responses received, and $m$ is the number of terms in the target monotone DNF.*

**Proof:** We modify Angluin's algorithm for learning monotone DNF from ordinary membership and equivalence queries [3], by using Algorithm DELIMIT. The complete code of the algorithm is given in Figure 8. Note that the same algorithm works for both the standard and strict models.

The UP subroutine of Algorithm DELIMIT guarantees that all terms we put in $h$ are either correct, or err only by classifying negative points with LMQ $\perp$ as positive.

```
EQToo()
{
    h = "the empty formula"
    While ((v = EQ(h)) ≠ "yes")
        If (h(v) == 0)
        {
            (T, P) = Delimit(v)
            Add terms of T to h
        }
        Else
            For (each term t of h)
                If (t(v) == 1)
                    Delete term t from h
    Output h
}
```

Figure 8: Algorithm EQToo for learning monotone DNF from EQ's and LMQ's.

Lemma 1 guarantees that each time we call Algorithm Delimit we get a new term of the target monotone DNF.

In the nonstrict model, possible "negative $\perp$ becomes positive" errors will not matter, so we will have at most one call to Algorithm Delimit and one equivalence query per term. In the strict model there may be such $\perp$ terms that have to be deleted. We will need one extra equivalence query for each such term, provided that we modify the algorithm to remember such deleted terms and never add them again to the hypothesis. This implies the bounds for the number of equivalence queries. The bound for the number of LMQ's then follows from Lemma 2. ■

The running time of the algorithm is polynomial in the number of queries it makes.

Note that polynomial learnability of monotone DNF from equivalence and limited membership queries is implied by the stronger result of Section 4. It is at least as difficult to learn from a malicious membership oracle as it is from a limited membership oracle, as pointed out in Subsection 2.3, so that algorithm for monotone DNF could be used here. The direct algorithm does, however, give better bounds on the number of queries required.

# 4 Malicious Membership Queries

In this section, we present and analyze an algorithm that uses equivalence and malicious membership queries to learn monotone DNF formulas. The key idea is to depend on equivalence queries as much as possible, since they are correct.

## 4.1 The Algorithm

The algorithm keeps track of all the counterexamples and their labels received through equivalence queries and consults them first, before asking a membership query. The pairs of counterexamples and their labels are kept in a set named *CounterExamples*. Obviously, for a positive counterexample $v$, if $x \geq v$ then it is not worth making a membership query about $x$; it must be a positive point. Similarly, for a negative counterexample $v$, if $x \leq v$ then $x$ has to be a negative point of the target formula. For this reason we define a subroutine CheckedMQ and use it instead of a membership query. The subroutine is given in Figure 9.

```
CheckedMQ(x)
{
    If (∃(v,1) ∈ CounterExamples s.t. x ≥ v)
        Return 1
    If (∃(v,0) ∈ CounterExamples s.t. x ≤ v)
        Return 0
    Return MMQ(x)
}
```

Figure 9: Subroutine CheckedMQ.

As in [3] and [7], our algorithm also uses a subroutine Reduce in order to move down in the lattice from a positive counterexample. All the membership queries are done using the subroutine CheckedMQ, which possibly lets the algorithm avoid some incorrect answers. The subroutine Reduce is given in Figure 10.

The algorithm for exactly identifying monotone DNF formulas using equivalence queries and malicious membership queries is given in Figure 11.

The algorithm is based on a few simple ideas. A positive counterexample is reduced to a point that is added as a term to the existing hypothesis $h$, which is a

```
REDUCE(v)
{
    For (each child w of v)
        If (CHECKEDMQ(w) == 1)
            Return REDUCE(w)
    Return v
}
```

Figure 10: Subroutine REDUCE.

monotone DNF. That is, the new hypothesis will classify the latest counterexample and possibly some other points as positive.

Negative counterexamples are used to detect inconsistencies between membership and equivalence queries. They show that there have been errors in membership queries that have caused wrong terms to be added to the hypothesis. The algorithm reacts by removing all the terms that are inconsistent with the latest counterexample. These are the terms that have the negative counterexample above them. A term can be removed only when there is a negative counterexample above it.

## 4.2 Analysis of LEARNMONDNF

**Theorem 5** LEARNMONDNF *learns the class of monotone DNF formulas in polynomial time using equivalence and malicious membership queries.*

We need a definition and a simple lemma before proving the theorem.

Let $h^*$ be a monotone boolean function on $\{0,1\}^n$, and let $h'$ be an arbitrary boolean function on $\{0,1\}^n$. Let $C$ be any subset of $\{0,1\}^n$. The *monotone correction of* $h'$ *with* $h^*$ *on* $C$, denoted $mc(h', h^*, C)$, is the boolean function $h''$ defined for each string $x \in \{0,1\}^n$ as follows.

$$h''(x) \stackrel{\text{def}}{=} \begin{cases} 1 & \text{if there exists } y \in C \text{ such that } y \leq x \text{ and } h^*(y) = 1, \\ 0 & \text{if there exists } y \in C \text{ such that } x \leq y \text{ and } h^*(y) = 0, \\ h'(x) & \text{otherwise.} \end{cases}$$

Note that since $h^*$ is monotone, the first two cases above cannot hold simultaneously. It is also clear that if the value of $h''(x)$ is determined by one of the first

```
LearnMonDNF()
{
    CounterExamples = ∅
    h = "the empty formula"
    While ((v = EQ(h)) ≠ "yes")
    {
        Add (v, 1 − h(v)) to CounterExamples
        If (h(v) == 0)
        {
            w = Reduce(v)
            Add term w to h
        }
        Else
            For (each term t of h)
                If (t(v) == 1)
                    Delete term t from h
    }
    Output h
}
```

Figure 11: The algorithm for learning monotone DNF from EQ's and MMQ's.

two cases, $h''(x) = h^*(x)$. We prove a simple monotonicity property of the monotone correction operation.

**Lemma 7** *Suppose $h^*$ is a monotone boolean function and $h'$ is an arbitrary boolean function on $\{0,1\}^n$. Let $C_1 \subseteq C_2$ be two subsets of $\{0,1\}^n$. Let $h_1 = mc(h', h^*, C_1)$ and $h_2 = mc(h', h^*, C_2)$. Then the set of points on which $h_2$ and $h^*$ differ is contained in the set of points on which $h_1$ and $h^*$ differ. That is, $Err(h_2, h^*) \subseteq Err(h_1, h^*)$.*

**Proof:** Let $x$ be an arbitrary point on which $h_2(x) \neq h^*(x)$. Then it must be that $h_2(x) = h'(x)$ and there does not exist any point $y \in C_2$ such that $x \leq y$ and $h^*(x) = 0$ or $y \leq x$ and $h_*(x) = 1$. Since $C_1$ is contained in $C_2$, there is no point $y \in C_1$ such that $x \leq y$ and $h^*(x) = 0$ or such that $y \leq x$ and $h_*(x) = 1$. Thus, $h_1(x) = h'(x)$ and $h_1(x) \neq h^*(x)$. Consequently, $Err(h_2, h^*) \subseteq Err(h_1, h^*)$. ■

Now we start the proof of Theorem 5.

