On PAC learning algorithms for rich Boolean function classes

Lisa Hellerstein*
Department of Computer and Information Science
Polytechnic University
Brooklyn, NY U.S.A.
hstein@duke.poly.edu

Rocco A. Servedio†
Department of Computer Science
Columbia University
New York, NY U.S.A.
rocco@cs.columbia.edu

January 30, 2007

Abstract

We give an overview of the fastest known algorithms for learning various expressive classes of Boolean functions in the Probably Approximately Correct (PAC) learning model. In addition to surveying previously known results, we use existing techniques to give the first known subexponential-time algorithms for PAC learning two natural and expressive classes of Boolean functions: sparse polynomial threshold functions over the Boolean cube $\{0,1\}^n$ and sparse GF_2 polynomials over $\{0,1\}^n$.

*Supported in part by NSF Award IIS-0534908, while visiting the University of Wisconsin, Madison.

†Supported in part by NSF CAREER award CCF-0347282, by NSF award CCF-0523664, and by a Sloan Foundation Fellowship.

1 Introduction

Computational learning theory is the study of the inherent abilities and limitations of algorithms that learn from data. A broad goal of the field is to design computationally efficient algorithms that can learn Boolean functions $f : \{0,1\}^n \to \{-1,1\}$. A general framework within which this question is often addressed is roughly the following:

1. There is a fixed class C of possible target functions over $\{0,1\}^n$ which is *a priori* known to the learning algorithm. (Such function classes are often referred to as *concept classes*, and the functions in such classes are referred to as *concepts*.)

2. The learning algorithm is given some form of access to information about the unknown target concept $c \in C$.

3. At the end of its execution, the learning algorithm outputs a hypothesis $h : \{0,1\}^n \to \{-1,1\}$, which ideally should be equivalent or close to c.

Different ways of instantiating (2) and (3) above – what form of access to c is the learner given? what is required of the hypothesis function h? etc. – give rise to different learning models. Within a given learning model, different choices of the Boolean function class C (i.e. different ways of instantiating (1) above) give rise to different learning problems such as the problem of learning an unknown conjunction, an unknown linear threshold function, or an unknown decision tree.

In this paper we will focus exclusively on the widely studied Probably Approximately Correct (PAC) learning model introduced by Valiant [41]. In this learning model, which we define precisely in Section 2.1, the learning algorithm is only given access to *independent random examples labelled according to c,* i.e. access to input-output pairs $(x, c(x))$ where each x is independently drawn from the same unknown probability distribution. Thus the learning algorithm has no control over the choice of examples used for learning. Such a model may be viewed as a good first-order approximation of commonly encountered scenarios in machine learning where one must learn from a given training set of examples generated according to some unknown random process.

(We note that a wide range of models exist in which the learning algorithm has other forms of access to the target function; in particular several standard models allow the learner to make black-box queries to the target function, which are often known as *membership queries.* Many powerful and elegant learning algorithms are known in various models that permit membership queries, see e.g. [1, 3, 9, 19, 28], but we will not discuss this work here. A rich body of results have also been obtained for the *uniform-distribution* variant of the PAC learning model, in which the learner need only succeed when given uniform random examples from $\{0,1\}^n$; see e.g. [42, 29, 11, 20, 36] for some representative work in this setting. Finally, we note that there also exist well-motivated and well-studied learning models in which the learning algorithm only has some more limited form of access to c than random labeled examples, see e.g. [4, 21].)

There are well-known polynomial-time PAC learning algorithms for concept classes consisting of simple functions such as conjunctions and disjunctions [41], decision lists [37], parity functions [14, 18], and halfspaces [8]. We give a concise overview of the current state of the art for learning richer concept classes consisting of more expressive Boolean functions such as decision trees, Disjunctive Normal Form (DNF) formulas, intersections of halfspaces, and various restricted classes of Boolean formulas. For each of these "rich" concept classes true polynomial-time algorithms are not (yet) known, but as we describe below, it is possible to give provable guarantees which improve substantially over naive exponential runtime bounds.

Class of functions over $\{0,1\}^n$	**Runtime**	**Technique**
size-s decision trees	$n^{O(\log s)}$	Recursive algorithm [13], DLs [5], PTFs [folklore]
s-term DNF formulas	$n^{O(n^{1/3} \log s)}$	PTFs [26]
size-s, depth-d Boolean formulas	$n^{s^{1/2}(\log s)^{O(d)}}$	PTFs [35]
intersections of k halfspaces of weight W	$\min\{n^{O(W^{1/2} \log k)}, n^{O(k \log k \log W)}\}$	PTFs [25]
arbitrary functions of k halfspaces of weight W	$n^{O(k^2 \log W)}$	PTFs [25]
degree-d PTFs	$n^{O(d)}$	PTFs [folklore]
weight-W PTFs	$n^{O(n^{1/3} \log W)}$	PTFs [25]
length-s PTFs	$n^{O((n \log s)^{1/2})}$	PTFs, generalized DLs [this paper]
s-sparse GF_2 polynomials	$n^{O((n \log s)^{1/2})}$	generalized DLs [this paper]

Table 1: Fastest known runtimes of PAC learning algorithms for various classes of Boolean functions. In the "Techniques" column, "DL" refers to a decision list learning algorithm and "PTF" refers to a polynomial threshold function learning algorithm.

One perhaps surprising point which emerges from our survey is that a single linear programming based algorithm for learning *low-degree polynomial threshold functions* gives the current state-of-the-art results for learning a wide range of rich concept classes, including all those we will discuss in Sections 3.1 through 3.4. In Section 3.5 we extend the known scope of applicability of this algorithm by showing that it can be used to learn the class of *sparse* polynomial threshold functions over $\{0,1\}^n$ (regardless of their degree or the size of their coefficients) in subexponential time:

Theorem 1. *The class of s-sparse polynomial threshold functions over $\{0,1\}^n$ can be PAC learned in time $2^{O((n \log s)^{1/2} \log n)}$.*

In Section 4 we describe a different approach to obtaining PAC learning algorithms for rich function classes; this is essentially an augmented version of an algorithm for learning decision lists due to Rivest [37]. Bshouty *et al.* [12] have used this approach to learn a restricted class of branching programs in polynomial time. We show that the approach can also be used to obtain the first known subexponential-time algorithm for PAC learning sparse GF_2 polynomials:

Theorem 2. *The class of s-sparse GF_2 polynomials over $\{0,1\}^n$ can be PAC learned in time $2^{O((n \log s)^{1/2} \log n)}$.*

We feel that exploring further applications of this approach is an interesting and potentially fruitful direction for future work.

