least $\Omega(\frac{d}{\epsilon})$ examples to see $\frac{d}{\epsilon}$ of the examples in $S'$. This is because we want $\frac{d}{\epsilon}$ successful Bernoulli trials, each with $\frac{d}{\epsilon^2}$ chance of success.

Thus combining this with the first part of the proof, we get that the PAC algorithm needs

$$\Omega \left( \frac{1}{\epsilon} \ln \frac{1}{\delta} + \frac{\text{VCD}(C)}{\epsilon} \right)$$

examples to find an $\epsilon$-good hypothesis with probability at least $(1 - \delta)$. 

7.5 Some Relations on the VC Dimension

In this section, we will see some inequalities describing the relationships between the VC dimension of two or more concept classes, the relationships between the VC dimension of a concept class and the size of the concept class, etc. These bounds are very useful to compute the VC dimension of complicated concept classes that are constructed with simpler concept classes for which the VC dimension is known.

If $C_1 \subseteq C_2$, then clearly

$$\text{VCD}(C_1) \leq \text{VCD}(C_2).$$

Next we consider the relation between the VC dimension of $C$ and the cardinality of $C$. Given $\text{VCD}(C)$ points, we need $2^{\text{VCD}(C)}$ concepts to shatter them. Thus,

$$|C| \geq 2^{\text{VCD}(C)}$$

Taking the base-2 logarithm of both sides

$$\log_2 |C| \geq \text{VCD}(C)$$

For finite $C$, we had earlier shown that

$$m \geq \frac{1}{\epsilon} \left( \ln |C| + \ln \frac{1}{\delta} \right)$$

examples were sufficient to learn $C$. Note that using $\text{VCD}(C)$ instead of $\ln |C|$ gives a better bound.

Consider the complement $\tilde{C}$ of the concept class $C$, defined as $\tilde{C} = \{ X - c : c \in C \}$. Then

$$\text{VCD}(\tilde{C}) = \text{VCD}(C).$$

Consider the union of two concept classes. That is, let $C = C_1 \cup C_2$ where $C$, $C_1$ and $C_2$ are defined over the same instance space. Then,

$$\text{VCD}(C) \leq \text{VCD}(C_1) + \text{VCD}(C_2) + 1.$$
Consider the concept class \( \mathcal{C}_s \) formed by the union (or intersection) of up to \( s \) concepts from \( \mathcal{C} \) where \( \text{VCD}(\mathcal{C}) = d \). For the union, \( \mathcal{C}_s = \bigcup_{i=1}^{d} c_i : c_i \in \mathcal{C} \), and similarly for intersection. Then, for all \( s \geq 1, \text{VCD}(\mathcal{C}_s) \leq 2ds \lg(3s) \). Thus,

\[
\text{VCD}(\mathcal{C}_s) = O(s \ln(s) \text{VCD}(\mathcal{C})).
\]

Earlier, we have defined \( \Pi_{\mathcal{C}}(S) \) as the distinct ways that \( \mathcal{C} \) can classify the instances in \( S \). We shall now bound \( |\Pi_{\mathcal{C}}(S)| \) by \( |S| \) and \( \text{VCD}(\mathcal{C}) \).

**Definition 7.7** For all \( d \geq 0, m \geq 0, \)

\[
\Phi_d(m) = \begin{cases} 
\sum_{i=0}^{d} \binom{m}{i} & \text{if } m \geq d \\
2^m & \text{otherwise}
\end{cases}
\]

For concept class \( \mathcal{C} \), let \( \text{VCD}(\mathcal{C}) = d \) and let \( S \) be a set of \( m \) distinct instances, then

\[
|\Pi_{\mathcal{C}}(S)| \leq \Phi_d(m).
\]

There are some inequalities relating \( \Phi_d(m) \) using combinatorial arguments and Stirling’s approximation.

\[
\Phi_d(m) = \Phi_{d-1}(m-1) + \Phi_d(m-1).
\]

For all \( d \geq 0 \) and \( m \geq 0, \)

\[
\Phi_d(m) \leq m^d + 1.
\]

For all \( d \geq 2 \) and \( m \geq 2, \)

\[
\Phi_d(m) \leq m^d.
\]

For all \( m \geq d \geq 1, \)

\[
\Phi_d(m) \leq 2 \left( \frac{m^d}{d!} \right) \leq \left( \frac{em}{d} \right)^d.
\]

Thus, whenever \( \text{VCD}(\mathcal{C}) \) is finite, then \( \Pi_{\mathcal{C}}(m) \) grows only polynomially in \( m \).
8.1 Introduction

In Topic 3 we discussed a hardness result for learning $k$-term-DNF when the learner is restricted to use a hypothesis class of $k$-term-DNF. However, we then showed that $k$-term-DNF is learnable using the hypothesis class of $k$-CNF. While such representation-dependent hardness results provide some information, what one would really like to obtain is a hardness result for learning a concept class using any reasonable (i.e. polynomially evaluable) hypothesis class. In this lecture we will briefly introduce a representation-independent hardness result for learning several simple concept classes such as Boolean formulas, deterministic finite automata, and constant-depth threshold circuits (a simplified form of "neural networks"). These hardness results are based on assumptions regarding the intractability of various cryptographic schemes such as factoring Blum integers and breaking the RSA function. The material presented here is just a brief introduction to the paper, "Cryptographic Limitations on Learning Boolean Formulae and Finite Automata," by Michael Kearns and Leslie Valiant [26]. We only give the intuition behind the hardness results—no details are described here. For the complete proofs we refer to reader to the Kearns and Valiant paper. Also a more complete discussion of these results is contained in Chapter 7 of Kearns' thesis [27].

8.2 Previous Work

Before describing the results of Kearns and Valiant we first briefly review the only previously known representation-independent hardness result. This previous result follows from the work of Goldreich, Goldwasser and Micali [19]. Let $\text{CKT}^p(n)$ denote the class of Boolean circuits over $n$ inputs with at most $p(n)$ gates and let $\text{CKT}^p(n) = \bigcup_{n \geq 1} \text{CKT}^p(n)$. Goldreich, Goldwasser, and Micali showed that if there exists a one-way function, then for some polynomial $p(n)$, $\text{CKT}^p(n)$ is not polynomially learnable by any polynomially evaluable hypothesis class.

Observe that the most powerful (polynomially evaluable) hypothesis that can be output by any polynomial-time learning algorithm is a hypothesis that is itself a polynomial-time algorithm (or equivalently a polynomial-size Boolean circuit). Thus we cannot find efficient learning algorithms for concept classes that do not have small circuits—no learning algorithm even has time to just write down the representation. Schapire [37] has formally shown that any representation class that is not polynomially evaluable cannot be learned in
polynomial time. With this in mind, observe that the result of Goldreich, et al. shows that not everything with a small representation is efficiently learnable (assuming the existence of one-way functions). However, there is a large gap between the computational power of $\mathcal{CT}P^{(s)}$ and the classes for which we have efficient learning algorithms. Thus we would like to prove representation-independent hardness results for less powerful concept class such as the classes of Boolean formulas.

### 8.3 Intuition

In this section we describe the intuition behind the hardness results of Kearns and Valiant. We begin by very informally describing the type of cryptographic scheme on which this hardness result is based.

Suppose that Alice and Bob wish to communicate privately inspite of an eavesdropper Eve who has bugged the communication line between Alice and Bob. (We make no assumptions about Eve except that she has a polynomial bound on her computing resources.) Furthermore, we want a scheme which does not require Alice and Bob to meet privately ahead of time to set up their encoding scheme. Such encryption scheme can be devised using a *trapdoor function*. Informally, a trapdoor function is one that can be computed in polynomial time (i.e., it is easy to compute $f(x)$ on input $x$) but cannot be inverted in polynomial time (i.e., it is hard to compute $x$ on input $f(x)$) unless one is the “creator” of the function and thus possesses a piece of “trapdoor” information that makes inversion possible in polynomial time.

Cryptographic schemes based on such trapdoor functions are known as *public-key cryptographic systems*. In such a scheme each user creates a trapdoor function $f$ and publishes a program for computing $f$. (This program must reveal no information about $f^{-1}$.) Then anyone can send messages to the given user over the nonsecure communication line and only that user can decode such messages. So Alice and Bob can communicate with each other using such a scheme where they have both created their own trapdoor function. We say that this system is secure if Eve cannot do noticeably better than random guessing in trying to decode a bit that has been encoded using this encryption scheme. More formally, if

$$\Pr[\text{Eve predicts correct decoded bit}] \geq \frac{1}{2} + \frac{1}{p(n)}$$

for some polynomial $p(n)$ then the system is not secure.