**Proof:** Let $h^*$ denote the target concept, an arbitrary monotone DNF formula over $\{0,1\}^n$ with $m$ terms. Let $\ell$ be a bound on the number of strings whose MMQ's are answered incorrectly. Because equivalence queries are answered correctly, if the algorithm ever halts, the hypothesis output is correct, so we may focus on proving a polynomial bound on the running time.

Since LearnMonDNF is deterministic and the target concept $h^*$ is fixed, we may assume that the adversary chooses in advance how to answer all the queries, that is, chooses a sequence $y_1, y_2, \ldots$ of counterexamples to equivalence queries and a set $S$ of strings on which to answer MMQ's incorrectly. Note that $|S| \leq \ell$.

In turn, these choices determine a particular computation of LearnMonDNF which we now focus on. It suffices to bound the length of this computation. In this computation the answers to MMQ's agree with the boolean function $h_0 = xcpt(h^*, S)$. Also, if CheckedMQ is called at step $t$ of this computation on the string $x$, the answer agrees with the boolean function $mc(h_0, h^*, C)$, where $C$ is set of counterexamples received up to step $t$.

The set *CounterExamples* only changes when a new counterexample is received. Therefore, the successive distinct sets of counterexamples in this computation can be denoted by $C_0, C_1, \ldots$, where $C_0 = \emptyset$ and $C_i = C_{i-1} \cup \{y_i\}$, for $i = 1, 2, \ldots$. If we also define

$$h_i = mc(h_0, h^*, C_i)$$

for $i = 1, 2, \ldots$, then CheckedMQ answers according to $h_0$ until the first counterexample is received, then according to $h_1$ until the second counterexample is received, and so on.

Clearly, since $h_0$ disagrees with $h^*$ on at most $\ell$ strings, $d(h_0, h^*) \leq \ell$. Since the sets $C_0, C_1, \ldots$ are monotonically nondecreasing, Lemma 7 shows that $Err(h_i, h^*) \subseteq Err(h_{i-1}, h^*)$ for $i = 1, 2, \ldots$.

We say that a counterexample $y_i$ *corrects a positive error* at point $x$ if $h_{i-1}(x) = 1$ but $h_i(x) = h^*(x) = 0$. We say that a counterexample $y_i$ *corrects a negative error* at point $x$ if $h_{i-1}(x) = 0$ but $h_i(x) = h^*(x) = 1$. Note that from the construction of CheckedMQ it follows that positive errors can be corrected only by negative counterexamples and negative errors can be corrected only by positive counterexamples. Let there be $\ell_p$ positive and $\ell_n$ negative errors corrected in the whole computation. Of course, $\ell_p + \ell_n \leq \ell$.

**Claim 1** *If* Reduce *is called after counterexample $y_i$ and before counterexample*

$y_{i+1}$, *it returns a local minimum point of* $h_i$.

**Proof:** After $y_i$ is added to *CounterExamples*, CHECKEDMQ answers according to $h_i$. The claim follows from the construction of REDUCE. ∎

**Claim 2** *Condition preserved: at the* $(i+1)$*th equivalence query* EQ$(h)$*, each term of* $h$ *is a positive point of* $h_i$.

**Proof:** We prove the claim by induction.

> **Basis:** The first EQ is made on an empty formula. Thus, the claim is vacuously true.
>
> **Induction step:** Suppose the claim is true up to the $i$th EQ. Let $h'$ be the hypothesis $h$ at the $i$th EQ and $h''$ be the hypothesis $h$ at the $(i+1)$th EQ. There are two cases to consider.
>
> > **Case 1:** $y_i$ is a positive counterexample. Then $h_i(x) = 1$ if and only if $h_{i-1}(x) = 1$ or $x \geq y_i$. Let $t$ be the term returned by REDUCE$(y_i)$. Then $h'' = h' \vee t$. Let $t''$ be a term in $h''$. Then either $t''$ is a term of $h'$ or $t'' = t$. If $t''$ is a term of $h'$ then $h_{i-1}(t'') = 1$ by the inductive assumption and therefore $h_i(t'') = 1$. If $t'' = t$ then $h_i(t'') = 1$ since $t$ was returned by REDUCE$(y_i)$ which used CHECKEDMQ, which answered according to $h_i$.
> >
> > **Case 2:** $y_i$ is a negative counterexample. Then $h_i(x) = 1$ if and only if $h_{i-1}(x) = 1$ and $x \not\leq y_i$. Let $t''$ be a term in $h''$, which consists of all those terms $t'$ of $h'$ such that $t' \not\leq y_i$. Therefore, $t'' \not\leq y_i$ and by the inductive assumption $h_{i-1}(t'') = 1$. It follows that $h_i(t'') = 1$. ∎

**Claim 3** *Once a term* $x$ *is deleted from hypothesis* $h$*, it can never reappear in it.*

**Proof:** Since $x$ was deleted, there must have been a negative counterexample $y_i$ such that $y_i \geq x$. But then $(y_i, 0)$ belongs to *CounterExamples* and CHECKEDMQ$(x)$ can never return 1 again, which is necessary for $x$ to be added to $h$. ∎

We divide the run of the algorithm into non-overlapping *stages*. A new stage begins either at the beginning of the run or with a new negative counterexample. Thus with each new stage *CounterExamples* contains one more negative counterexample and some (possibly none) new positive counterexamples. The following claim establishes that the distance $d(h_i, h^*)$ decreases with every new stage.

**Claim 4** *Every negative counterexample corrects at least one error. More formally, if $y_i$ is a negative counterexample, then there exists $x \in \{0,1\}^n$ such that $h_{i-1}(x) = 1$ and $h_i(x) = h^*(x) = 0$.*

**Proof:** Let $y_i$ be a negative counterexample returned by EQ$(h)$. Hence $h(y_i) = 1$, and there is some term $x \leq y_i$ in $h$. By Claim 2, $h_{i-1}(x) = 1$.

Since $h^*(y_i) = 0$ and $y_i \geq x$ it follows that $h^*(x) = 0$. By the definition of $h_i$ it follows that $h_i(x) = 0$. ∎

From Claim 4 it follows that there are at most $\ell_p$ negative counterexamples. Hence there are at most $\ell_p + 1$ stages in the run of the algorithm.

We divide each stage of the algorithm into non-overlapping *substages*. A substage begins either at the beginning of a stage or with a new positive counterexample that corrects an error. Obviously there can be no more than $\ell_n$ positive counterexamples that correct errors and hence no more than $\ell_p + \ell_n + 1$ substages in the whole run of the algorithm. The distance $d(h_i, h^*)$ decreases with every new substage. If, however, functions $h_i$ and $h_j$ belong to the same substage, they are equivalent and their local minima are the same. This allows us to bound the total number of positive counterexamples.

**Claim 5** *Every new positive counterexample is reduced to a local minimum point of $h_0, h_1, \ldots$ that has not been found earlier.*

**Proof:** Let $v$ be a positive counterexample that REDUCE is started with. Let $t$ be the point REDUCE$(v)$ returns. Assume, by way of contradiction, that $t$ has already been found before. From Claim 3 it follows that $t$ is a term in $h$. Since $v \geq t$, it follows that $h(v) = 1$. This is a contradiction to the assumption that $v$ is a positive counterexample. ∎

We denote the set of local minimum points of a boolean function $f$ by *Lmp*$(f)$. We bound the total number of different local minima of the functions $h_0, h_1, \ldots$.