Throughout the paper we highlight various open questions, with an emphasis on problems where progress both would be of interest and (in the view of the authors) would seem most likely to be feasible.

2 Distribution-Independent Learning

2.1 The learning model

In an influential 1984 paper Valiant introduced the *Probably Approximately Correct* (PAC) model of learning Boolean functions from random examples [41]. (See the book [22] for an excellent and detailed introduction to the model.) In the PAC model a learning algorithm has access to an *example oracle* $EX(c, \mathcal{D})$ which, when queried, provides a labeled example $(x, c(x))$ where x is drawn from a fixed but unknown distribution $\mathcal{D}$ over $\{0,1\}^n$ and $c \in C$ is the unknown target concept which the algorithm is trying to learn. Given Boolean functions h, c on $\{0,1\}^n$, we say that h is an *ϵ-approximator for c under $\mathcal{D}$* if $\Pr_{x \in \mathcal{D}}[h(x) = c(x)] \geq 1 - \epsilon$. The goal of a PAC learning algorithm is to output a hypothesis h which is an ϵ-approximator for the unknown target concept c with high probability.

More precisely, an algorithm A is a *PAC learning algorithm for concept class C* if the following condition holds: for any $c \in C$, any distribution $\mathcal{D}$ on $\{0,1\}^n$, and any $0 < \epsilon < \frac{1}{2}, 0 < \delta < 1$, if A is given ϵ, δ as input and has access to $EX(c, \mathcal{D})$, then A outputs (a representation of) some $h : \{0,1\}^n \to \{-1,1\}$ which satisfies $\Pr_{x \in \mathcal{D}}[h(x) \neq c(x)] \leq \epsilon$ with probability at least $1 - \delta$. We say that *A PAC learns C in time $t = t(n, \epsilon, \delta, s)$* if A runs for at most t time steps and outputs a hypothesis h which can be evaluated on any point $x \in \{0,1\}^n$ in time t; here $s = \text{size}(c)$ is a measure of the "size" of the target concept $c \in C$. Note that no restriction is put on the form of the hypothesis h other than that it be efficiently evaluatable. In particular, h need not belong to the concept class C (i.e. we do not restrict ourselves to "proper" learning algorithms).

PAC learning algorithms are closely related to *consistency algorithms*. Given two concept classes C and H, where $C \subseteq H$, a consistency algorithm for C *using* H takes as input a sample $S = \{(a_1, b_1), (a_2, b_2), \ldots, (a_m, b_m)\}$, where each $a_i \in \{0,1\}^n$ and each $b_i \in \{-1,1\}$. The goal of the algorithm is to output the representation of a concept $h \in H$ that is *consistent* with S, meaning that $h(a_i) = b_i$ for all $(a_i, b_i) \in S$. If no such c exists, the algorithm outputs "FAILURE". If $H = C$, the algorithm is called a consistency algorithm for C.

There are well-known relationships between the existence of a consistency algorithm for C and the PAC learnability of C. One such relationship is as follows:

Fact 3. *[7, 8] Let C and H be concept classes defined on $\{0,1\}^n$, such that $C \subseteq H$. Suppose that $\mathcal{A}$ is a consistency algorithm for C using H. Then the following is a PAC learning algorithm for C: Draw $\frac{1}{\epsilon} \ln \frac{|H|}{\delta}$ examples from $EX(c, \mathcal{D})$ and run $\mathcal{A}$ on the set of examples obtained.*

In this result, the number of examples drawn depends linearly on $\ln |H|$. In a similar (and deeper) result that is often cited, the number of examples drawn depends linearly on the *VC-dimension* of H [8]. We refer the reader to relevant references (e.g. [8], [22]) for the definition of VC-dimension and discussion of its relation to learning. For simplicity, we use Fact 3 in what follows, since it suffices to prove the results stated below.

It is well known (see e.g. [22]) that the runtime dependence of a PAC learning algorithm on δ can always be made logarithmic in $\frac{1}{\delta}$. Moreover, for all the results we discuss, the runtime dependence on ϵ is polynomial in $\frac{1}{\epsilon}$. Thus throughout this paper we discuss the running time of PAC learning algorithms as functions only of n and (when appropriate) the size parameter s. Finally, we often refer to algorithms that PAC learn a class of *representations* (e.g. decision trees of a certain size), when technically we should say that the algorithms learn the concepts expressed by those representations. In general, where it is unlikely to cause confusion, we will not distinguish between representations and their associated concepts.

2.2 The main technique: polynomial threshold functions

A polynomial threshold function is defined by a polynomial $p(x_1, \dots, x_n)$ with real coefficients. The output of the polynomial threshold function on input $x \in \{0,1\}^n$ is 1 if $p(x_1, \dots, x_n) \geq 0$ and is -1 otherwise. Because the domain of the function is $\{0,1\}^n$, different polynomials can specify the same polynomial threshold function. The *degree* of a polynomial threshold function is simply the degree of the polynomial p. A *linear threshold function* or *halfspace* is a polynomial threshold function of degree 1. Since we will only be concerned with the input space $\{0,1\}^n$, we may without loss of generality consider only polynomial threshold functions which correspond to multilinear polynomials.