We now describe how we can use the existence of such a public-key cryptographic system, such as RSA, to obtain a representation-independent hardness result for learning. For the hardness result view Eve as a learning algorithm. Since a program for $f$ is available she can create any polynomially number of pairs of the form $(f(x), x)$ that she likes by simply choosing $x$ and then computing $f(x)$. If we set $y = f(x)$ observe that such pairs have the form $(y, f^{-1}(y))$ and thus can be viewed as labeled examples of $f^{-1}$. Thus public-key cryptography assumes the existence of functions that are not learnable from examples. In fact, unlike the PAC learning model in which we ask the learner to be able to make
arbitrarily good predictions, in the cryptographic scheme we only ask the learner, Eve, to make predictions that are noticeably better than random guessing. (This "weak learning" model will be discussed further in the next topic.)

Observe that in the learning problem described above \( f^{-1} \) is "simple" in the sense that it has a small circuit (determined by the trapdoor used for decoding.) So the theory of cryptography provides simple functions that are difficult to learn. Kearns and Valiant show how to refine the functions provided by cryptography to find the simplest functions that are difficult to learn. Using this basic approach (of course, with lots of details which we will not discuss here) they show that polynomial-sized Boolean formulas, and constant-depth threshold circuits are not even weakly learnable by any polynomially evaluable hypothesis class. Then using a prediction-preserving reduction (described below) of Pitt and Warmuth [34] it follows that deterministic finite automata are also not learnable.

8.4 Prediction Preserving Reductions

Given that we now have a representation-independent hardness result (assuming the security of the various cryptographic schemes) one would like a "simple" way to prove that other problems are hard in a similar fashion as one proves a desired algorithm is intractable by reducing a known NP-complete problem to it. Such a complexity theory for predictability has been provided by Pitt and Warmuth [34]. They formally define a prediction-preserving reduction \( (A \leq B) \) that enables a prediction algorithm for concepts of type \( B \) to solve a prediction problem for concepts of type \( A \). This reduction consists of the following two mapping:

1. A polynomial time computable function \( f \) that maps unlabeled examples of \( A \) to unlabeled examples of \( B \).

2. A function \( g \) that maps representations of \( A \) to representations of \( B \). Furthermore, this mapping \( g \) need not be computable—it is only required to be length preserving within a polynomial.

Then given a polynomial prediction algorithm for concepts of type \( B \) one can use it to obtain a polynomial prediction algorithm for concepts of type \( A \) in the obvious manner. Thus if \( A \) is known not to be learnable then \( B \) also cannot be learnable. They describe several such reductions in their paper, one of which is a reduction from Boolean formula predictability to DFA predictability (Boolean formula \( \leq \) DFA). Thus since Kearns and Valiant have shown that Boolean formula are not predictable (under cryptographic assumptions) it immediately follows that DFA are not predictable.


9.1 Introduction

In the first lecture we introduced the probably approximately correct (PAC) or distribution free model of learning. In this model, the learner is presented with examples which are randomly and independently drawn from an unknown but fixed distribution. The learner must, with arbitrarily high probability, produce a hypothesis that is arbitrarily close to the target concept.

As we saw in the last lecture, the representation-independent hardness results based on the security of various cryptographic schemes motivated the study of learning algorithms which do not attain arbitrarily high accuracy, but do just slightly better than random guessing (by an inverse polynomial in the size of the problem). This leads to the notion of weak learning which is the subject of this lecture. In particular we describe an algorithm to convert any weak learning algorithm into a PAC-learning algorithm. The material presented here comes from the paper, “The Strength of Weak Learnability,” by Robert Schapire [37].

9.2 Preliminaries

We begin by formally defining the weak learning model. As in the PAC model, for a given size parameter $n$, there is a set of instances $X_n$. A concept $c$ is a Boolean function $c : X_n \rightarrow \{0,1\}$. A concept class $C_n \subseteq 2^{X_n}$ is a set of concepts. The learner has access to a source $EX$ of labeled examples. Each time $EX$ is called, an example is drawn randomly and independently according to a fixed but unknown distribution $D$ on $X_n$, and returned in unit time.

After drawing some examples and running for a time, the learning algorithm must output an hypothesis $h$. This is a polynomially (in $n$) evaluable hypothesis which, when given an instance $v \in X_n$, returns a prediction. The hypothesis may be randomized (this is important for weak learning).

We write $\text{Prob}[\pi(v)]$ to denote the probability that the predicate $\pi$ holds for a particular instance $v$. This probability may be between 0 and 1 because of randomness in evaluating $\pi(v)$. For example, think of $\pi(v)$ as the event that $h(v) = 1$ for some randomized hypothesis $h$. By $\text{Prob}_{v \in D}[\pi(v)]$ we denote the probability that, after drawing $v$ randomly from distribution $D$, $\pi(v)$ holds. Thus, assuming that these two random events (i.e., choice of $v$ and evaluation of $\pi(v)$) are independent, we have $\text{Prob}_{v \in D}[\pi(v)] = \sum_{v \in X_n} D(v) \text{Prob}[\pi(v)]$ where $D(v)$ denotes the probability of instance $v$ being chosen under distribution $D$. 
Finally, we can define the error of an hypothesis \( h \) on \( c \) under distribution \( D \) as

\[
\text{error}_D(h) = \text{Prob}_{v \in D}[h(v) \neq c(v)].
\]

If the \( \text{error}_D(h) \leq \epsilon \), we say \( h \) is \( \epsilon \)-close to \( c \) under \( D \). The accuracy of \( h \) is one minus its error.

We say that a concept class \( C \) is strongly learnable (called simply learnable or PAC-learnable elsewhere) if there is an algorithm \( A \) such that, for all \( n \geq 1 \), for all target concepts \( c \in C_n \), for all distributions \( D \) and parameters \( 0 < \delta, \epsilon \leq 1 \), algorithm \( A \), given \( n, \delta, \epsilon \), and access to oracle \( EX \), outputs a hypothesis \( h \) such that, with probability \( \geq 1 - \delta \), \( \text{error}_D(h) \) is at most \( \epsilon \). Algorithm \( A \) should run in time polynomial in \( n, 1/\epsilon \), and \( 1/\delta \).

To be weakly learnable, there must be a polynomial \( p \) in addition to the algorithm \( A \) such that, for all \( n \geq 1 \), \( c \in C_n \), for all distributions \( D \), and for all \( 0 < \delta \leq 1 \), algorithm \( A \), given \( n, \delta \), and access to oracle \( EX \), outputs a hypothesis \( h \) such that, with probability \( \geq 1 - \delta \), \( \text{error}_D(h) \) is at most \( (1/2 - 1/p(n)) \). Algorithm \( A \) should run in time polynomial in \( n \) and \( 1/\delta \).

### 9.3 The Equivalence of Strong and Weak Learning

It should be clear that if \( C \) is strongly learnable, then it is weakly learnable—just fix \( \epsilon = 1/4 \) (or any constant less than \( 1/2 \)). The converse (weak learnability implying strong learnability) is not at all obvious. In fact, if one restricts the distributions under which the weak learning algorithm runs then weak learnability does not imply strong learnability. In particular, Kearns and Valiant [26] have shown that under a uniform distribution monotone Boolean functions are weakly, but not strongly, learnable. Thus it will be important to take advantage of the requirement that the weak learning algorithm must work for all distributions. The remainder of this section will be devoted to proving the main result of these notes.

**Theorem 9.1** If concept class \( C \) is weakly learnable, then it is strongly learnable.

Note that weak learning, even though it may be "weak", is not necessarily easy to do. Consider the simple-minded algorithm which draws and memorizes a polynomial \( q(n, 1/\delta) \) number of examples. It then outputs a hypothesis \( h \) which looks up known results and otherwise flips a fair coin to determine the answer. Suppose that \( |X_n| = 2^n \) and the distribution \( D \) is the uniform distribution. Then the \( \text{error}_D(h) = (1/2 - q(n, 1/\delta)/2^n) \), which is larger than \( (1/2 - 1/p(n)) \) for any polynomial \( p \). Thus it is non-trivial to devise an algorithm that weakly learns a concept class.

### 9.3.1 Hypothesis Boosting

Proving that weak learnability implies strong learnability has also been called the hypothesis boosting problem, because a way must be found to boost the accuracy of slightly-better-than-half hypotheses to be arbitrarily close to 1.
Suppose that we have a learning algorithm \( A \) which produces hypotheses with error at most \( \alpha \) (for \( \alpha < 1/2 \)). Simply running \( A \) twice to produce hypotheses \( h_1 \) and \( h_2 \) may not help at all—they may err in a very similar way in their classification. After obtaining \( h_1 \), we need a way to force the next hypothesis output to be more accurate. The answer lies in the power of \( A \) to work given any distribution \( D \) on the instances. If we give \( A \) a distribution \( D' \) such that the error of \( h_1 \) is exactly 1/2, then the hypothesis \( h_2 \) produced must have an error \( \leq \alpha \) on it, which means that \( h_2 \) has learned something about the target concept which \( h_1 \) did not. Unfortunately, the probability that \( h_1 \) and \( h_2 \) classify a random instance the same (their “overlap”) may be very low; in this case, it would seem that \( h_1 \) and \( h_2 \) learned almost completely different parts of the \( D \). To handle this, we ask \( A \) to learn a “tie-breaker” hypothesis \( h_3 \). This time, \( A \) is given a distribution \( D'' \) on the instances on which \( h_1 \) and \( h_2 \) disagree.