**Lemma 8** *Let $f$ and $f'$ be $n$-argument boolean functions such that $Err(f, f') = \{x\}$. Then*

*(a) If $f'(x) = 1$ then $|Lmp(f') - Lmp(f)| \leq 1$.*

*(b) If $f'(x) = 0$ then $|Lmp(f') - Lmp(f)| \leq n$.*

**Proof:**

(a) The only point that can be a local minimum of $f'$ and is not a local minimum of $f$, is $x$ itself. The claim follows immediately.

(b) Any point which is a local minimum of $f'$ but not of $f$ is a parent of $x$. Since $x$ has at most $n$ parents, the claim follows. ■

**Corollary 2** *Let $f$ and $f'$ be $n$-argument boolean functions such that $Err(f, f')$ contains $d_p$ positive points of $f'$ and $d_n$ negative points of $f'$. Then*

$$|Lmp(f') - Lmp(f)| \leq nd_n + d_p.$$

**Corollary 3** *Let $g_0, g_1, \ldots, g_r$ be the subsequence of $h_0, h_1, \ldots$, such that each $g_i$ is the first of all the $h_j$'s in its substage. Let $Err(h^*, g_{i-1}) - Err(h^*, g_i)$ contain $\ell_{p,i-1}$ positive and $\ell_{n,i-1}$ negative points of $h^*$ for all $i = 1, 2, \ldots, r$. Let $Err(h^*, g_r)$ contain $\ell_{p,r}$ positive and $\ell_{n,r}$ negative points of $h^*$. Then the total number of different local minima of functions $g_0, g_1, \ldots, g_r, h^*$ is bounded above by $m + n\sum_{i=0}^{r} \ell_{n,i} + \sum_{i=0}^{r} \ell_{p,i}$.*

**Proof:** Note that $g_0, g_1, \ldots, g_r$ are the different functions in $h_0, h_1, \ldots$, and that CheckedMQ first answers according to $g_0$, then according to $g_1$ and so on. Obviously, $Err(h^*, g_i) \subseteq Err(h^*, g_{i-1})$ and $Err(g_{i-1}, g_i) = Err(h^*, g_{i-1}) - Err(h^*, g_i)$ for all $i = 1, 2, \ldots, r$. Also note that for each $i = 0, 1, \ldots, r-1$, one of $\ell_{p,i}$ and $\ell_{n,i}$ is 0, but $\ell_{p,r}$ and $\ell_{n,r}$ may both be positive.

We want to find $|\bigcup_{i=0}^{r} Lmp(g_i) \cup Lmp(h^*)|$, knowing that $|Lmp(h^*)| = m$. Since

$$\bigcup_{i=0}^{r} Lmp(g_i) \cup Lmp(h^*) \subseteq Lmp(h^*) \cup (Lmp(g_r) - Lmp(h^*)) \cup \bigcup_{i=0}^{r-1} (Lmp(g_i) - Lmp(g_{i+1})),$$

from Corollary 2 it follows that

$$|\bigcup_{i=0}^{r} Lmp(g_i) \cup Lmp(h^*)| \leq |Lmp(h^*)| + (n\ell_{n,r} + \ell_{p,r}) + \sum_{i=0}^{r-1}(n\ell_{n,i} + \ell_{p,i})$$

and the bound follows. ■

Since each error can be corrected at most once, it follows that $\sum_{i=0}^{r} \ell_{n,i} \leq \ell_n$ and $\sum_{i=0}^{r} \ell_{p,i} \leq \ell_p$. Hence the total number of the local minima and the total number of positive counterexamples that can be found in a computation is bounded by $m + n\ell_n + \ell_p$. The number of negative counterexamples in a complete run is bounded by the number of positive errors. The total number of counterexamples is therefore bounded by $m + \ell_n n + \ell_p + \ell_p \leq m + \ell(n+1) = O(m + \ell n)$.

We now count the number of membership queries in a complete run of the algorithm. Each positive counterexample $v$ may cause at most $n(n+1)/2$ membership queries, before REDUCE($v$) returns. Therefore there can be at most $O(mn^2 + \ell n^3)$ membership queries in a complete run of the algorithm.

It is also clear that the running time of the algorithm is polynomial in $m$, $n$ and $\ell$. This concludes the proof of Theorem 5. ■

# 5 Finite Exceptions

The relevant definitions for this section were introduced in Subsection 2.2. We start the section with a few examples and lemmas about finite exceptions in general, and then give an algorithm for learning the class of monotone DNF formulas with finite exceptions and analyze it.

## 5.1 Examples and Lemmas

**Example 1** The class of regular languages represented by DFA's is polynomially closed under finite exceptions. Board and Pitt give an algorithm that takes as input a DFA $M$ and an exception set $S$, and produces a new DFA for $xcpt(M, S)$ [10]. The DFA's size is polynomial in the size of $M$ and $S$.

**Example 2** Another example of a class that is polynomially closed under finite exceptions is the class of boolean decision trees. This result is taken from [10] but since the construction is not given there, we sketch it here.

**Lemma 9** *The class of boolean decision trees is polynomially closed under finite exceptions.*

**Proof:** Let $T$ be a decision tree on $n$ variables. Let $S$ be the exception set for $T$. We construct the decision tree for *xcpt*$(T, S)$ as follows. We treat each exception point $x \in S$ individually. First we walk down from the root of the original tree $T$ to see where $x$ is located in it. If this leads us to a leaf with depth $n$ (i.e., if all variables are tested on this path), then we just reverse the value of the leaf, because this path is for $x$ only. However, if we find ourselves at a leaf with depth less than $n$, we have to add new internal nodes to the tree. Denote the value of this leaf by $b$. We then continue the path that led us to this leaf with a path in which all the remaining variables are tested. We end the path by a leaf with value $1 - b$. For each new internal node on the path, we make the other child (the one not on the path) a leaf, and give it the original value $b$. Thus, each counterexample adds at most $n$ new internal nodes to the tree. The size of the new tree, measured as the number of internal nodes, is bounded by $|T| + n \times |S| = |T| + ||S||$. ■

**Example 3** One more interesting example is the class of DNF formulas.

**Lemma 10** *The class of DNF formulas is polynomially closed under finite exceptions.*

**Proof:** Let $f$ be an $m$-term DNF formula over $n$ variables and $S$ be an exception set for it. Let $S$ be partitioned into the sets of positive and negative exceptions ($S_+$ and $S_-$, respectively), as described in Section 2.2. We construct a DNF formula for *xcpt*$(f, S)$ from the formula $(f \wedge f_-) \vee f_+$, where $f_-$ is a DNF formula which is true on all the points in its domain except the ones in $S_-$, and $f_+$ is a DNF formula which is true exactly on the points in $S_+$. The domain for all these formulas is $\{0, 1\}^n$.