It is well known that there are poly(n)-time PAC learning algorithms for the concept class of linear threshold functions over $\{0,1\}^n$. This follows from the fact that the problem of finding a linear threshold function consistent with a sample can be expressed as a linear programming problem, thus polynomial-time algorithms for linear programming [23] can be used as consistency algorithms for the class of linear threshold functions. Since the number of linear threshold functions on $\{0,1\}^n$ is $2^{\Theta(n^2)}$ [32], Fact 3 implies a poly(n)-time PAC learning algorithm for the class.[1]

As various authors have noted [6, 26], such PAC learning algorithms for learning linear threshold functions can be run over an expanded feature space of $N = \sum_{i=1}^{d} \binom{n}{d}$ monomials of degree at most d to learn degree-d polynomial threshold functions in time poly(N). (This approach is closely related to using a Support Vector Machine with a degree-d polynomial kernel, see e.g. [39].)

For example, consider the problem of learning degree-2 polynomial threshold functions defined on n variables. Associate with each degree-2 polynomial

$$p(x_1, \dots, x_n) = a_0 + \sum_{i=1}^{n} a_i x_i + \sum_{j=1}^{n} \sum_{k=j+1}^{n} a_{j,k} x_j x_k$$

a linear polynomial

$$\hat{p}(x_1, \dots, x_n, y_{1,2}, y_{1,3}, \dots, y_{n-1,n}) = a_0 + \sum_{i=1}^{n} a_i x_i + \sum_{j=1}^{n} \sum_{k=j+1}^{n} a_{j,k} y_{j,k}$$

where the $y_{j,k}$ are new variables. Further, for $d = (d_1, \dots, d_n)$ in $\{0,1\}^n$ let

$$\hat{d} = (d_1, \dots, d_n, \hat{d}_{1,2}, \hat{d}_{1,3}, \dots, \hat{d}_{n-1,n})$$

where each $\hat{d}_{j,k} = d_j d_k$. Clearly, $p(d) = \hat{p}(\hat{d})$ for all $d \in \{0,1\}^n$.

Using this association, it is easy to convert a consistency algorithm for linear threshold functions into a consistency algorithm for degree-2 polynomial threshold functions as follows: Given a sample $S \subseteq \{0,1\}^n \times \{-1,1\}$ of a degree-2 polynomial threshold function p in n variables, form the sample $S' = \{(\hat{a}, b) | (a,b) \in S\}$. Run the consistency algorithm for linear threshold functions on S'. Since $\hat{p}$ is consistent with S', the consistency algorithm will output a linear polynomial h' over the variables x_i and $y_{j,k}$. Clearly h' is a function of $N = \binom{n}{1} + \binom{n}{2}$ variables. Convert h' into a degree-2 polynomial

[1] We note that for the class of linear threshold functions, using the VC-dimension version of Fact 3, as is done in [8], does reduce both the number of examples drawn by the PAC algorithm and the runtime of the algorithm by a polynomial factor. This is because the number of linear threshold functions on $\{0,1\}^n$ is $2^{\Theta(n^2)}$ [32] while the VC-dimension of the class is $n+1$. A similar phenomenon occurs with the class of degree-d polynomial threshold functions. In future sections, where we present applications of the polynomial-threshold function learning algorithm, the polynomial factor improvement that results from using the VC version of Fact 3 is always obscured by the big-Oh notation in our bounds.

threshold function h over the variables $x_1, \ldots, x_n$, by replacing each $y_{j,k}$ occurence in the linear polynomial h' with $x_j x_k$. The function h is consistent with S.

Generalizing this approach, it follows that any poly(n)-time consistency algorithm for linear threshold functions can be converted into a poly(N)-time consistency algorithm for degree-d polynomial threshold functions, where $N = \sum_{i=1}^{d} \binom{n}{i} \leq (\frac{en}{d})^d$. Further, since the number of linear threshold functions over n variables is $2^{\Theta(n^2)}$, the number of degree-d polynomial threshold functions is $2^{O(N^2)}$. Fact 3 now implies the following:

Fact 4. *Let C be a class of functions each of which can be expressed as an degree-d polynomial threshold function over $\{0,1\}^n$. Then there is a poly(N)-time PAC learning algorithm for C, where $N = \sum_{i=1}^{d} \binom{n}{i} \leq (\frac{en}{d})^d$.*

Thus, in order to get an upper bound on the runtime required to learn a concept class C, it is enough to bound the degree of polynomial threshold functions which represent the concepts in C. This approach has proved quite powerful as we now describe.

3 Learning rich Boolean function classes via polynomial threshold representations

3.1 Decision Trees

A *Boolean decision tree* T is a rooted binary tree in which each internal node has two ordered children and is labeled with a variable, and each leaf is labeled with a bit $b \in \{-1, +1\}$. A decision tree T computes a Boolean function $f : \{0,1\}^n \to \{-1,1\}$ in the obvious way: on input x, if variable x_i is at the root of T we go to either the left or right subtree depending on whether x_i is 0 or 1. We continue in this way until reaching a bit leaf; the value of this bit is $f(x)$. A decision tree is *reduced*, if each variable appears at most once in any path from the root down to a leaf. Every decision tree can be easily converted into an equivalent reduced decision tree since testing a variable a second time along a root-leaf path does not yield any new information.

The *size* of a decision tree is the number of leaves in the tree. Another measure of the complexity of a tree is its *rank*. The *rank* of a decision tree is defined recursively as follows. If T has exactly one node, then $\text{rank}(T) = 0$. If T has more than one node, then its rank depends on the ranks of its two subtrees, T_0 and T_1. If $\text{rank}(T_0) \neq \text{rank}(T_1)$, then $\text{rank}(T) = \max(\text{rank}(T_0), \text{rank}(T_1))$. If $\text{rank}(T_0) = \text{rank}(T_1)$, then $\text{rank}(T) = \text{rank}(T_0) + 1$ $(= \text{rank}(T_1) + 1)$. Rank and size are related by the following: Given a reduced decision tree T of rank q and size s over n variables, we have $2^q \leq s \leq (en/q)^q$ [13].