Thus to improve the accuracy of the weak learning algorithm \( A \), the algorithm \( A' \) simulates \( A \) on distributions \( D, D' \), and \( D'' \) and outputs a hypothesis that takes the majority vote of the three hypotheses obtained. Of course, algorithm \( A' \) must use \( D \) to simulate the distributions \( D' \) and \( D'' \). We now describe how this task is achieved. These two modified distributions are created by filtering the original distribution. In other words, \( A' \) will draw samples from \( D \) and discard undesirable samples to obtain the desired distribution. While this requires additional samples (thus increasing both the time and sample complexity) it can be made to run in polynomial time. The key observation is that we will only apply the filter if the probability of obtaining an example which passes through the filter is \( \Omega(\epsilon) \). The distribution \( D' \) given to learn \( h_2 \) is produced by the procedure \( EX_2 \) defined below. Likewise, the distribution \( D'' \) for learning \( h_3 \) is produced by \( EX_3 \).

\[
EX_2(EX, h_1)
\]
flip fair coin
if heads then return the first instance \( v \) from \( EX \) for which \( h_1(v) = c(v) \)
else return the first instance \( v \) from \( EX \) for which \( h_1(v) \neq c(v) \)

\[
EX_3(EX, h_1, h_2)
\]
return the first instance \( v \) from \( EX \) for which \( h_1(v) \neq h_2(v) \)

Thus we can now formally state the hypothesis boosting procedure as follows:

1. Run the weak learning algorithm using \( EX \). Let \( h_1 \) be the hypothesis output.
2. Run the weak learning algorithm using \( EX_2(EX, h_1) \). Let \( h_2 \) be the hypothesis output.
3. Run the weak learning algorithm using \( EX_2(EX, h_1, h_2) \). Let \( h_3 \) be the hypothesis output.
4. Return \( h = \text{Majority}(h_1, h_2, h_3) \).

An important question to address is: How much better is the accuracy of \( h \) than the accuracy of \( h_1 \)? Suppose that the given weak learning algorithm is guaranteed to output a
\( \alpha \)-close hypothesis. (So \( \text{error}_D(h_1) \leq \alpha \).) We shall show that for the hypothesis \( h \) output by the above hypothesis boosting procedure, \( \text{error}_D(h) \leq g(\alpha) = 3\alpha^2 - 2\alpha^3 \). See Figure 9.1 for a graph of \( g(\alpha) \). Observe that both 0 and 1/2 are fixed points for the function \( g \). Thus, as must be the case, for this boosting procedure to improve the error of the hypothesis, \( \alpha < 1/2 \). Note that for small \( \alpha \), \( \text{error}_D(h) = O(\alpha^2) \ll \alpha \) and thus the new hypothesis is much better than the initial hypothesis. On the other hand, as \( \alpha \) nears 1/2 the improvement is not as significant. However, as we shall show the number of iterations required to improve the error to be at most \( \epsilon \) is polynomial in \( n \) and 1/\( \epsilon \).

### 9.3.2 The Learning Algorithm

In this section we describe the complete learning algorithm for converting a weak learning algorithm into a strong learning algorithm. The algorithm is shown in Figure 9.2. The basic idea is to recursively boost the hypothesis. The lowest level of recursion only requires the accuracy that \texttt{WeakLearn} (the weak learning algorithm given) can provide. Each higher level produces hypotheses with higher accuracy, until it has been boosted to be at least 1 - \( \epsilon \).

A technical difficulty which arises is that we may get lucky early and obtain an \( \epsilon \)-close hypothesis for \( h_1 \) or \( h_2 \). If this happens, we are in trouble since either \( EX_2 \) or \( EX_3 \) would take too long. Therefore, the algorithm performs some hypothesis testing to prevent this from happening.

### 9.3.3 Correctness

We now prove that the algorithm of Figure 9.2 will produce a hypothesis meeting the PAC criterion.
Learn(ε, δ, EX)

Input:
- ε - allowed error
- δ - confidence desired
- EX - oracle for examples
- n - (implicit) size

Output:
- h - hypothesis that with probability ≥ 1 − δ is ε-close to target

Procedure:
- if ε ≥ (1/2 − 1/p(n)) then return WeakLearn(δ, EX)
- else α := g⁻¹(ε)
  - h₁ := Learn(α, δ/5, EX)
  - estimate error_D(h₁) to within ε/3 of true value with confidence ≥ 1 − δ/5
  - if h₁ is ε-good then return h₁
  - else construct EX₂
    - h₂ := Learn(α, δ/5, EX₂)
    - estimate error_D(h₂) to within (1 − 2α)ε/8 of true value with confidence ≥ 1 − δ/5
    - if h₂ is ε-good then return h₂
    - else construct EX₃
      - h₃ := Learn(α, δ/5, EX₃)
      - return h = MAJ(h₁, h₂, h₃)
  - endif
- endif

Figure 9.2: An algorithm to convert a weak learning algorithm into a strong learning algorithm.
Theorem 9.2 If $0 < \epsilon < 1/2$ and $0 < \delta \leq 1$, then with probability $\geq 1 - \delta$, the hypothesis returned by calling Learn($\epsilon, \delta, EX$) is $\epsilon$-close to the target $c$ under $D$.

Proof: Define a good run of Learn to be one in which everything goes right. That is, every call of WeakLearn produces a hypothesis with the required accuracy, and every error estimation is within its prescribed tolerance of being correct. We first show that the probability of having a good run is $\geq 1 - \delta$. We then show that whenever there is a good run, the hypothesis returned is $\epsilon$-close to the target. These two facts imply the theorem.

We now argue by induction on the depth of the recursion that the probability of having a good run is at least $1 - \delta$. The base case is depth = 0, which occurs when $\epsilon \geq (1/2 - 1/p(n))$. In this case, the allowed error is high enough that WeakLearn can handle the situation. It produces a good run with probability $\geq 1 - \delta$. The induction step occurs if $\epsilon < (1/2 - 1/p(n))$. In this case, there are (at most) three recursive calls to Learn and (at most) two error estimations. In the worst case, all of the calls are performed. Since each of the calls and estimations succeeds (independently) with probability $\geq 1 - \delta/5$ (the calls succeed with this probability by the induction hypothesis), the probability that they all succeed is $\geq 1 - \delta$.

From now on, we assume that we have a good run, and our goal is to prove that the returned hypothesis is $\epsilon$-good. We again proceed by induction on the depth of recursion. The base case is trivial, assuming a good run, since the call to WeakLearn produces an $\epsilon$-good hypothesis.

The inductive step is trickier. In his paper Schapire gives a very detailed and formal proof—what follows here is a simpler but less formal (i.e. important details are ignored) proof.

If either $h_1$ or $h_2$ is tested to be $\epsilon$-good, then it is returned and the Theorem is proved. If $h = MAJ(h_1, h_2, h_3)$, then there are two cases for $h$ to be incorrect.

1. Both $h_1$ and $h_2$ give the wrong answer. Since this is a good run, \(error_{D'}(h_2) \leq \alpha\). (Recall that $D'$ is the distribution of examples given by EX$_2$). To make the probability of $h_1 = h_2 \neq c$ as high as possible, all of this error should coincide with $h_1$'s error. With the filtering that occurs to get from $D$ to $D'$, the portion on which $h_1$ is incorrect expands from probability $\alpha$ to $1/2$. If $h_2$ has all of its error on that $1/2$ portion, it shrinks by multiplying it by $2\alpha$ when translated back into $D$. Therefore $h_2$'s error on $D$ is $\leq 2\alpha(\alpha) = 2\alpha^2$.

2. The hypotheses $h_1$ and $h_2$ give different answers, and $h_3$ breaks the tie incorrectly. Since $h_3$ is the deciding vote, it should be wrong as often as possible to make the overall error large. Therefore we should make the distribution $D''$ (produced by EX$_3$) "cover" as much of the original distribution $D$ as possible. This is done by making $h_1$ and $h_2$ disagree as often as possible. So, make $h_2$ wrong on the portion $\alpha$ of $D'$ on which $h_1$ is correct. When translating back to $D$, this portion expands by a factor of $2(1 - \alpha)$. Therefore, $h_2$'s error on $D$ is $\leq 2(1 - \alpha)\alpha = 2\alpha^2$, and the portion of $D$ on which exactly one of $h_1$ and $h_2$ is wrong is $\leq 2\alpha - 2\alpha^2 + \alpha = 3\alpha - 2\alpha^2$. Thus $h_3$ can be wrong on a fraction $\alpha$ of that, so the error of $h_3$ on $D$ is $\leq 3\alpha^2 - 2\alpha^3$. 