Obtaining $f_+$ is easy — straightforward disjunction of all the *terms* in $S_+$, where we make terms from points by substituting the respective variable for a 1 value of a coordinate and its negation for a 0 value. Obtaining $f_-$ is harder. First we make a decision tree corresponding to $f_-$. We put each point from $S_-$ individually in the tree as a 0-valued leaf at the end of a path of length $n$. All the remaining leaves get value 1. Then for each leaf with value 1 we make a term that will go into $f_-$ by following the path from this leaf to the root. Obviously $f_-$ has at most $n \times |S_-|$ terms. Thus, after "multiplying" the terms out, the formula $(f \wedge f_-) \vee f_+$ will have at most $mn \times |S_-| + |S_+| \leq (mn+1) \times |S|$ terms. ∎

**Example 4** By duality it follows that the class of CNF formulas is polynomially closed under finite exceptions.

Note that stronger bounds on the size of the new formula can be obtained by using the result in [25]. We, however, chose to present a simpler argument. Also note that the size bound is insufficient for *strong polynomial closure under exception lists* as defined in [10].

**Example 5** As our final example we show that any class that is obtained by adding exception tables to another class is polynomially closed under finite exceptions.

**Lemma 11** *Let $(R, Dom, \mu)$ be any class of concepts. Then the concept class obtained from it by adding exception tables is polynomially closed under finite exceptions.*

**Proof:** Let $(R', Dom', \mu')$ be the class obtained from $(R, Dom, \mu)$ by adding exception tables, as defined in Section 2.2. Let $(X', f')$ be any concept from $(R', Dom', \mu')$ and let $r' \in R'$ be a shortest representation of $(X', f')$. Then there exists a concept $r \in R$ and a finite set $S \subseteq Dom(r)$, such that $(Dom'(r'), \mu'(r')) = xcpt((Dom(r), \mu(r)), S)$ and $|r'| = 2(1 + |r| + ||S||)$. Let $S' \subseteq Dom'(r') = Dom(r)$ be any finite set. Let concept $h''$ be defined as $h'' \stackrel{\text{def}}{=} xcpt((Dom'(r'), \mu'(r')), S')$. It is easy to see that $h'' = xcpt((Dom(r), \mu(r)), S \triangle S')$ and thus $h''$ is represented by some $r'' \in R'$ with size $2(1 + |r| + ||S \triangle S'||) \leq 2(1 + |r| + ||S|| + ||S'||) = |r'| + 2||S'||$. ∎

**Corollary 4** *The class of monotone DNF formulas with finite exceptions is polynomially closed under finite exceptions.*

## 5.2 Learning Monotone DNF Formulas With Finite Exceptions

In this section, we present an algorithm that learns the class of monotone DNF formulas with finite exceptions. The target concept is a boolean function on $n$ variables

$h^* \stackrel{\text{def}}{=} xcpt(h^*_M, S^*)$, where $h^*_M$ is some monotone DNF formula and $S^*$ is a set of exceptions for it. The domain of the target concept is $\{0,1\}^n$.

We assume that we have an upper bound on the cardinality of $S^*$ and denote it by $l$ (i.e., $|S^*| \leq l$). If this bound is not known, we can start out by assuming it to be any positive integer and doubling it whenever convergence is not achieved within the proper time bound, which will be given later. We assume that $h^*_M$ is minimized and has $m$ terms.

Like LearnMonDNF, our current algorithm also has a set *CounterExamples* that stores all labeled counterexamples received from equivalence queries. The purpose of it is slightly different: it lets the algorithm conclude that some points cannot be classified by $h^*_M$ alone, and, therefore, have to be included in the exception set.

The algorithm tries to find a suitable monotone DNF formula, which, coupled with a proper exception set, would give the target concept. The equivalence queries are made on a pair $(h, S)$ of a monotone DNF formula $h$ and a set of exceptions $S$. The algorithm focuses only on building $h$, and sets $S$ to be those elements of the set *CounterExamples* that are currently misclassified by $h$. It uses a simple subroutine GetExceptions for building $S$. The subroutine is given in Figure 12.

```
GetExceptions()
{
    S = ∅
    For (each (x, b) ∈ CounterExamples)
        If (h(x) ≠ b)
            Add x to S
    Return S
}
```

Figure 12: Subroutine GetExceptions.

In order to classify the counterexamples received, the algorithm needs to evaluate the current function $xcpt(h, S)$. This is done by another very simple subroutine TheFunction, given in Figure 13.

As in [3], [7], and Section 4, our algorithm also uses a subroutine Reduce to move down in the lattice from a positive counterexample. Its goal is to reduce the positive counterexample to some point that can be added as a term to the formula $h$. Then the new hypothesis would classify the counterexample and possibly some other points as positive. However, this may not always be possible. There can be overwhelming

```
TheFunction(x)
{
    If (x ∈ S)
        Return 1 − h(x)
    Else
        Return h(x)
}
```

Figure 13: Subroutine TheFunction.

evidence that the candidate point is just a positive exception and thus should not be added to $h$. More precisely, if there are more than $l$ negative counterexamples above a term of $h$, then they all have to be in the exception set, which is then too big. Therefore the current subroutine Reduce is somewhat more complex and checks whether a point has enough evidence to be an undoubted exception point or not. The subroutine is given in Figure 14.

```
Reduce(v)
{
    For (each child w of v)
        If ((MQ(w) == 1)  &&  (|{y ≥ w | (y, 0) ∈ CounterExamples}| ≤ l))
            Return Reduce(w)
    Return v
}
```

Figure 14: Subroutine Reduce.

The algorithm for learning monotone DNF formulas with at most $l$ exceptions using equivalence queries and membership queries is given in Figure 15.

The algorithm is based on the following ideas. Each positive counterexample is reduced if possible to a new term to be added to the formula, as was explained above. In case this is not possible, the algorithm benefits anyway by storing it in the set *CounterExamples*.

Negative counterexamples imply that there are not as many positive points in the target concept as we thought. Sometimes more exception points are necessary for the hypothesis to be correct. Other times some terms have to be removed from the formula. Deleting a term happens only when there is enough evidence that a term is wrong, namely, when there are more than $l$ negative counterexamples above it.

```
LearnMonDNFwithFX()
{
    S = CounterExamples = ∅
    h = "the empty formula"
    While ((v = EQ((h, S))) ≠ "yes")
    {
        Add (v, 1 − TheFunction(v)) to CounterExamples
        If (TheFunction(v) == 1)
            For (each term t of h)
                If (|{w ≥ t | (w, 0) ∈ CounterExamples}| > l)
                    Delete term t from h
        For (each (x, 1) ∈ CounterExamples)
            If ((h(x) == 0)  &&  (|{y ≥ x | (y, 0) ∈ CounterExamples}| ≤ l))
            {
                w = Reduce(x)
                Add term w to h
            }
        S = GetExceptions()
    }
    Output (h, S)
}
```

Figure 15: The algorithm for learning monotone DNF formulas with finite exceptions.