Algorithms for learning decision trees have received much attention both from applied and theoretical perspectives. Ehrenfeucht and Haussler [13] gave a recursive algorithm which learns any size-s decision tree in $n^{O(\log s)}$ time; while no faster algorithms are known, various alternative algorithms with the same quasipolynomial runtime have since been given. Blum [5] showed that every size-s decision tree is equivalent to some $\log(s)$-decision list. An r-decision list is defined by a list $(T_1, b_1), \ldots, (T_m, b_m), b_{m+1}$ where each T_i is a conjunction of at most r literals and each b_i is an output bit. The value of the decision list on input x is b_i where i is the first index such that T_i is satisfied by x; if x satisfies no T_i then the output is b_{m+1}. (See Section 4 for a definition of a generalized notion of decision lists.) Since r-decision lists are PAC learnable in $n^{O(r)}$ time [37], this gives an equally efficient alternative algorithm to [13].

An easy argument shows that any r-decision list can be expressed as a degree-r polynomial threshold function; thus Blum's result implies that any size-s decision tree is computed by a $\log(s)$-degree polynomial threshold function. Thus for decision trees we may use Fact 4 to obtain the fastest known algorithm, but as described above other equally fast algorithms are also known. However, for each of the concept classes discussed below in Sections 3.2 through 3.5, the Fact 4 approach is the only known way to achieve the current fastest runtimes.

3.2 DNF formulas

A *disjunctive normal form* formula, or DNF, is a disjunction $T_1 \vee \cdots \vee T_s$ of conjunctions of Boolean literals. An s-term DNF is one which has at most s conjunctions (also known as terms). Learning s-term DNF formulas in time $\mathrm{poly}(n, s)$ is a longstanding open question which goes back to Valiant's inception of the PAC learning model.

The first subexponential time algorithm for learning DNF was due to Bshouty [10] and learns any s-term DNF over n variables in time $2^{O((n \log s)^{1/2} \log^{3/2} n)}$. At the heart of Bshouty's algorithm is a structural result which shows that that any s-term DNF can be expressed as an $O((n \log n \log s)^{1/2})$-decision list; together with the aforementioned algorithm of [37] this gives the result. Subsequently Tarui and Tsukiji [40] gave a different algorithm for learning DNF with a similar runtime bound. Their algorithm adapted the machinery of "approximate inclusion/exclusion" developed by Linial and Nisan [30] in combination with hypothesis boosting [15] and learns s-term DNF in time $2^{O(n^{1/2} \log n \log s)}$.

In [26], Klivans and Servedio showed that any DNF formula with s terms can be expressed as a polynomial threshold function of degree $O(n^{1/3} \log s)$. By Fact 4 this yields an algorithm for learning s-term DNF in time $2^{O(n^{1/3} \log n \log s)}$, which is the fastest known time bound for most interesting values of s.

Several lower bounds on polynomial threshold function degree for DNFs are known which complement the $O(n^{1/3} \log s)$ upper bound of [26]. A well-known theorem of Minsky and Papert [31] shows that the "one-in-a-box" function (which is equivalent to an $n^{1/3}$-term DNF on n variables) requires polynomial threshold function degree $\Omega(n^{1/3})$. Minsky and Papert also proved that the parity function on k variables require polynomial threshold function degree at least k; since s-term DNF formulas can compute the parity function on $\log s$ variables, this gives an $\Omega(\log s)$ lower bound for s-term DNF as well. These known results motivate:

Question 5. *Can we close the remaining gap between the $O(n^{1/3} \log s)$ upper bound and the $\max\{n^{1/3}, \log s\}$ lower bound on polynomial threshold function degree for s-term DNF?*

Note that for decision trees no gap at all exists; Blum's approach gives a $\lfloor \log s \rfloor$ degree upper bound for size-s decision trees, and the parity function shows that this is tight.

3.3 Boolean Formulas

Known results on learning Boolean formulas of depth greater than two are quite limited. O'Donnell and Servedio [35] have shown that any unbounded fan-in Boolean AND/OR/NOT formula of depth d and size (number of leaves) s is computed by a polynomial threshold function of degree $\sqrt{s}(\log s)^{O(d)}$. By Fact 4 this gives a $2^{\tilde{O}(n^{1/2+\epsilon})}$ time PAC learning algorithm for linear-size Boolean formulas of depth $o(\frac{\log n}{\log \log n})$. (Here we write $\tilde{O}(n^c)$ to indicate a function that is $n^c \cdot (\log n)^{O(1)}$.)

It would be very interesting to weaken the dependence on either size or depth in the results of [35]:

Question 6. *Does every AND/OR/NOT formula of size s have a polynomial threshold function of degree $O(\sqrt{s})$, independent of its depth?*

An $O(\sqrt{s})$ degree bound would be the best possible since size-s formulas can express the parity function on $\sqrt{s}$ variables.

Question 7. *Does every depth-3 AND/OR/NOT formula of size poly(n) have a polynomial threshold function of degree $o(n)$?*

The strongest degree lower bound known for poly(n)-size formulas of small depth is $\Omega(n^{1/3}(\log n)^{2(d-2)/3})$ for formulas of depth $d \geq 3$ [35]. A lower bound of $\Omega(n^{2/5})$ for an explicit linear-size, depth-3 formula is conjectured in [35]. Some related results were proved by Krause and Pudlak [27], who gave an explicit depth-3 formula that requires any polynomial threshold function to have $2^{n^{\Omega(1)}}$ many monomials.

We note that there is some reason to believe that the class of arbitrary constant-depth, polynomial-size AND/OR/NOT Boolean formulas (e.g. the class of AC^0 circuits) is not PAC learnable in poly(n) time. Kharitonov [24] has shown that an $n^{(\log n)^{o(d)}}$-time algorithm for learning poly(n)-size, depth-d Boolean formulas for sufficiently large constant d would contradict a strong but plausible cryptographic assumption about the hardness of integer factorization (essentially the assumption is that factoring n-bit integers is $2^{n^{\epsilon}}$-hard in the average case for some absolute constant $\epsilon > 0$; see [24] for details).