66
The above two cases are mutually exclusive, and they exhaust all possibilities that \( h \) is wrong. Case (1) covers when \( h_1 \) and \( h_2 \) agree as often as possible, and case (2) covers when they disagree as often as possible. Thus \( h \)'s real error will be somewhere in between these values. The larger value (when \( 0 \leq \alpha \leq 1/2 \)) is \( g(\alpha) = 3\alpha^2 - 2\alpha^3 = \epsilon \), proving the theorem.

\[ \blacksquare \]

### 9.3.4 Analysis of Time and Sample Complexity

Although in the previous section we proved that \( \text{Learn} \) is correct, we have said nothing about its time or sample complexity. To complete the proof that strong and weak learning are equivalent, we must prove that \( \text{Learn} \) has time and sample complexity that are polynomial in \( n, 1/\epsilon \), and \( 1/\delta \).

First, consider the shape of the recursion tree obtained by the recursive calls occurring during \( \text{Learn} \)'s execution. If there are no early returns caused by serendipitously discovering an \( \epsilon \)-good hypothesis, then the recursion tree is a rooted full ternary tree. Every node has either zero or three children, and every leaf occurs at the same depth. Denote this depth by \( B(\epsilon, p(n)) \), since it depends only on the arguments given.

If early returns do occur, then a node at depth less than \( B \) will have either one or two children and their corresponding subtrees "clipped". It is simpler to ignore such cases when determining worst-case time complexity, since the run time is greatest when there is no clipping.

The number of leaves (which correspond with calls to \( \text{WeakLearn} \)) is exactly \( 3^B \). Therefore we would like to bound \( B \) to be logarithmic in \( n, 1/\epsilon \), and \( 1/\delta \). That is the result of Lemma 9.1 below.

At each level of recursion, \( \epsilon \) is replaced by \( g^{-1}(\epsilon) \), with \( \epsilon \geq (1/2 - 1/p(n)) \) at the bottom level. Therefore

\[
B(\epsilon, p(n)) = \min_{i \geq 0} \left\{ g^i \left( \frac{1}{2} - \frac{1}{p(n)} \right) \leq \epsilon \right\}.
\]

Note that \( g^i(\alpha) \) is monotonically increasing with \( \alpha \) for \( 0 \leq \alpha \leq 1/2 \) for all \( i \), and \( g(\alpha) \leq \alpha \) on the same range.

**Lemma 9.1** \( B(\epsilon, p(n)) = O(\log(p(n)) + \log \log(1/\epsilon)) \)

**Proof:** Note that if \( g^b(1/2 - 1/p(n)) \leq d \) and \( g^c(d) \leq \epsilon \) for some values of \( b, c, \) and \( d \), then \( B(\epsilon, p(n)) \leq a + b \).

For \( 0 \leq x \), it is clear that \( g(x) \leq 3x^2 \). From this it can be proved easily by induction on \( i \) that \( g^i(x) \leq (3x)^2 \) for all \( 0 \leq x \). Therefore \( g^i(1/4) \leq \epsilon \) if \( c \geq \log_{4/3}(1/\epsilon) \).

If \( 1/4 \leq x \leq 1/2 \), then \( 1/2 - g(x) = (1/2 - x)(1 + 2x - 2x^2) \geq (11/8)(1/2 - x) \) (Note that 11/8 is the minimum of \( 1 + 2x - 2x^2 \) on the given interval). It can be proved by induction on \( i \) that \( g^i(x) \geq (11/8)^i(1/2 - x) \), as long as \( x, g(x), \ldots, g^{i-1}(x) \) are all at least \( 1/4 \). Therefore \( g^b(1/2 - 1/p(n)) \leq 1/4 \) if \( b \geq \log_{11/8}(p(n)/4) \).

Quantities which are relevant to the time and sample complexity of \( \text{Learn} \) are \( T(\epsilon, \delta) \), the expected running time of \( \text{Learn}(\epsilon, \delta, \text{EX}) \), \( U(\epsilon, \delta) \), the time needed to evaluate a hypothesis.
returned by \text{Learn}, and \( M(\epsilon, \delta) \), the expected number of samples needed by \text{Learn}. There are corresponding quantities \( t(\delta), u(\delta), \) and \( m(\delta) \) for \text{WeakLearn}(\delta, EX). All of these functions also depend (implicitly) on \( n \).

These functions can be bounded quite effectively by the use of recurrence relations. These recurrences can be obtained by examination of \text{Learn}, with the base case being \text{WeakLearn} and its complexity. The final results will be mentioned here as lemmas, with proofs omitted. See Schapire's paper for the complete proofs.

**Lemma 9.2** The time to evaluate a hypothesis returned by \text{Learn}(\epsilon, \delta, EX) is \( U(\epsilon, \delta) = O(3^B \cdot u(\delta/5^B)) \).

**Lemma 9.3** Let \( r \) be the expected number of examples drawn from \( EX \) by any oracle \( EX \), simulated by \text{Learn} on a good run when asked to provide a single example. Then \( r \leq 4/\epsilon \).

**Lemma 9.4** On a good run, the expected number of examples \( M(\epsilon, \delta) \) needed by \text{Learn}(\epsilon, \delta, EX) is

\[
O \left( \frac{36^B}{\epsilon^2} \cdot (p^2 \log(5^B/\delta) + m(\delta/5^B)) \right)
\]

**Lemma 9.5** On a good run, the expected execution time of \text{Learn}(\epsilon, \delta, EX) is given by

\[
T(\epsilon, \delta) = O \left( 3^B \cdot (t(\delta/5^B) + \frac{108^B \cdot u(\delta/5^B)}{\epsilon^2}) \cdot (p^2 \log(5^B/\delta) + m(\delta/5^B)) \right)
\]

The fact that nearly all of the expected values above are only taken over good runs of \text{Learn} can be taken into account by "borrowing" some of the confidence \( \delta \) for that purpose.

### 9.4 Consequences of Equivalence

There are many very interesting consequences that follow from this main result (Theorem 9.1. We shall just briefly mention a few here. We refer the reader to Schapire's paper for more details on these consequences and a discussion of other interesting corollaries.

One of the more notable corollaries of Theorem 9.1 is a partial converse of Occam's Razor (see Topic 6). That is, if a concept class is learnable, then any sample can be compressed with high probability. More specifically, if \( C \) is PAC learnable then there exists a polynomial time algorithm which with confidence \( 1 - \delta \) outputs a consistent hypothesis of size polynomial in \( n \) and \( \log m \) for any sample of size \( m \) labeled according to a concept \( c \in nC \). This corollary leads to the result that any representation class that is not polynomially evaluable cannot be learned in polynomial.

Another interesting result discussed is an improved version of the learning algorithm given here, which achieves asymptotic performance which is poly-logarithmic in terms of \( 1/\epsilon \). Thus using this improved version of \text{Learn} an existing strong learning algorithm performance can be made to be poly-logarithmic in terms of \( 1/\epsilon \) by first fixing the error of the given strong learning algorithm at, say, \( \epsilon = 1/4 \) to obtain a weak learning algorithm. Then use this improved version of \text{Learn} to convert the weak learning algorithm back into a strong learning algorithm.
10.1 Introduction

In these notes we study the model of learning with queries. The material presented in this lecture comes from the paper, “Queries and Concept Learning,” by Dana Angluin [3]. In the model of learning with queries, the learner must identify an unknown hypothesis \( L_* \) from some countable concept space of subsets of a universal set of instances. Unlike in the PAC model, the algorithm is not given a stochastically generated sequence of labeled instances. It is instead allowed to actively explore its environment by asking questions of an oracle.

Throughout these notes we use the following notation.

- \( \mathcal{C} = \{L_1, L_2, \ldots \} \) is the set of concepts
- \( L_* \) is the target concept
- \( U \) is the instance space

Angluin [3] defines many different kinds of queries. We now define the types of queries we will consider here.

Membership Query The oracle will respond to input \( x \in U \) “yes” if \( x \in L_* \), otherwise it replies “no”.

Equivalence Query The oracle will respond to input hypothesis \( h \in \mathcal{C} \) “yes” if \( h = L_* \), otherwise it will return some counterexample \( x \in h \oplus L_* \). The counterexample returned by the oracle is selected by an all-powerful adversary, and no assumptions can be made about it.

Restricted Equivalence Query The oracle will respond to input hypothesis \( h \in \mathcal{C} \ “yes” \) if \( h = L_* \), otherwise it responds “no.” It does not return a counterexample.

Generalized Equivalence Query The oracle will respond as an equivalence query to any input hypothesis that is a subset of \( U \).

Unlike the PAC model in which the learner need only output a hypothesis that is a good approximation to the target concept, in the model of learning with queries, the learner must achieve exact identification. We say an algorithm exactly identifies the target concept \( L_* \) with access to certain types of queries if it always halts and outputs a concept equivalent to \( L_* \). That is, for each element of \( x \in U \) the concept output classifies \( x \) as \( L_* \) does.
10.2 General Learning Algorithms

In this section we explore some generic algorithms for learning with membership and equivalence queries. Then in the next section, we explore the relationship between this learning model and the PAC model. Finally, we will consider learning algorithms for particular concept classes.