## 5.3 Correctness and Complexity of the Algorithm

**Theorem 6** LearnMonDNFwithFX *learns the class of monotone DNF formulas with exceptions in polynomial time using equivalence and standard membership queries.*

**Proof:** We begin the analysis with this simple claim.

**Claim 6** *Once a term $t$ is deleted from hypothesis $h$, it can never reappear in it.*

**Proof:** A term $t$ can be deleted only if there are more than $l$ negative counterexamples above it. To reappear, $t$ must be returned by Reduce. But every point returned by Reduce must have at most $l$ negative counterexamples above it at the time it is returned, so Reduce cannot return $t$ again. ■

The following lemma shows what points REDUCE can return.

**Lemma 12** REDUCE *always returns either a local minimum of* $h^*$ *or a parent of a positive exception in* $S^*$.

**Proof:** First note that REDUCE can only be called on points $x$ such that $h^*(x) = 1$ and can only return points $w$ such that $h^*(w) = 1$. Let $w$ be a point returned by REDUCE. Assume $w$ is not a local minimum point of $h^*$. Then for some child $y$ of $w$, $h^*(y) = 1$ and the number of negative counterexamples above $y$ is greater than $l$. Hence, $y$ cannot be above any term $t$ of $h^*_M$, since each term $t$ can have at most $l$ negative counterexamples above it. Therefore, $y$ is a positive exception in $S^*$. ■

Now we are ready to bound the number of different points that can be returned by the subroutine REDUCE.

**Claim 7** *The number of different points that* REDUCE *can return is at most* $m + (n+1)l$.

**Proof:** Let set $S^*$ contain $l_p$ positive exceptions and $l_n$ negative exceptions, where $l_p + l_n \leq l$. The formula $h^*_M$ has $m$ terms and thus $m$ local minima. By Lemma 8, the number of local minima of $h^*$ is at most $m + l_p + nl_n$. Each positive exception can have at most $n$ parents which allows us to bound the number of parents of positive exceptions by $nl_p$. Therefore, by Lemma 12, the number of different points that REDUCE can return is at most $m + (n+1)l_p + nl_n \leq m + (n+1)l$. ■

All equivalence queries are asked about the current function $xcpt(h, S)$. Since $S$ is constructed right before a new equivalence query from counterexamples that are misclassified by $h$ alone, the argument of an equivalence query is always consistent with all the counterexamples seen to that point. Therefore the function $xcpt(h, S)$ is different for each equivalence query. This allows us to bound the number of equivalence queries.

**Claim 8** *The number of equivalence queries before success is bounded by* $O(m^2n^2l^3)$.

**Proof:** We examine how $xcpt(h, S)$ changes. Either $h$ itself changes, or $h$ remains the same and $S$ changes; namely, it contains exactly one more point, the most recent counterexample.

By Claim 6, each term of $h$ can appear in $h$ or disappear from it only once. Thus each possible term can induce at most two changes in formula $h$ — first by appearing in it and then by disappearing. Thus, $h$ can only change twice as many times as the number of terms that REDUCE can return. Therefore, by Claim 7, there can be at most $2(m + (n + 1)l) + 1$ different functions $h$ in a complete run of the algorithm.

We now count the number of times $S$ can change while $h$ remains the same. Set $S$ grows larger by one with each new counterexample. It contains some (possibly none) points $x$ such that $h(x) = 1$ and some (possibly none) points $x$ such that $h(x) = 0$. We bound the number of each of these separately.

Each point $x \in S$ such that $h(x) = 1$ is above some term of $h$. No term can have more than $l$ negative counterexamples above it. Therefore, the number of points $x \in S$ such that $h(x) = 1$ can be bounded by $l$ times the bound $m + (n + 1)l$ on the number of different terms of $h$, that is, by $ml + (n + 1)l^2$.

Each point $x \in S$ such that $h(x) = 0$ is a positive counterexample, and thus is not above any term in $h$. Such an $x$ must have more than $l$ negative counterexamples above it. Otherwise, the algorithm would have called REDUCE on $x$ and added a new term $t \leq x$ to $h$. If $x$ has more than $l$ negative counterexamples above it, then it cannot be above a term in $h^*_M$ and thus has to be a positive exception in $S^*$. Hence we have a bound of $l_p$ on the number of points $x \in S$ such that $h(x) = 0$.

Altogether, we can bound the cardinality of $S$ by $|S| \leq ml + (n + 1)l^2 + l_p \leq (m + 1)l + (n + 1)l^2$. While $h$ stays the same, the number of possible different sets $S$ is at most $(m + 1)l + (n + 1)l^2 + 1$.

Hence, the total number of equivalence queries in a complete run of the algorithm is bounded by $(2(m + (n + 1)l) + 1) \times ((m + 1)l + (n + 1)l^2 + 1) = O(m^2n^2l^3)$. ∎

We now count the total number of membership queries. Each positive counterexample $v$ may cause at most $n(n + 1)/2$ membership queries, before REDUCE($v$) returns. Therefore there can be at most $O(m^2n^4l^3)$ membership queries in a complete run of the algorithm.

It is not difficult to see that the total running time of the algorithm is polynomial in $n$, $m$ and $l$. This concludes the proof of Theorem 6. ∎

# 6 Comparison of the Models

In this section, we compare the models of learning discussed earlier, and give a relation between learning concepts with exceptions and learning with malicious membership queries.

## 6.1 Exceptions and Lies

In this subsection, we give a generic algorithm transformation. This transformation shows that any class of concepts that is polynomially closed under finite exceptions and learnable in polynomial time with equivalence and standard membership queries is also learnable in polynomial time using equivalence and malicious membership queries.

**Theorem 7** *Let $H$ be a class of concepts that is polynomially closed under finite exceptions and learnable in polynomial time with equivalence and standard membership queries. Then $H$ is learnable in polynomial time with equivalence and malicious membership queries.*

**Proof:** Let $H = (R, Dom, \mu)$ be a target class of concepts that is polynomially closed under finite exceptions. We assume that LEARN is an algorithm to learn $H$ using equivalence (EQ) and standard membership queries (MQ) in time $p_A(s, n)$, for some polynomial $p_A$. Without loss of generality, $p_A$ is non-decreasing in both arguments. We transform this algorithm into algorithm LEARNANYWAY, which learns any concept $h^* \in H$ using equivalence and malicious membership queries in time polynomial in $|h^*|$, $n$ and the table-size $L$ of the set of strings on which MMQ may lie.

As in Sections 4 and 5.2 the main idea is to keep track of all the counterexamples seen and to use them to avoid unnecessary membership queries. For this purpose we use a set *CounterExamples* again. As before it stores pairs of counterexamples and their labels. Now, before asking a membership query about string $x$, we scan *CounterExamples* to see whether it already contains $x$ and a label for it. If $x$ and the label are found, the algorithm knows the answer and does not make the query. (For some concept classes, such as monotone DNF formulas, it might be possible to infer the classification of $x$ according to the target concept $h^*$ even though $x$ and its label

```
NewMQ(x)
{
    If ((x, b) ∈ CounterExamples)
        Return b
    b = MMQ(x)
    Add (x, b) to MembershipAnswers
    Return b
}
```

Figure 16: Subroutine NewMQ.

are not contained in *CounterExamples*. However, this simple checking suffices for our algorithm and, what is more important, works in the general case.)