3.4 Intersections of Halfspaces

In addition to the concept classes of Boolean formulas discussed in the previous sections, there is considerable interest in studying the learnability of various geometrically defined concept classes. As noted in Section 2.2, efficient algorithms are known which can learn a single halfspace over $\{0,1\}^n$. Algorithms for learning a single halfspace are at the heart of some of the most widely used and successful techniques in machine learning such as support vector machines [39] and boosting algorithms [15, 16]. Thus it is of great interest to obtain such algorithms for learning richer functions defined in terms of several halfspaces, such as intersections of two or more halfspaces.

A halfspace f has *weight* W if it can be expressed as $f(x) = \mathrm{sgn}(w_1x_1 + \cdots + w_nx_n - \theta)$ where each w_i is an integer and $\sum_{i=1}^n |w_i| \leq W$. Well known results of Muroga *et al.* [33] show that any halfspace over $\{0,1\}^n$ is equivalent to some halfspace of weight $2^{O(n \log n)}$, and Håstad [17] has exhibited a halfspace which has weight $2^{\Omega(n \log n)}$. All of the current fastest algorithms for learning intersections of halfspaces have a significant runtime dependence on the weight W.

Using techniques of Beigel *et al.* [2], Klivans *et al.* [25] showed that any intersection of k halfspaces of weight W is computed by a polynomial threshold function of degree $O(k \log k \log W)$. By Fact 4, this gives a quasipolynomial-time ($n^{\mathrm{polylog}(n)}$) algorithm for learning an intersection of polylog(n) many polynomial-weight halfspaces. Since the "one-in-a-box" function on k^3 variables can be expressed as an intersection of k halfspaces each of weight $W = k^2$, we have that for $W = k^2$ there is an $\Omega(k)$ degree lower bound which nearly matches the $O(k \log k \log w)$ upper bound. It is also shown in [25] that any intersection of k halfspaces of weight W can be expressed as a polynomial threshold function of degree $O(\sqrt{W} \log k)$; this gives a stronger bound in cases where W is small and k is large.

More generally, [25] showed that any Boolean function of k halfspaces of weight W is computed by a polynomial threshold function of degree $O(k^2 \log W)$. It follows that not just intersections, but in fact any Boolean function of polylog(n) many polynomial-weight halfspaces can be learned in quasipolynomial time.

While the above results are useful for intersections of halfspaces whose weights are not too large, in the general case they do not give a nontrivial bound even for an intersection of two halfspaces. A major open question is:

Question 8. *Is there a $2^{o(n)}$ time algorithm which can PAC learn the intersection of two arbitrary halfspaces over $\{0,1\}^n$?*

An affirmative answer to the above question would immediately follow from an affirmative answer to the following:

Question 9. *Can every intersection of two halfspaces over $\{0,1\}^n$ be computed by a polynomial threshold function of degree $o(n)$?*

The strongest known lower bound on polynomial threshold function degree for intersections of two halfspaces is quite weak; in [35] it is shown that an intersection of two majority functions (which are weight-n halfspaces) requires polynomial threshold function degree $\Omega(\frac{\log n}{\log \log n})$. Thus there is an exponential gap in our current knowledge of the answer to Question 9.

3.5 Polynomial Threshold Functions

Fact 4 states that degree-d polynomial threshold functions can be PAC learned in time $n^{O(d)}$. It is natural to consider the learnability of polynomial threshold functions in terms of other measures of their complexity relating to the *coefficients* of the polynomial rather than the degree.

One natural measure of the complexity of a polynomial threshold function is its *weight*. A polynomial threshold function defined by a polynomial $p(x_1, \ldots, x_n)$ has weight W if p is a sum of monomials with integer coefficients whose magnitudes sum to W. (This is the natural extension of the notion of the weight of a halfspace, which is simply a degree-1 polynomial threshold function, defined in Section 3.4.) A weight-W polynomial threshold function may be viewed as a depth-two circuit composed of a fanin-W MAJORITY gate on the top level with inputs that are (possibly negated) AND gates of arbitrary fanin. Klivans *et al.* [25] have shown that regardless of its degree, any weight-W polynomial threshold function can be expressed as a polynomial threshold function of degree $n^{1/3}\log(W)$. This is a generalization of the degree bound for DNF formulas stated in Section 3.2.

Another natural measure of the complexity of a polynomial threshold function is its *sparsity* or *length*. A polynomial threshold function given by $p(x_1, \ldots, x_n)$ has length s if p has s monomials with nonzero coefficients. These coefficients may be arbitrary; thus, while it is clear that the weight of a polynomial threshold function is an upper bound on its length, it is possible for a polynomial threshold function to have small length but large weight.

The result we now present follows in a straightforward way from the approach of [26], but it does not seem to have appeared previously in the literature:

Theorem 10. *Any length-s polynomial threshold function over $\{0,1\}^n$ can be expressed as a polynomial threshold function of degree $O(\sqrt{n \log s})$.*

By Fact 4 this immediately gives Theorem 1.