10.2.1 Exhaustive Search

The most trivial algorithm that uses only equivalence queries is exhaustive search where each concept in $C$ is tried in turn until the correct hypothesis is found. This method clearly requires, in the worst case, $|C| - 1$ equivalence queries. It is interesting to note that for some classes this algorithm is no worse than any other. For example, consider the singletons, where for fixed instance space $U$, $C = \{ \{x\} \mid x \in U \}$. In this class, an adversary can respond to an equivalence query by returning the element in the hypothesis as the counterexample. This strategy allows the learner to eliminate at most one concept for each equivalence query. Also, $C$ even requires $|C| - 1$ membership queries because the adversary could reply "no" to each query.

This observation is generalized in the following lemma.

Lemma 10.1 Suppose $C = \{ L_1, \ldots, L_N \}$, and there exists a set $L_\top \notin C$ such that

$$L_i \cap L_j = L_\top, \quad i \neq j$$

Then any algorithm that achieves exact identification for any $L_* \in C$ must make at least $N - 1$ equivalence and membership queries in the worst case.

Proof: We must exhibit an adversary which answers each query in such a way that only a single concept can be eliminated. Keep in mind that the adversary can choose the target concept as the learning session progresses. Namely, the adversary will keep a list of target concepts that are consistent with all previous queries. The learner cannot achieve exact identification until only one concept remains in the adversary’s list.

For a membership query on instance $x$, if $x \in L_\top$ then reply “yes,” otherwise reply “no.” In the first case, no concepts from the adversary’s list are eliminated because $x$ is in every concept. In the second, at most one concept, the concept containing $x$, is eliminated—if two were eliminated, $L_i$ and $L_j$, then $L_i \cap L_j \supseteq L_\top \cup \{x\}$, a contradiction.

For an equivalence query on hypothesis $L \neq L_*$, reply “no” and return any $x \in L \cap L_\top$. Such an $x$ exists since $L_\top \notin C$, so $L \cap L_\top \neq \emptyset$. Also, if $x \in L_\top$ then no concepts are eliminated, and if $x \notin L_\top$ then at most one concept is eliminated.

There is a dual result for union.

10.2.2 The Halving Algorithm

If the learner is allowed a more powerful equivalence query, better results are obtainable. Recall that the generalized equivalence query is similar to the equivalence query but allows
as input any subset of $U$. Using such a query, any finite concept class can be learned in no more than $\lceil \lg |C| \rceil$ queries using the following algorithm.

**Halving Algorithm($C$)**

If $C$ contains one element $L$, then $L = L_*$. Halt.

Else

Let $M_C = \{ x \mid x \in C \text{ for at least } \frac{|C|}{2} \text{ concepts } \}$.

Make generalized equivalence query with hypothesis $M_C$

If “yes” then $M_C$ is equivalent to $L_*$. Halt and output $M_C$.

Else let $x$ be the counterexample.

If $x \in M_C$ then $C' = C \setminus \{ L \in C \mid x \in L \}$

Else $C' = \{ L \in C \mid x \notin L \}$

Halving Algorithm($C'$)

**Theorem 10.1** The number of queries performed by the halving algorithm is at most $\lceil \lg |C| \rceil$.

**Proof:** A query is made if $|C| > 1$ and every query eliminates at least half of the concepts.

The upper bound is often not tight. For example, for the class of singletons the halving algorithm makes only one mistake versus the upper bound of $\lg n$ given in Theorem 10.1. Littlestone [29] gives an algorithm whose performance on any class is shown to be optimal. (This algorithm is discussed in Topic 12.)

A major drawback of the halving algorithm is that it is often not computationally feasible—often exponential computation time is needed to construct $M_C$. However, it can sometimes be efficiently implemented. As an example note that Valiant’s algorithm for learning $k$-CNF [43] indirectly implements the halving algorithm. Let $h$ be defined as the conjunction of all clauses that have been true in all counterexamples. (Recall that if $h \subseteq L_*$, then $h$ is consistent with all negative examples and thus for any counterexample $x$ to $h$, $L_*(x) = +$.) Now let $C' = \{ \text{ concepts consistent with all counterexamples } \}$, so $C'$ consists of every $k$-CNF formula that is a conjunction of some subset of the clauses in $h$. The hypotheses $h$ constructed in this manner predicts according to the halving algorithm. If for a particular instance all clauses in $h$ are true, the majority is true. Similarly, if for an instance some clause $l$ in $h$ is false, for every $L \in C'$ that is true, there is some other hypothesis, $L' \in C'$ (namely $L \land l$), that is false. So, if ties are false, the majority is false.

See Topic 20 for a discussion on how approximate counting schemes can be used to efficiently implement a variant of the halving algorithm.

### 10.3 Relationship Between Exact Identification and PAC Learnability

In this section we describe how an algorithm that uses equivalence queries can be converted to a PAC-learning algorithm. We then give an example to demonstrate the converse is not true.
Theorem 10.2 Any algorithm that uses equivalence queries to achieve exact identification can be modified to achieve the PAC criterion (i.e., $Pr[error(h) \geq \epsilon] \leq \delta$) using calls to EX instead of equivalence queries. Furthermore, if the algorithm for learning with equivalence queries runs in polynomial time, then so will the PAC-learning algorithm.

Proof: We need to simulate an equivalence query with calls to EX. For the $i^{th}$ equivalence query, make $q_i = \left\lceil \frac{1}{\epsilon} (\ln \frac{1}{\delta} + i \ln 2) \right\rceil$ calls to EX. If $h_i$ (the $i^{th}$ hypothesis) is not consistent with a particular example, then that is the counterexample. Given that $h_i$ is consistent with all $q_i$ examples,

$$Pr[error(h_i) > \epsilon] \leq (1 - \epsilon)^{q_i}$$

Let $h$ be the final hypothesis. So

$$Pr[error(h) > \epsilon] \leq \sum_{i=1}^{\infty} (1 - \epsilon)^{q_i}$$

$$\leq \sum_{i=1}^{\infty} e^{-\epsilon q_i}$$

$$\leq \sum_{i=1}^{\infty} \frac{\delta}{2^i}$$

$$= \delta$$

where the second inequality holds because for all $x$, $1 - x \leq e^{-x}$. Finally, observe that the number of calls to EX made by the simulation is polynomial in the number of equivalence queries made by the original algorithm.

One can consider a variation of the PAC model in which the learner can either ask for a random example from EX or ask a membership query. We say a concept class is PAC-learnable with membership queries if the learner can meet the PAC-criterion using a polynomial number of calls to EX and polynomial number of membership queries. As an immediate corollary to the above theorem we obtain.

Corollary 10.1 A concept class that is learnable with equivalence and membership queries is PAC-learnable with membership queries. Furthermore, if the algorithm for learning with equivalence and membership queries uses polynomial time, then so will the PAC-learning algorithm.

The converse to the preceding theorem is false if efficiency must be preserved. Consider the class of singletons defined on the instance space of all binary strings of length $n$. We have already seen that the class of singletons requires $|C| - 1 = 2^n - 1$ equivalence queries, but it only requires $\left\lceil \frac{1}{\epsilon} (n \ln 2 - \ln \delta) \right\rceil$ calls to EX in the PAC model.

10.4 Examples of Exactly Identifiable Classes

In this section we will exhibit examples, some with proofs, of classes that can be exactly identified by membership and equivalence queries.

72
10.4.1 \textit{k-CNF} and \textit{k-DNF} Formulas

As we saw in the last section, Valiant’s algorithm for learning \textit{k-CNF} can be used to efficiently implement the halving algorithm. Thus it follows that this class is learnable using a polynomial number of equivalence queries. There is a logically dual method for \textit{k-DNF} formulas. Littlestone [29] describes an algorithm for learning these classes that may use significantly fewer queries. (This algorithm is described in the notes for Topic 12.) Finally, we note that by applying Lemma 10.1 to the class of 1-CNF formulas one gets that \(2^{n-1}\) restricted equivalence and membership queries are needed in the worst case. A similar hardness result can be obtained for 1-DNF.

10.4.2 Monotone DNF Formulas

In this section we describe an efficient algorithm for learning any monotone DNF formula using membership and equivalence queries.

\textbf{Theorem 10.3} The class of monotone DNF formulas over \(n\) variables is exactly identifiable by equivalence and membership queries in time polynomial in \(n\) and the number of terms in the target concept.

Before we prove this we give a definition and some observations.

\textbf{Definition 10.1} A prime implicant \(t\) of a propositional formula \(\phi\) is a satisfiable product of literals such that \(t\) implies \(\phi\) but no proper subterm of \(t\) implies \(\phi\).

\textbf{Example 10.1} \(\phi = ab \lor bc\) has prime implicants \(ac, bc,\) and \(ab\).

The number of prime implicants of a general DNF formula may be exponentially larger than the number of terms in the formula. However, for monotone DNF, the number of prime implicants is not greater than the number of terms in the formula.

\textbf{Proof:} [of Theorem 10.3] In the following let \(\{y_1, y_2, \ldots, y_n\}\) be the variables.