Another idea is to keep track of the answers received from membership queries, and to use them to conclude that MMQ has lied. For this purpose LearnAnyway has a set *MembershipAnswers*. This set stores pairs $(x, b)$ for which MMQ was called on string $x$ and returned answer $b$. After receiving a new counterexample from EQ, the algorithm stores it in *CounterExamples* and checks whether this counterexample is already contained in *MembershipAnswers*. If it is present in *MembershipAnswers* with the wrong label, the algorithm discards everything except the set *CounterExamples* and starts from scratch. If this is not the case, the algorithm continues the simulation of Learn, which we now describe in detail.

The new algorithm simulates Learn on the target concept, but modifies Learn's queries as follows:

- Each membership query of Learn, MQ$(x)$, is replaced by a subroutine call NewMQ$(x)$. The subroutine is given in Figure 16.
- Each equivalence query of Learn, $x = \mathrm{EQ}(h)$, as well as the output statement, **Output** $h$, is replaced by the block of code given in Figure 17.

Note that when the simulation is restarted, only the set *CounterExamples* reflects any work done so far. We now show that LearnAnyway is correct and runs in time polynomial in $|h^*|$, $n$, and $L$. We partition the run of the algorithm into *stages*, where a stage begins with a new simulation of Learn. First we show that a stage cannot last forever.

**Claim 9** *Every stage ends in time polynomial in* $|h^*|$, $n$, *and* $L$.

```
{
    x = EQ(h)
    If (x == "yes")
    {
        Output h
        Return
    }
    Add (x, 1 − h(x)) to CounterExamples
    If ((x, h(x)) ∈ MembershipAnswers)
    {
        MembershipAnswers = ∅
        Restart Simulation, retaining CounterExamples
    }
}
```

Figure 17: The block of code replacing "$x = \mathrm{EQ}(h)$" or "**Output** $h$".

**Proof:** Note that $H$ is polynomially closed under finite exceptions, which means that there is a polynomial $p(\cdot,\cdot)$ such that for every concept $h \in H$ and every finite set $S \subseteq Dom(h)$ there exists a concept $h' \in H$ equal to $xcpt(h, S)$ such that size $|h'| \leq p(|h|, ||S||)$. Without loss of generality we can assume that $p$ is non-decreasing in both arguments. We now prove that each stage ends in time bounded by $p_A(p(|h^*|, L), n)$, where we count only the time spent on LEARN operations (i.e., we do not count the simulation and bookkeeping overhead).

We prove this by contradiction. Assume that stage $i$ goes over the limit. Let us look at the situation right after the number of simulated steps of LEARN exceeds our stated time bound. Let $S_i$ denote the set of strings the MMQ has lied about during this stage, up to the time bound. Let $n$ denote the length of the longest counterexample received during this stage, up to the time bound.

None of the strings in $S_i$ can belong to *CounterExamples*. Assume by way of contradiction otherwise. Let $x \in S_i$ be a string contained in *CounterExamples* with some label. Set $S_i$ contains exactly the strings that the MMQ lied on in this stage and time bound, so there was a query $\mathrm{MMQ}(x)$. It must have happened before $x$ was added to *CounterExamples*. But then at the moment $x$ was added to *CounterExamples* it already belonged to *MembershipAnswers* and an inconsistency had to be found. The stage had to end.

Therefore, considering $S_i$ as an exception set, all the information received by LEARN in this stage and within the given time bound is consistent with the concept $h' = xcpt(h^*, S_i) \in H$. LEARN either has to output $h'$ in time bounded by $p_L(p(|h^*|, ||S_i||), n) \leq p_L(p(|h^*|, L), n)$, or it has to receive a counterexample $x \in S_i$. In the former case, LEARNANYWAY makes an equivalence query $\mathrm{EQ}(h')$ and receives a counterexample $x \in S_i$, since only counterexamples from $S_i$ are possible at that point. In either case, an element of $S_i$ is added to *CounterExamples* by the above time bound, which we showed above was impossible. This is a contradiction to the assumption that stage $i$ goes over this bound. ■

What remains is to show that there can be only a small number of stages. That is, we do not restart the simulation too many times.

**Claim 10** *There are at most $L+1$ stages in the run of the algorithm* LEARNANYWAY.

**Proof:** At the beginning of each stage (except the first one) the algorithm discovers a new string where the MMQ lies and from then on MMQ can never lie on this string again, because it is added to *CounterExamples*. To be more precise, MMQ does not get a chance to lie on this string because it is never asked about it again. Let $S$ be the set of the strings that MMQ lies on. Since $|S| \leq ||S|| \leq L$, in stage $L + 1$ the MMQ can lie on no strings (i.e., it is not asked queries about any of the strings where it may lie). Therefore LEARN has to converge to the target concept $h^*$. ■

The time spent on simulation and bookkeeping is clearly polynomial in $|h^*|$, $n$, and $L$. Thus, LEARNANYWAY is a polynomial-time algorithm that uses equivalence and malicious membership queries to learn the class of concepts $H = (R, Dom, \mu)$. This concludes the proof of Theorem 7. ■

As corollaries of Theorem 7 we have the following.

**Corollary 5** *The class of regular languages, represented by DFA's, is learnable in polynomial time with equivalence and malicious membership queries.*

**Proof:** In [10] it was shown that this class of concepts is polynomially closed under finite exceptions. In [2] it was shown that it is learnable in polynomial time using membership and equivalence queries. ■

**Corollary 6** *The class of boolean decision trees is learnable in polynomial time with extended equivalence and malicious membership queries.*

**Proof:** Lemma 9 shows that the class of boolean decision trees is polynomially closed under finite exceptions. In [11] it was shown that it is learnable in polynomial time using membership and extended equivalence queries. ∎

**Corollary 7** *The class of monotone DNF formulas with finite exceptions is learnable in polynomial time with equivalence and malicious membership queries.*

**Proof:** Corollary 4 shows that the class of monotone DNF formulas with exceptions is polynomially closed under finite exceptions. In Section 5.2 we gave an algorithm that learns this class in polynomial time with membership and equivalence queries. ∎

Note that we can also learn the class of monotone DNF formulas without any exceptions with this generic algorithm, using extended equivalence and malicious membership queries, since it is just a subclass of the class that allows exceptions. However, the algorithm is much less efficient than the one described in Section 4.

## 6.2 Learning with and without "Don't knows"

In this subsection, we digress from exceptions and malicious membership queries, and focus again on limited membership queries and standard membership queries. We present a lower bound result, the proof of which has ideas useful in further subsections.

We start by briefly describing a method for converting any algorithm for exact identification from membership and equivalence queries to one that works for limited membership queries. We can also show that in some cases an exponential blowup in the number of queries is necessary.

**Theorem 8** *Every concept class that is learnable with $m$ equivalence and membership queries is learnable with $2^{\ell}(m-\ell+1)+\ell-1$ equivalence and limited membership queries, where $\ell$ is the number of $\perp$ responses received.*

**Proof:** Let Algorithm $A$ exactly identify concept class $\mathcal{C}$ from at most $m$ equivalence and membership queries. We construct a learning algorithm for equivalence and limited membership queries as follows: For each instance $x$ such that $\mathrm{LMQ}(x) = \perp$, start running two copies of $A$ in parallel, one assuming $x$ is positive and the other assuming that $x$ is negative. The bound follows by noticing that the simulation terminates in at most $m$ phases. Note that each copy of $A$ (except one) may need to make one extra equivalence query at the end to determine whether it has the correct answer. ■

The next theorem shows that in some cases such an exponential blowup in the number of queries is in fact necessary.