Proof of Theorem 10. Let f be a length-s polynomial threshold function. Lemma 10 in [26] gives us the following:

Lemma 11. *Let $f : \{0,1\}^n \to \{-1,1\}$ be computed by a length-s polynomial threshold function. For any value $1 \le t \le n$, f can be expressed as a decision tree T in which*

- *each internal node is labelled with a variable;*
- *each leaf of T contains a polynomial threshold function of degree at most t;*

- *T has rank at most $(2n/t)\ln s + 1$.*

(The actual lemma in [26] is stated for s-term DNF formulas rather than length-s polynomial threshold functions, but the proof in [26] can be trivially modified to establish the variant stated above as well.) Now just as in [26], the result of Blum [5] (mentioned in Section 3.1) shows that there is a generalized r-decision list $(T_1(x), f_1(x)), \ldots, (T_m(x), f_m(x)), f_{m+1}(x)$ that computes f, where each T_i is a conjunction of at most $r = (2n/t)\ln s + 1$ literals and each f_i is a polynomial threshold function of degree at most t (see Section 4 for a precise definition of generalized decision lists). An identical approach to the proof of Theorem 2 of [26] (see also the end of the proof of Theorem 23 in [34]) then directly gives us that f is computed by a polynomial threshold function of degree $r + t = (2n/t)\ln s + 1 + t$. Optimizing the choice of t by taking $t = (n \ln s)^{1/2}$, we obtain the theorem. ■

It is important to note that all the results on weight and length of polynomial threshold functions that we have presented here in Section 3.5 rely crucially on the fact that the domain of our polynomial threshold functions is $\{0,1\}^n$. Another natural choice of domain for polynomial threshold functions is $\{-1,1\}^n$ (see e.g. [38, 34]); since monomials over $\{-1,1\}$-valued inputs are simply parity functions, polynomial threshold functions over $\{-1,1\}^n$ correspond to threshold-of-parity circuits. While the choice of domain does not affect the degree of polynomial threshold functions, it can have a very substantial impact on both the optimal length and weight of polynomial threshold representations for Boolean functions. Indeed, the degree bounds $O(n^{1/3}\log W)$ and $O(\sqrt{n \log s})$ as functions of the weight and length respectively are not true for polynomial threshold functions over $\{-1,1\}^n$. This can be easily seen from the fact that the parity function on n variables has a weight-1, length-1 polynomial threshold function over $\{-1,1\}^n$ which is simply $x_1 x_2 \ldots x_n$.

This motivates the following open question:

Question 12. *Is there a $2^{o(n)}$ time algorithm which can PAC learn polynomial threshold functions of weight poly(n) over the domain $\{-1,1\}^n$?*

4 Another approach: generalized decision lists

4.1 Learning via generalized decision lists

The approach to PAC learning that we describe in this section exploits what we will call *generalized decision lists*. Let C_1 and C_2 be classes of Boolean functions over $\{0,1\}^n$. A *generalized* (C_1, C_2)-*decision list* is defined by a list $(f_1(x), g_1(x)), \ldots, (f_m(x), g_m(x)), g_{m+1}(x)$ where each f_i belongs to C_1 and each g_j belongs to C_2. The value of the decision list on input x is $g_i(x)$ where i is the first index such that f_i is satisfied by x; if x satisfies none of $f_1, \ldots, f_m$ then the output is $g_{m+1}(x)$. We write $\mathrm{DL}(C_1, C_2)$ to denote the class of all generalized (C_1, C_2)-decision lists. Note that standard r-decision lists (mentioned in Section 3.1) are the class $\mathrm{DL}(C_1, C_2)$ in which C_1 is the class of Boolean conjunctions of size at most r and C_2 is the class consisting of the two constant functions 0 and 1.

A *strict width-2 branching program* is a width-2 branching program that contains exactly one sink labeled 0 and exactly one sink labeled 1. Bshouty *et al.* solved the problem of learning $\mathcal{SW}_2$, the class of strict width-2 branching programs, by using an algorithm for learning $\mathrm{DL}(C_1, C_2)$ for a particular C_1 and C_2 which we describe below[12]. However, as we now explain, the algorithm they used can also be applied to a much wider range of classes C_1 and C_2; this is the approach which enables us to prove Theorem 2.

We need the following definition. A representation c of a Boolean function f is *polynomial-time evaluatable* if given an input x, $c(x)$ can be computed in time polynomial in $|x|$ and $|c|$.

We can now present the following Lemma, which is a straightforward generalization of Lemma 4.2 of Bshouty *et al.* [12].

Lemma 13. *Let C_1 and C_2 be Boolean concept classes defined on $\{0,1\}^n$. Suppose there exists a consistency algorithm for C_2 that runs in time poly$(m) \cdot t(n)$ when run on a sample of m labeled examples. Suppose moreover that there exists an algorithm that runs in time polynomial in $|C_1|$ and outputs a list of polynomially-evaluatable representations of all functions in C_1. Then DL(C_1,C_2) can be PAC learned in time poly($|C_1|, \log|C_2|, t(n)$).*

(We note that other generalizations of this lemma are possible in which the consistency algorithm for C_2 is replaced by a PAC learning algorithm; we have chosen to present the most straightforward generalization here which suffices for our purposes.)

Proof of Lemma 13. Like Rivest's original algorithm for learning decision lists, the algorithm for learning generalized decision lists works by drawing a sample S (of size poly$(|C_1|, \log C_2)$) and greedily constructing a generalized decision list consistent with S. In the i-th greedy stage, an appropriate pair (f_i, g_i) with $f_i \in C_1, g_i \in C_2$ is added to the list. As in Rivest's algorithm, finding a candidate f_i requires essentially brute force search among all $c \in C_1$ (hence the need to be able to enumerate C_1). For each candidate f_i, the question is whether there is a valid g_i to go with it; this question is answered by applying the consistency algorithm for C_2 to the subsample of S that would reach this point in the list if f_i were added to it, and then checking whether a consistent $g_i \in C_2$ is found. Determining which examples belong in the subsample requires evaluation of the $f_1, \ldots, f_{i-1}$ already in the list, and of the candidate f_i. ■

4.2 The role of generalized decision lists in previous learning results

Bshouty *et al.* showed that $\mathcal{SW}_2$ is learnable by showing that the class of strict width-2 branching programs over n Boolean variables represents precisely the same functions as the class DL($\oplus_2$,$\oplus_n$). Here $\oplus_k$ denotes the class of all (possibly negated) parity functions over at most k variables, i.e. all functions of the form $(-1)^{x_{i_1}+\cdots+x_{i_k}+b}$ where $j \leq k$, $b \in \{0,1\}$, and addition is performed modulo 2. It is clear that the number of elements in $\oplus_2$ is $O(n^2)$, and it is well known that finding a (possibly negated) parity function consistent with a labeled sample (or determining that none exists) can be done in polynomial time using Gaussian elimination modulo 2, see e.g. [18, 14]. Lemma 13 thus directly implies a polynomial-time PAC algorithm for learning $\mathcal{SW}_2$.