\textbf{Learn-Monotone-DNF}

\(\phi \leftarrow \text{FALSE}\)

\textbf{Forever}

Make equivalence query with \(\phi\)

\text{If “yes,” output } \phi; \text{ halt.}

\text{Else Let } x = b_1 b_2 \cdots b_n \text{ be the counterexample}

\text{Let } t = \Lambda_{b_i = 1} y_i

\text{For } i = 1, \ldots, n

\text{If } b_i = 1 \text{ perform membership query on } x \text{ with } i^{\text{th}} \text{ bit flipped}

\text{If “yes,” } t \leftarrow t \setminus \{y_i\} \text{ and } x = b_1 \cdots \bar{b}_i \cdots b_n

\text{Let } \phi \leftarrow \phi \lor t
We need to show that at each iteration, the term $t$ is a new prime implicant of $\phi_*$, the target concept. The proof will proceed by induction. For notation, let $\phi_i$ be the value of $\phi$ after the $i^{th}$ iteration. Firstly, $\phi_0 = \text{FALSE}$ is trivially a prime implicant of $\phi_*$. Assuming that $\phi_i$ is a prime implicant, the counterexample produced by the equivalence query is an instance $x$ for which $\phi_i(x) = 1$ and $\phi_i(x) = 0$. So $t$ is an implicant of $\phi_*$ but not of $\phi_i$. Clearly the “trimming” procedure leaves a prime implicant.

Finally, the loop iterates exactly once for each prime implicant, and as stated above, this is bounded above by the number of terms in $\phi_*$. Exactly $n$ membership queries are performed during each iteration. So, assuming queries take polynomial time, the algorithm runs in time polynomial in both $n$ and the number of terms in $\phi_*$. ■

We now show that the counterexamples returned are essential to learn monotone DNF formulas efficiently.

**Theorem 10.4** For any $n > 0$ there is a class $\mathcal{D}$ of monotone DNF formulas with $2n$ variables and $n + 1$ terms such that any algorithm that exactly identifies every formula in $\mathcal{D}$ using restricted equivalence queries and membership queries must make $2^n - 1$ queries in the worse case.

**Proof:** Given $n > 0$, let $\phi_n = x_1y_1 + x_2y_2 + \cdots + x_ny_n$ and define $\mathcal{D}$ to be the $2^n$ formulas of the form $T + \phi_n$, where $T = \ell_1 \cdot \ell_2 \cdots \ell_n$ and $\ell_i = x_i$ or $y_i$.

Now, since for any formula $\phi$ in $\mathcal{D}$ there is exactly one assignment that satisfies $\phi$ but does not satisfy $\phi_n$, any pair of distinct formulas $\phi_1, \phi_2 \in \mathcal{D}$ the assignment that satisfies both formulas are exactly those that satisfy $\phi_n$. That is, $\phi_i \cap \phi_j = \phi_n \notin \mathcal{D}$. By Lemma 10.1 and $|\mathcal{D}| = 2^n$, at least $2^n - 1$ queries are needed. ■

We now consider the question: Is the class of monotone DNF formulas learnable with a polynomial number of equivalence queries? To prove that a concept class is not efficiently learnable by equivalence queries alone, Angluin [4] developed the “Method of Approximate Fingerprints”. We now briefly describe this technique and apply it to the class of monotone DNF formulas.

The goal is to generalize the hardness result obtained for the class of singletons. Recall that for this class the adversary can reply to each equivalence query in such a way that only one concept is eliminated. This phenomenon is too much to ask, but also it is stronger than needed to prove a superpolynomial number of equivalence queries are needed.

**Definition 10.2** The instance $w_n$ is an approximate fingerprint for hypothesis $h$ in the concept class $C$ if few (a superpolynomial fraction) concepts in $C$ classify $w_h$ the same as $h$.

Approximate fingerprints can be used by the adversary to generate uninformative counterexamples to equivalence queries. Given any $h \in C$, $h \neq L_*$, the adversary would respond "no", and return $w_h$, eliminating at most a superpolynomial fraction of the concept class. Thus the learner would require a superpolynomial number of equivalence queries in order to correctly eliminate all concepts but the target.

Angluin [4] has proven that monotone DNF formulas have approximate fingerprints, thus proving that exact identification cannot be achieved with a polynomial number of equivalence queries.
queries. The key property used to prove this result is that for every monotone DNF formula \( \phi \) there is an assignment with few 1's that satisfies \( \phi \) or an assignment with few 0's that falsifies \( \phi \). Moreover, not many formulas share the value of \( \phi \) on this assignment. This assignment serves as the approximate fingerprint. See Angluin’s paper for the complete proof.

Combining these two hardness results, we get that the both equivalence and membership queries are required to learn DNF formulas. Thus the algorithm described in the beginning of the section, is the best one can expect.

10.4.3 Other Efficiently Learnable Classes

We now briefly mention a few of the many other concept classes that are learnable using membership and equivalence queries.

1. **k-Term DNF, k-Clause CNF.** Angluin [1] gives an algorithm that using equivalence and membership queries identifies any \( k \)-term DNF formula using time polynomial in \( n^k \). A dual result holds for \( k \)-clause CNF formulas. Furthermore, note that a simple modification of the proof of Pitt and Valiant [33] that \( k \)-term DNF is not PAC-learnable by \( k \)-term DNF can be used to show that for \( k > 1 \), the class of \( k \)-term DNF or \( k \)-clause CNF formulas cannot be exactly identified by any algorithm that uses just equivalence queries and runs in time polynomial in \( n^k \) unless \( P = NP \). Thus membership queries appear to be essential to Angluin’s algorithm.

2. **Regular Sets.** Angluin [2] shows that if the minimum deterministic finite automaton that generates a given regular set has \( n \) states, then the DFA can be determined in time polynomial in \( n \) and in the length of the longest counterexample. This algorithm it covered in Topic 13.

3. **Read-once Formulas.** Angluin, Hellerstein, and Karpinski [6] give an efficient algorithm to exactly identify any read-once formula using membership and equivalence queries. (A read-once formula is one in which every literal appears at most once.) Furthermore, they show that monotone read-once formulas can be learned using only membership queries.

4. **Conjunction of Horn Clauses.** A *Horn clause* is a disjunction of literals at most one of which is negated. A *Horn sentence* is a conjunction of Horn clauses. Angluin, Frazier, and Pitt [5] show that Horn sentences are efficiently learnable using membership and equivalence queries. This algorithm is described in the notes for Topic 11. Note that the class of monotone DNF formulas is properly contained in the class of Horn sentences, thus it follows that neither equivalence queries nor membership queries alone suffice.
11.1 Introduction

The material presented in this lecture comes from the paper, "Learning Conjunctions of Horn Clauses," by Dana Angluin, Michael Frazier, and Leonard Pitt [5]. We will present an algorithm the uses both membership and equivalence queries for learning the class of Boolean formulas that are expressible as conjunctions of Horn clauses. (A Horn clause is a disjunction of literals, all but at most one of which is a negated variable.) The computation time used by the algorithm is polynomial in the number of variables and the number of clauses in the target formula.

11.2 Preliminaries

We begin by defining the concept class of Horn sentences. Let $V = v_1, \ldots, v_n$ be the set of variables. A Horn sentence is defined as follows:

**Definition 11.1** A Horn clause is a disjunction of literals in which at most one literal is unnegated. A Horn sentence is a conjunction of Horn clauses.

For example suppose we had the variables, $a, b, c, d$. Then one possible Horn sentence is $(\overline{a} \lor \overline{b} \lor c) \land (\overline{d} \lor b)$. Observe that an alternate way to represent a Horn clause is as an implication in which the consequent contains at most one variable. Thus the above Horn sentence could be represented as: $(a \land b \Rightarrow c) \land (d \Rightarrow b)$. We will use this representation throughout the remainder of these notes.

What is the representational power of Horn sentences with respect to some of the other Boolean concept classes we have considered? It is easily seen that the class of Horn sentences over variable set $V$ is a proper subset of the class of Boolean formulas over $V$. Furthermore, observe that by applying DeMorgan's Law it can be shown that the class of monotone DNF formulas over variable set $V$ is a proper subset of the class of Horn sentences over $V$. Thus it follows from the work of Angluin [3, 4] that either membership or equivalence queries alone are insufficient for polynomial-time learning of Horn sentences.

Observe that the dual of the class of Horn sentences is the class of “almost monotone” DNF formulas—a disjunct of terms, where each term is a conjunct of literals, at most one of which is negated. Thus the algorithm presented in the next section can easily be modified to learn this class. Finally, note that the problem of determining whether two Horn sentences
are equivalent (and producing a counterexample if they are not) is solvable in polynomial time. Thus the equivalence query oracle could be replaced by a teacher with polynomially bounded computational resources.

11.3 The Algorithm

Before describing the algorithm, we need a few more definitions. We will use T to denote the logic constant "true" and F to denote the logic constant "false".

**Definition 11.2** Let \( x \) be an example; then \( \text{true}(x) \) is the set of variables assigned the value \( T \) by \( x \) and \( \text{false}(x) \) is the set of variables assigned the value \( F \) by \( x \).