**Theorem 9** *There is a concept class learnable with $m$ equivalence and membership queries that requires $2^{\ell}(m-\ell+1)-1$ equivalence and limited membership queries, where $\ell$ is the number of $\perp$ responses received.*

**Proof:** We construct a concept class $\mathcal{C}$ that is a variant of ADDRESSING [20]. Let the instance space be $X = \bigcup_{i=0}^{2^{\ell}} X_i$, where the $X_i$'s are disjoint, $X_0 = \{1, \ldots, \ell\}$, and $|X_i| = m - l + 1$ for $1 \leq i \leq 2^{\ell}$. Since $|X_0| = \ell$, each of its subsets can be viewed as an $\ell$-bit number. A set $c \subseteq X$ is in $\mathcal{C}$ if and only if it has the following form. It contains exactly one element $x$ that is not in $X_0$, and if $i$ denotes the number represented by $c \cap X_0$, then that $x$ is in $X_i$.

Concept class $\mathcal{C}$ can be learned by $\ell$ membership queries for the elements in $X_0$ followed by $m - \ell$ membership queries for the elements of $X_i$, where $i$ is the number represented by the responses obtained in the first phase.

On the other hand, the following adversary strategy shows that at least $2^{\ell}(m - \ell + 1) - 1$ equivalence and limited membership queries are required to learn $\mathcal{C}$. The limited membership oracle responds $\perp$ to all instances in $X_0$. Membership queries for other elements are answered by "No." Equivalence queries are answered by providing a negative counterexample outside $X_0$. ■

Taking $\ell = m$ in the proof of Theorem 9 gives us the original concept class ADDRESSING, and an example where $m$ $\perp$ responses increase the number of queries required from $m$ for ordinary membership queries to $2^m - 1$ for limited membership and equivalence queries.

Note that ADDRESSING also causes the incomplete membership query model [7] to have an expected exponential blowup over ordinary membership queries when the probability of a $\perp$ response is a constant. For constant probability $p$ of $\perp$, the expected number of instances in $X_0$ answered $\perp$ is $pm$. This will increase the number of queries required from $m$ for ordinary membership queries to $2^{pm}$ for incomplete membership and equivalence queries.

If instead of equivalence queries one allows equivalence queries with *any set*, then such a blowup cannot occur. This follows from a result of Auer and Long [8] showing that in this model membership queries can speed up learning by only a constant factor.

## 6.3 Strict versus Nonstrict

Recall that in the nonstrict model the final hypothesis need only agree with the target concept on points $x$ such that $\mathrm{LMQ}(x) \neq \perp$, while in the strict model, they must be exactly equal.

Every learning algorithm that works in the strict model can be run in the nonstrict model without increasing its complexity. A relationship in the other direction can be established by a method similar to the one used in the proof of Theorem 7 in subsection 6.1.

Every learning algorithm that works in the nonstrict model can be adapted to work in the strict model as follows. First note that the problem that may occur when running a nonstrict algorithm in the strict model is that it may receive as a counterexample to an equivalence query a point that was previously classified as a "Don't know" in a membership query. In this case, the execution of the algorithm is interrupted. The algorithm is then restarted, remembering the point and its classification for possible later use in answering a membership query. Since each interruption corresponds to a new "Don't know," this simulation essentially adds a multiplicative factor of $\ell$ to the complexity of the learning algorithm.

Hence, from the point of view of polynomial learnability, the strict and nonstrict models using EQ's and LMQ's are equivalent.

## 6.4 Lies versus Omissions

As noted in Subsections 2.3 and 3.4, learning with MMQ's and EQ's is at least as difficult as with LMQ's and EQ's. In order to provide an example where learning with MMQ's and EQ's is in fact more difficult, let us again consider the concept class ADDRESSING described above. That concept class can be learned with at most $m + 2^\ell$ LMQ's and EQ's for any $\ell$. On the other hand, the following adversary strategy shows that at least $\sum_{i=0}^{\ell} \binom{m}{i} - 1$ MMQ's and EQ's are necessary. Answer 0 for all MMQ's, and provide a negative counterexample from outside $X_0$ for every EQ.

This example, with $\ell = \log m$, shows that the number of MMQ's and EQ's necessary to learn a concept class cannot be bounded by any polynomial in the number of LMQ's and EQ's. On the other hand, since in this example the number of LMQ's and EQ's is exponential in $\ell$, it does not answer the question "Are there any concept classes polynomially learnable from EQ's and LMQ's that are not polynomially learnable from EQ's and MMQ's?"

# 7 Summary

We can summarize the results proven in this paper as follows:

1. Strict and nonstrict models of learning from equivalence and limited membership queries are polynomial-time equivalent. (Subsection 6.3.)
2. Polynomial-time learnability from equivalence and malicious membership queries implies polynomial-time learnability from from equivalence and limited membership queries. (Subsection 2.3.)
3. Monotone monomials are learnable in the nonstrict model from limited membership queries alone using at most $n\ell + n + 1$ queries. (Theorem 1.)
4. Learning monotone monomials in the nonstrict model from limited membership queries alone may require $\Omega(n^{c+1})$ queries when $O(n^c)$ omissions are given. (Corollary 1.)
5. Monotone $k$-term DNF formulas are learnable in the nonstrict model from limited membership queries alone with at most $O(k(n^k + n^2\ell))$ queries. (Theorem 3.)
6. Monotone DNF formulas are learnable in the nonstrict model from at most $n(m + \ell)$ limited membership queries and $m + 1$ equivalence queries, and in the

strict model from at most $n(m+\ell)$ limited membership queries and $m+\ell+1$ equivalence queries. (Theorem 4.)

7. Monotone DNF formulas are learnable from $O(mn^2+\ell n^3)$ malicious membership queries and $O(m+\ell n)$ equivalence queries. (Theorem 5.)

   Note that the bound on the number of queries is better if limited membership queries are used instead of malicious ones.

8. The class of monotone DNF formulas with exceptions can be learned from $O(m^2n^4\ell^3)$ standard membership queries and $O(m^2n^2\ell^3)$ equivalence queries. (Theorem 6.)

9. Any class of concepts that is polynomially closed under finite exceptions and is learnable in polynomial time from equivalence and standard membership queries is also learnable in polynomial time from equivalence and malicious membership queries. (Theorem 7.)

10. The classes of:

    (a) regular languages, represented by DFA's,

    (b) boolean decision trees[1],

    (c) monotone DNF formulas with finite exceptions

    are learnable in polynomial time with equivalence and malicious membership queries. (Corollaries 5, 6 and 7.)

11. Every concept class that is learnable from $m$ equivalence and standard membership queries is learnable in the strict model from $2^\ell(m-\ell+1)+\ell-1$ equivalence and limited membership queries. (Theorem 8.)