While the algorithm of Lemma 13 is not required for any of the learning results presented in Section 3, we note that generalized decision lists appear within the proofs of a number of those results. The proofs of the DNF learning results begin by converting DNF formulas into generalized decision lists in DL(C_1, C_2), where C_1 is a class of all conjunctions of some bounded length and C_2 is a class of DNF formulas with a bounded number of terms all of which are of bounded length. These functions in DL(C_1, C_2) are subsequently converted into standard r-decision lists (in [10]) or polynomial threshold functions (in [26]). Similarly, the proof of Theorem 10 shows that polynomial threshold functions over the domain $\{0,1\}^n$ can also be represented by elements of DL(C_1, C_2), where C_1 is as above and C_2 is a class of polynomial threshold functions of bounded degree. It is thus possible to learn DNFs and $\{0,1\}^n$ polynomial threshold functions using the generalized decision list algorithm; however, the results cited in Section 3 show that these classes can also be learned simply by using the low-degree polynomial threshold algorithm (and for DNF, learning via the low-degree polynomial threshold procedure yields a better bound than the approach of Lemma 13).

In contrast to DNFs and $\{0,1\}^n$ polynomial threshold functions, functions in $\mathcal{SW}_2$ cannot in general be converted to equivalent low-degree polynomial threshold functions. This is because the parity function on n variables is computed by a branching program in $\mathcal{SW}_2$, and as mentioned earlier, this function requires a polynomial threshold function of degree n. In the next section we present another concept class – s-sparse GF_2 polynomials – for which the approach of Lemma 13 yields the only known subexponential time learning algorithm.

4.3 Learning sparse GF_2 polynomials

In this section it is convenient for us to view Boolean functions as having range $\{0,1\}$ rather than $\{-1,1\}$. It is well known that every Boolean function $f : \{0,1\}^n \to \{0,1\}$ has a unique representation as a multilinear *GF_2 polynomial*; this is simply a sum modulo 2 of monomials over variables $x_1, \ldots, x_n$. A GF_2 polynomial is *s-sparse* if it is a sum of at most s monomials; equivalently such a polynomial may be viewed as a parity of at most s monotone conjunctions.

Theorem 2, presented in the introduction, states that s-sparse GF_2 polynomials can be PAC learned in subexponential time. We now present the proof, which uses the generalized decision list algorithm.

Proof of Theorem 2. The proof is nearly identical to the proof of Theorem 1, except that we do not convert the generalized decision list into a polynomial threshold function. The proof of Lemma 11 can be directly applied to s-sparse GF_2 polynomials rather than to length-s polynomial threshold functions, yielding a restatement of Lemma 11 for s-sparse GF_2 polynomials. Again, the decision tree specified by the lemma can be converted into a generalized decision list of the form $(T_1(x), f_1(x)), ..., (T_m(x), f_m(x)), f_{m+1}$, where each T_i is a conjunction of bounded size and now each f_i is a GF_2 polynomial of bounded degree. More precisely, we have the structural result that every s-sparse GF_2 polynomial is equivalent to some member of $\mathrm{DL}(C_1, C_2)$, where C_1 is the class of conjunctions of degree at most $r = (2n/t)\ln s + 1$, and C_2 is the class of GF_2 polynomials of degree at most t.

As noted in the previous subsection, Gaussian elimination over GF_2 can be used to either find a possibly negated parity function (i.e. a GF_2 polynomial of degree 1) that is consistent with a labeled sample or show that none exists. Applying this procedure over an expanded feature space of all (at most $O(n^d)$ many) monomials of degree at most d yields a consistency algorithm for the class of degree-d GF_2 polynomials that runs in time $\mathrm{poly}(m, n^d)$. There are $n^{O(r)}$ many monomials of degree at most r, and they can be easily enumerated. Taking $t = (n \ln s)^{1/2}$ and applying the generalized decision list algorithm yields the theorem. ■

Since the n-variable parity function can be represented by an n-sparse GF_2 polynomial, s-sparse GF_2 polynomials, like $\mathcal{SW}_2$ functions, do not all have equivalent low-degree $\{0,1\}^n$ polynomial threshold functions.

Finally, we note that functions in $\mathcal{SW}_2$ and s-sparse GF_2 polynomials cannot in general be expressed by low-degree GF_2 polynomials. Both $\mathcal{SW}_2$ and 1-sparse GF_2 polynomials include the n-variable AND function $x_1 \wedge \cdots \wedge x_n$, and the unique GF_2 polynomial representing this function, which consists of the single monomial $x_1 \cdots x_n$, has degree n. The generalized decision list approach is currently the only approach we know that yields a polynomial-time algorithm for $\mathcal{SW}_2$ or a subexponential algorithm for s-sparse GF_2 polynomials.

5 Acknowledgement

We thank Adam Klivans for helpful suggestions in preparing this survey, and we thank an anonymous referee for useful suggestions regarding the presentation.

References

[1] D. Angluin. Learning Regular Sets from Queries and Counterexamples. *Information and Computation*, 75(2):87–106, 1987.

[2] R. Beigel, N. Reingold, and D. Spielman. PP is closed under intersection. *Journal of Computer and System Sciences*, 50(2):191–202, 1995.

[3] A. Beimel, F. Bergadano, N. Bshouty, E. Kushilevitz, and S. Varricchio. Learning functions represented as multiplicity automata. *J. ACM*, 47(3):506–530, 2000.

[4] S. Ben-David and E. Dichterman. Learning with restricted focus of attention. *Journal of Computer and System Sciences*, 56(3):277–298, 1998.

[5] A. Blum. Rank-r decision trees are a subclass of r-decision lists. *Information Processing Letters*, 42(4):183–185, 1992.