By convention, \( T \in \text{true}(x) \) and \( F \in \text{false}(x) \).

**Definition 11.3** Let \( C \) be a Horn clause. Then \( \text{antecedent}(C) \) is the set of variables that occur negated in \( C \). If \( C \) contains an unnegated variable \( z \), then \( \text{consequent}(C) \) is just \( z \). Otherwise, \( C \) contains only negated variables and \( \text{consequent}(C) \) is \( F \).

**Definition 11.4** An example \( x \) covers a Horn clause \( C \) if \( \text{antecedent}(C) \subseteq \text{true}(x) \). Otherwise \( x \) does not cover \( C \). The example \( x \) violates the Horn clause \( C \) if \( x \) covers \( C \) and \( \text{consequent}(C) \in \text{false}(x) \).

Observe that if \( x \) violates \( C \) then \( x \) must cover \( C \), but the converse does not necessarily hold.

We are now ready to describe the algorithm. Let \( H_* \) be the target Horn sentence. The basic idea of the algorithm is that every negative example \( x \) violates some clause \( C \) of \( H_* \). As in the algorithm presented for learning monotone DNF formulas, we would like to generate the target Horn sentence clause by forming the clause \( C \) from \( H_* \) that \( x \) violates and simply add it to our current hypothesis. However, here we cannot exactly determine \( C \). We know that \( \text{antecedent}(C) \subseteq \text{true}(x) \), and \( \text{consequent}(C) \in \text{false}(x) \). Thus we will add to our current hypothesis all elements of the set

\[
\text{clauses}(x) = \left\{ \left( \bigwedge_{v \in \text{true}(x)} v \right) \Rightarrow z : z \in \text{false}(x) \right\}
\]

whenever a new negative counterexample \( x \) is obtained.

There are two problems which can occur. First of all, clauses with the "wrong" consequent can be added to the hypothesis. However, we can clean this up using the positive counterexamples. The second problem that occurs is the antecedent we select possible contains more variables than it should and thus it may not be false when the "real" antecedent is false. In order for the algorithm to work, it is important to ensure that we only have one set of clauses (as defined by Equation (11.1)) in the hypothesis corresponding to each clause in the target. Thus on a negative counterexample, we must first try to reduce the number
\begin{algorithm}
\textbf{Learn-Horn-Sentence()}
1 $S \leftarrow \emptyset$ \textit{($s_i$ denotes $i$th element of $S$)}
2 $H \leftarrow \emptyset$ \textit{(always true hypothesis)}
3 \textbf{UNTIL} equivalent($H$) returns “yes” \textbf{DO}
4 \hspace{1em} \textbf{BEGIN}
5 \hspace{2em} Let $x$ be the counterexample returned
6 \hspace{2em} IF $x$ violates at least one clause of $H$
7 \hspace{3em} THEN ($x$ is a positive example)
8 \hspace{3em} remove from $H$ every clause that $x$ violates
9 \hspace{2em} ELSE ($x$ is a negative example)
10 \hspace{3em} \textbf{BEGIN}
11 \hspace{4em} FOR each $s_i \in S$ such that true($s_i \cap x$)
12 \hspace{4em} \hspace{1em} is properly contained in true($s_i$)
13 \hspace{5em} \textbf{BEGIN}
14 \hspace{6em} query member($s_i \cap x$)
15 \hspace{6em} \textbf{END}
16 \hspace{4em} IF any of these queries is answered “no”
17 \hspace{5em} THEN let $i$ be the least number such that
18 \hspace{5em} \hspace{1em} member($s_i \cap x$) was answered “no”
19 \hspace{5em} \hspace{1em} refine $s_i \leftarrow s_i \cap x$
20 \hspace{4em} ELSE add $x$ to end of $S$
21 \hspace{4em} \textbf{ENDIF}
22 \hspace{3em} \textbf{END}
23 \hspace{2em} \textbf{END}
24 \hspace{1em} \textbf{RETURN} $H$
\end{algorithm}

Figure 11.1: Algorithm for learning Horn sentences.

of variables in the antecedent of a set of clauses currently in the hypothesis before using the formula in Equation (11.1) to create new clauses.

The algorithm maintains a sequence $S$ of negative examples. Each new negative counterexample is used to either refine one element of $S$, or is added to the end of $S$. In order to learn all of the clauses of $H_e$, we ensure that the clauses induced by the examples in $S$ approximate distinct clauses of $H_e$. This will occur if the examples in $S$ violate distinct clauses of $H_e$. To maintain this invariant, whenever a new negative counterexample could be used to refine several examples in the sequence $S$, only the earliest one in the set is refined. The algorithm is shown in Figure 11.1.
11.4 Example Run of Algorithm

Before analyzing this algorithm, we will examine an example run of it. Let $V = \{a, b, c, d\}$, and let $H = (ab \Rightarrow c) \land (d \Rightarrow b)$.

Initially $S = \emptyset$, $H = \emptyset$ (always true)

**EQUIV($H$)? 1101 (negative example)**
$S = \{1101\}$
$H = (abd \Rightarrow c) \land (abd \Rightarrow F)$
Observe that the clause $(abd \Rightarrow c)$ is an approximation to $(ab \Rightarrow c)$. But note that for the example 1100 the clause $(ab \Rightarrow c)$ is not satisfied, but $(abd \Rightarrow c)$ is satisfied.

**EQUIV($H$)? 1100 (negative example)**
**MEMB(1100)? “no”**
$S = \{1100\}$
$H = (ab \Rightarrow c) \land (ab \Rightarrow d) \land (ab \Rightarrow F)$

**EQUIV($H$)? 1001 (negative example)**
**MEMB(1000)? “yes”**
$S = \{1100, 1001\}$
$H = (ab \Rightarrow c) \land (ab \Rightarrow d) \land (ab \Rightarrow F) \land (ad \Rightarrow b) \land (ad \Rightarrow c) \land (ad \Rightarrow F)$

**EQUIV($H$)? 1110 (positive example)**
$S = \{1100, 1001\}$
$H = (ab \Rightarrow c) \land (ad \Rightarrow b) \land (ad \Rightarrow c) \land (ad \Rightarrow F)$
Observe that the positive example only modify $H$ and thus if there is every another negative example, it’s effect is removed.

**EQUIV($H$)? 0011 (negative example)**
**MEMB(0000)? “yes”**
**MEMB(0001)? “no”**
$S = \{1100, 0001\}$
$H = (ab \Rightarrow c) \land (ab \Rightarrow d) \land (ab \Rightarrow F) \land (d \Rightarrow a) \land (d \Rightarrow b) \land (d \Rightarrow c) \land (d \Rightarrow F)$

**EQUIV($H$)? 0101 (positive example)**
$S = \{1100, 1001\}$
$H = (ab \Rightarrow c) \land (d \Rightarrow b)$

**EQUIV($H$)? “yes”**
11.5 Analysis of Algorithm

Clearly since the algorithm only halts when it receives “yes” from an equivalence query, the hypothesis output is correct. Thus what must be shown is that the number of membership and equivalence queries is polynomial in $n$ and $m$ (the number of clauses in $H_*$), and the computation time is polynomial in $n$ and $m$. Here we shall just sketch the main ideas of the analysis, see the Angluin, Frazier, Pitt paper [5] for the details.

The key property which must be shown is that throughout the execution of the algorithm, at no time do two distinct elements of $S$ violate the same clause of $H_*$. Once this has been shown, the remainder of the analysis is fairly straightforward. This key property is proven using the following two lemmas.

**Lemma 11.1** For each execution of the main loop of line 3, the following invariant holds. Suppose that in step 5 of the algorithm a negative example $x$ is obtained such that for some clause $C$ of $H_*$ and for some $s_i \in S$, $x$ violates $C$ and $s_i$ covers $C$. Then there is some $j \leq i$ such that in step 17 the algorithm will refine $s_j$ by replacing it with $s_j \cap x$.

This lemma is proven by induction on the number of iterations $k$ of the main loop of line 3.

**Lemma 11.2** Let $S$ be a sequence of elements constructed for target $H_*$ by the algorithm. Then

1. $\forall k \forall (i < k) \forall (C \in H_*)$ if $s_k$ violates $C$ then $s_i$ does not cover $C$

2. $\forall k \forall (i \neq k) \forall (C \in H_*)$ if $s_k$ violates $C$, then $s_i$ does not violate $C$.

Here too, these two invariants are shown to hold by using an inductive proof.

An immediate corollary of the second property of the second lemma is that at no time do two distinct elements in $S$ violate the same clause of $H_*$. And since each of the elements in $S$ is a negative example if follows that every element of $S$ violates at least one clause of $H_*$. Thus at no time doing the execution of the algorithm does $S$ contain more than $m$ elements.

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

**Theorem 11.1** A Horn sentence consisting of $m$ clauses over $n$ variables can be learned exactly in time $\tilde{O}(m^3n^4)$ using $O(m^2n^2)$ equivalence queries and $O(m^2n)$ membership queries\(^5\).