12. There exists a concept class learnable with $m$ equivalence and standard membership queries that requires $2^\ell(m-\ell+1)-1$ equivalence and limited membership queries to be learned in the strict model. (Theorem 9.)

13. There exists a concept class learnable in the strict model with $m+2^\ell$ equivalence and limited membership queries that requires $\sum_{i=0}^{\ell}\binom{m}{i}-1$ equivalence and malicious membership queries to be learned. (Subsection 6.4.)

[1]If extended equivalence queries are used

# 8 Discussion and Open Problems

As noted in the introduction, there are many classes of concepts that are efficiently learnable with membership and equivalence queries. For some of them we now have learning algorithms that use equivalence and limited or malicious membership queries. Many other problems still remain unexplored. For example, there is not yet any algorithm for learning read-once formulas from equivalence and limited or malicious membership queries, even though there is an algorithm for learning read-once formulas ¿from equivalence and standard membership queries. A start in this direction is made in [4], which gives a randomized polynomial-time algorithm to learn $\mu$-DNF formulas with equivalence and malicious membership queries. In the model of PAC learning with membership queries, it would be interesting to see whether Baum's algorithm [9] can be modified to tolerate "I don't know" answers.

Another type of open problems is finding lower bounds for any of the classes of concepts for which we give learning algorithms using equivalence and limited or malicious membership queries. So far, we have lower bounds for only a specially constructed class and for monotone monomials in the model that uses only limited membership queries. For other classes, the following question is still relevant: can we prove something stronger than the trivial bound that there must be more membership queries than lies or omissions?

Moving on to the comparison of models, we have two very intriguing questions. The first one is, are there any classes of concepts that are polynomial-time learnable from equivalence and limited membership queries, but not polynomial-time learnable from equivalence and malicious membership queries? The second question is, are learning with exceptions and learning with lies equally difficult for classes that are polynomially closed under finite exceptions, or is learning with exceptions more difficult for these classes? That is, is there a class that is polynomially closed under finite exceptions, is learnable with malicious membership queries in polynomial time, and is not polynomial-time learnable with exceptions? We also do not know how the difficulty of learning with exceptions classes that are not polynomially closed under finite exceptions relates to learning such classes with malicious membership queries.

A less important extension of this work would be to improve the time bound for the algorithm that learns monotone DNF formulas with exceptions and possibly for the one that learns monotone DNF from equivalence and malicious membership queries.

# 9 Comments

This work appeared as two separate papers in the Proceedings of the 7th Annual ACM Conference on Computational Learning Theory [5], [23]. Part of it is also available as a technical report [6].

# References


[1] John R. Anderson. *Cognitive Psychology and Its Implications.* W. H. Freeman and Company, 1980.

[2] D. Angluin. Learning regular sets from queries and counterexamples. *Inform. Comput.*, 75(2):87–106, November 1987.

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

[4] D. Angluin. Exact learning of $\mu$-DNF formulas with malicious membership queries. Technical Report YALEU/DCS/TR-1020, Yale University Department of Computer Science, March 1994.

[5] D. Angluin and M. Kriķis. Learning with malicious membership queries and exceptions. In *Proc. 7th Annu. ACM Workshop on Comput. Learning Theory*, pages 57–66. ACM Press, New York, NY, 1994.

[6] D. Angluin and M. Krikis. Malicious membership queries and exceptions. Technical Report YALEU/DCS/TR-1019, Yale University Department of Computer Science, March 1994.

[7] D. Angluin and D. K. Slonim. Randomly fallible teachers: learning monotone DNF with an incomplete membership oracle. *Machine Learning*, 14(1):7–26, 1994.

[8] P. Auer and P. M. Long. Simulating access to hidden information while learning. In *Proc. of the 26th Annual ACM Symposium on Theory of Computing*, pages 263–272. ACM Press, New York, NY, 1994.

[9] E. Baum. Neural net algorithms that learn in polynomial time from examples and queries. *IEEE Transactions on Neural Networks*, 2:5–19, 1991.

[10] R. Board and L. Pitt. On the necessity of Occam algorithms. *Theoret. Comput. Sci.*, 100:157–184, 1992.

[11] N. Bshouty. Exact learning via the monotone theory. In *Proc. of the 34th Symposium on the Foundations of Comp. Sci.*, pages 302–311. IEEE Computer Society Press, Los Alamitos, CA, 1993.

[12] William J. Bultman. *Topics in the Theory of Machine Learning and Neural Computing*. PhD thesis, University of Illinois at Chicago Mathematics Department, 1991.

[13] T. Dean, D. Angluin, K. Basye, S. Engelson, L. Kaelbling, E. Kokkevis, and O. Maron. Inferring finite automata with stochastic output functions and an application to map learning. In *Proceedings of AAAI-92*, pages 208–214. AAAI, 1992.

[14] M. Frazier, S. Goldman, N. Mishra, and L. Pitt. Learning from a consistently ignorant teacher. In *Proc. 7th Annu. ACM Workshop on Comput. Learning Theory*, pages 328–339. ACM Press, New York, NY, 1994.

[15] S. A. Goldman, M. J. Kearns, and R. E. Schapire. Exact identification of read-once formulas using fixed points of amplification functions. *SIAM J. Comput.*, 22(4):705–726, 1993.

[16] S. A. Goldman and H. D. Mathias. Learning k-term DNF formulas with an incomplete membership oracle. In *Proc. 5th Annu. Workshop on Comput. Learning Theory*, pages 77–84. ACM Press, New York, NY, 1992.

[17] M. Kearns. Efficient noise-tolerant learning from statistical queries. In *Proc. 25th Annu. ACM Sympos. Theory Comput.*, pages 392–401. ACM Press, New York, NY, 1993.

[18] E. Kushilevitz and Y. Mansour. Learning decision trees using the Fourier spectrum. In *Proc. of the 23rd Symposium on Theory of Computing*, pages 455–464. ACM Press, New York, NY, 1991.

[19] Kevin J. Lang and Eric B. Baum. Query learning can work poorly when a human oracle is used. In *International Joint Conference on Neural Networks*, Beijing, 1992.

[20] W. Maass and G. Turán. Lower bound methods and separation results for on-line learning models. *Machine Learning*, 9:107–145, 1992.

[21] D. Ron and R. Rubinfeld. Learning fallible finite state automata. In *Proc. 6th Annu. Workshop on Comput. Learning Theory*, pages 218–227. ACM Press, New York, NY, 1993.

[22] Y. Sakakibara. On learning from queries and counterexamples in the presence of noise. *Inform. Proc. Lett.*, 37(5):279–284, March 1991.

[23] R. Sloan and G. Turan. Learning with queries but incomplete information. In *Proc. of the 7th Annual ACM Conference on Computational Learning Theory*, pages 237–245. ACM Press, New York, NY, 1994.

[24] L. G. Valiant. Learning disjunctions of conjunctions. In *Proceedings of the 9th International Joint Conference on Artificial Intelligence, vol. 1*, pages 560–566, Los Angeles, California, 1985. International Joint Committee for Artificial Intelligence.

[25] Y. Zhuravlev and Y. Kogan. Realization of boolean functions with a small number of zeros by disjunctive normal forms, and related problems. *Soviet Math. Doklady*, 32:771–775, 1985.