[6] A. Blum, P. Chalasani, and J. Jackson. On learning embedded symmetric concepts. In *Proceedings of the Sixth Annual Conference on Computational Learning Theory*, pages 337–346, 1993.

[7] A. Blumer, A. Ehrenfeucht, D. Haussler, and M. Warmuth. Occam's razor. *Information Processing Letters*, 24:377–380, 1987.

[8] A. Blumer, A. Ehrenfeucht, D. Haussler, and M. Warmuth. Learnability and the Vapnik-Chervonenkis dimension. *Journal of the ACM*, 36(4):929–965, 1989.

[9] N. Bshouty. Exact learning via the monotone theory. *Information and Computation*, 123(1):146–153, 1995.

[10] N. Bshouty. A subexponential exact learning algorithm for DNF using equivalence queries. *Information Processing Letters*, 59:37–39, 1996.

[11] N. Bshouty and C. Tamon. On the Fourier spectrum of monotone functions. *Journal of the ACM*, 43(4):747–770, 1996.

[12] N. Bshouty, C. Tamon, and D. Wilson. On learning width two branching programs. *Information Processing Letters*, 65:217–222, 1998.

[13] A. Ehrenfeucht and D. Haussler. Learning decision trees from random examples. *Information and Computation*, 82(3):231–246, 1989.

[14] P. Fischer and H.U. Simon. On learning ring-sum expansions. *SIAM Journal on Computing*, 21(1):181–192, 1992.

[15] Y. Freund. Boosting a weak learning algorithm by majority. *Information and Computation*, 121(2):256–285, 1995.

[16] Y. Freund and R. Schapire. A decision-theoretic generalization of on-line learning and an application to boosting. *Journal of Computer and System Sciences*, 55(1):119–139, 1997.

[17] J. Håstad. On the size of weights for threshold gates. *SIAM Journal on Discrete Mathematics*, 7(3):484–492, 1994.

[18] D. Helmbold, R. Sloan, and M. Warmuth. Learning integer lattices. *SIAM Journal on Computing*, 21(2):240–266, 1992.

[19] J. Jackson. An efficient membership-query algorithm for learning DNF with respect to the uniform distribution. *Journal of Computer and System Sciences*, 55:414–440, 1997.

[20] J. Jackson, A. Klivans, and R. Servedio. Learnability beyond AC^0. In *Proceedings of the 34th ACM Symposium on Theory of Computing*, pages 776–784, 2002.

[21] M. Kearns. Efficient noise-tolerant learning from statistical queries. *Journal of the ACM*, 45(6):983–1006, 1998.

[22] M. Kearns and U. Vazirani. *An introduction to computational learning theory.* MIT Press, Cambridge, MA, 1994.

[23] L. Khachiyan. A polynomial algorithm in linear programming. *Soviet Math. Dokl*, 20:1093–1096, 1979.

[24] M. Kharitonov. Cryptographic hardness of distribution-specific learning. In *Proceedings of the Twenty-Fifth Annual Symposium on Theory of Computing*, pages 372–381, 1993.

[25] A. Klivans, R. O'Donnell, and R. Servedio. Learning intersections and thresholds of halfspaces. *Journal of Computer & System Sciences*, 68(4):808–840, 2004. Preliminary version in *Proc. of FOCS'02*.

[26] A. Klivans and R. Servedio. Learning DNF in time $2^{\tilde{O}(n^{1/3})}$. *Journal of Computer & System Sciences*, 68(2):303–318, 2004. Preliminary version in *Proc. STOC'01.*

[27] M. Krause and P. Pudlak. Computing boolean functions by polynomials and threshold circuits. *Computational Complexity*, 7(4):346–370, 1998.

[28] E. Kushilevitz and Y. Mansour. Learning decision trees using the Fourier spectrum. *SIAM J. on Computing*, 22(6):1331–1348, 1993.

[29] N. Linial, Y. Mansour, and N. Nisan. Constant depth circuits, Fourier transform and learnability. *Journal of the ACM*, 40(3):607–620, 1993.

[30] N. Linial and N. Nisan. Approximate inclusion-exclusion. *Combinatorica*, 10(4):349–365, 1990.

[31] M. Minsky and S. Papert. *Perceptrons: an introduction to computational geometry.* MIT Press, Cambridge, MA, 1968.

[32] S. Muroga. *Threshold logic and its applications.* Wiley-Interscience, New York, 1971.

[33] S. Muroga, I. Toda, and S. Takasu. Theory of majority switching elements. *J. Franklin Institute*, 271:376–418, 1961.

[34] R. O'Donnell and R. Servedio. Extremal properties of polynomial threshold functions. In *Proceedings of the Eighteenth Annual IEEE Conference on Computational Complexity*, pages 325–334, 2003.

[35] R. O'Donnell and R. Servedio. New degree bounds for polynomial threshold functions. In *Proceedings of the 35th ACM Symposium on Theory of Computing*, pages 325–334, 2003.

[36] R. O'Donnell and R. Servedio. Learning monotone decision trees in polynomial time. In *Proceedings of the 21st Conference on Computational Complexity (CCC)*, pages 213–225, 2006.

[37] R. Rivest. Learning decision lists. *Machine Learning*, 2(3):229–246, 1987.

[38] M. Saks. *Slicing the hypercube*, pages 211–257. London Mathematical Society Lecture Note Series 187, 1993.

[39] J. Shawe-Taylor and N. Cristianini. *An introduction to support vector machines.* Cambridge University Press, 2000.

[40] J. Tarui and T. Tsukiji. Learning DNF by approximating inclusion-exclusion formulae. In *Proceedings of the Fourteenth Conference on Computational Complexity*, pages 215–220, 1999.

[41] L. Valiant. A theory of the learnable. *Communications of the ACM*, 27(11):1134–1142, 1984.

[42] K. Verbeurgt. Learning DNF under the uniform distribution in quasi-polynomial time. In *Proceedings of the Third Annual Workshop on Computational Learning Theory*, pages 314–326, 1990.