**Proof Sketch:** The sequence $S$ is only changed by appending a new element to it, or refining an existing element. Thus $|S|$ never decreases. Since $|S| \leq m$ it follows that line 18 is executed at most $m$ times. Observe that when an element of $S$ is refined in line 17, it now contains strictly fewer variables assigned the value “true”. Thus a given element can be refined at most $n$ times, and so line 17 is executed at most $mn$ times. Whenever the ELSE clause at line 9 is executed, either line 17 or 18 is executed. Thus lines 9–21 are executed at

\(^5\)The “soft-oh” notation $\tilde{O}$ is like the standard “big-oh” notation except that logarithmic terms are ignored.
most \(nm + m = (n + 1)m\) times. So the total number of membership queries made are at most \((n + 1)m^2\).

Observe that the cardinality of clauses(s) in any hypothesis \(H\) constructed in line 15 is at most \((n + 1)m\). Each positive counterexample obtained in line 5 causes at least one clause to be removed from \(H\), thus the equivalence queries can produce at most \((n + 1)m\) positive counterexamples between modifications of \(S\). Therefore, line 8 is executed at most \((n + 1)^2m^2\) times. Thus the total number of equivalence queries is at most \((n + 1)^2m^2 + (n + 1)m + 1\). Finally, it can be shown that the time needed for each execution of the main loop is \(\mathcal{O}(n^2m)\).

\[\]
12.1 Introduction

In these notes we first introduce the on-line (or mistake-bound) learning model. Then we consider when this model is applied to concept classes that contain a large number of irrelevant attributes. Most of this lecture comes from the paper, “Learning when Irrelevant Attributes Abound: A New Linear-threshold Algorithm,” by Nick Littlestone [29].

12.2 On-line Learning Model

Observe that one property of the PAC-learning model is that it is a batch model—there is a separation between the training phase and the performance phase. Thus in the PAC model a learning session consists of two phases: the training phase and the performance phase. In the training phase the learner is presented with a set of instances labeled according to the target concept \( c \in C_n \). At the end of this phase the learner must output a hypothesis \( h \) that classifies each \( x \in X_n \) as either a positive or negative instance. Then in the performance phase, the learner uses \( h \) to predict the classification of new unlabeled instances. Since the learner never finds out the true classification for the unlabeled instances, all learning occurs in the training phase.

We now give an example from Goldman’s thesis [18] to motivate the on-line learning model. This model is also known as the mistake-bound learning model. Suppose that when arriving at work (in Boston) you may either park in the street or park in a garage. In fact, between your office building and the garage there is a street on which you can always find a spot. On most days, street parking is preferable since you avoid paying the $10 garage fee. Unfortunately, when parking on the street you risk being towed ($50) due to street cleaning, snow emergency, special events, etc. When calling the city to find out when they tow, you are unable to get any reasonable guidance and decide the best thing to do is just learn from experience. There are many pieces of information that you might consider in making your prediction; e.g. the date, the day of the week, the weather. We make the following two assumptions: enough information is available to make good predictions if you know how to use it, and after you commit yourself to one choice or the other you learn of the right decision. In this example, the city has rules dictating when they tow; you just don’t know them. If you park on the street at the end of the day you know if your car was towed; otherwise when walking to the garage you see if the street is clear (i.e. you learn if you would have been towed).
The on-line model is designed to study algorithms for learning to make accurate predictions in circumstances such as these. Formally, an on-line learning algorithm for \( C \) is an algorithm that runs under the following scenario. A learning session consists of a set of trials. In each trial, the learner is given an unlabeled instance \( x \in X \). The learner uses its current hypothesis to predict if \( x \) is a positive or negative instance of the target concept \( c \in C \) and then the learner is told the correct classification of \( x \). If the prediction was incorrect, the learner has made a mistake. Note that in this model there is no training phase. Instead, the learner receives unlabeled instances throughout the entire learning session. However, after each prediction the learner "discovers" the correct classification. This feedback can then be used by the learner to improve its hypothesis. Observe that in this model it is beneficial for the learning algorithm to calculate hypothesis incrementally rather than starting from scratch each time.

In this learning model we shall evaluate the performance of a learning algorithm by the number of prediction mistakes made by the learning algorithm. We now describe the two most common ways in which this notion has been formalized.

**Probabilistic Mistake Bound:** Let \( D \) be some arbitrary and unknown probability distribution on the instance space. The probabilistic mistake bound is the probability that the learner's hypothesis disagrees with \( c \) on the \( t^{th} \) randomly drawn instance from \( D \). Formally, given any \( n \geq 1 \) and any \( c \in C_n \), the learner's goal is to output a hypothesis \( h \) such that the probability that \( h \) makes a mistake on the \( t + 1^{st} \) trial is at most \( p(n) t^{-\beta} \) for some polynomial \( p(n) \) and \( 0 < \beta \).

**Absolute Mistake Bound:** The absolute mistake bound is the worst-case total number of mistakes made when the learner must make predictions for any, possibly infinite, sequence of instances. (Even if the instance space is finite, repetitions may occur.) Formally, given any \( n \geq 1 \) and any \( c \in C_n \), the learner's goal is to make at most \( p(n) \) mistakes for some polynomial \( p(n) \).

Throughout the remainder of these notes we shall consider an on-line learning model where the absolute mistake bound criterion is used. In other words, we assume that an adversary selects the order in which the instances are presented to the learner and we evaluate the learner by the maximum number of mistakes made during the learning session. Our goal is to minimize the worst-case number of mistakes using an efficient learning algorithm (i.e. each prediction is made in polynomial time). Observe that such mistake bound are quite strong in that the order in which examples are presented does not matter; however, it is impossible to tell how early the mistakes will occur. See Haussler et al. [21] for a discussion of the relationship between the PAC model and on-line learning with the probabilistic mistake bound criterion.

After studying some general results about this on-line learning model we will focus on the situation in which the instances are drawn from the Boolean domain and proper classification for each instance can be determined by only a small fraction of the attribute space. Pattern recognition falls under this situation since a feature detector might extract a large number
of features for the learner's consideration not knowing which few will prove useful. Another example is building new concepts as Boolean functions of previously learned concepts that are stored in a library. For this problem, the learner may need to sift through a large library of available concepts to find the suitable ones to use in expressing each new concept. (The concept class of $k$-DNF fits into this model where the terms come from the library.)

### 12.3 Definitions and Notation

In this section we describe the notation used by Littlestone [29]. A concept class $C$ consists of concepts $c$ which are Boolean functions $c : \{0, 1\}^n \rightarrow \{0, 1\}$.

- For any algorithm $A$ and target concept $c \in C$, let $M_A(c)$ be the maximum over all possible sequences of examples of the number of mistakes that algorithm $A$ makes while learning the concept $c$.
- Define $M_A(C) = \max_{c \in C} M_A(c)$. If $C$ is the empty set then by definition $M_A(C) = -1$.
- Define $opt(C) = \min_A M_A(C)$, the minimum over all possible algorithms of the worst-case number of mistakes.
- An algorithm $A$ is optimal for $C$ if and only if $M_A(C) = opt(C)$.

### 12.4 Halving Algorithm

One algorithm which often yields a good mistake bound is the halving algorithm (as described in Section 10.2.2). We now review the halving algorithm using the notation of Littlestone. We shall then look at a variant of it that can be shown to perform optimally.

Let $\text{CONSIST}$ be a subset of the concepts in $C$ that are consistent with all previous examples. So initially, $\text{CONSIST} = C$. Given a target concept class $C$ and an instance $x$, we define $\xi_i(C, x) = \{c \in C | c(x) = i\}$ for $i = 0$ or $i = 1$. Thus the sets $\xi_0(C, x)$ and $\xi_1(C, x)$ are the set of concepts that are 0 at $x$, and the set of concepts that are 1 at $x$, respectively.

Upon receiving an instance $x$, the halving algorithm computes the sets $\xi_0(\text{CONSIST}, x)$ and $\xi_1(\text{CONSIST}, x)$. If $|\xi_1(\text{CONSIST}, x)| > |\xi_0(\text{CONSIST}, x)|$ then the algorithm predicts 1, otherwise it predicts 0. After receiving feedback the learner updates $\text{CONSIST}$: if $c(x) = 0$ then set $\text{CONSIST} = \xi_0(\text{CONSIST}, x)$, and if $c(x) = 1$ then set $\text{CONSIST} = \xi_1(\text{CONSIST}, x)$.

**Theorem 12.1** For any nonempty target class $C$, $M_{\text{HALVING}}(C) \leq \log_2 |C|$.

**Proof:** Since the halving algorithm predicts according to the majority, its response is consistent with at least half of $\text{CONSIST}$. Therefore the size of $\text{CONSIST}$ drops by a factor of at least two at each mistake. And since there is a consistent function, $|\text{CONSIST}| \geq 1$ always, so the algorithm can make no more than $\log_2 |C|$ mistakes. 

The halving algorithm tells us that we can always choose a concept class such that $\log_2 |C|$ mistakes are actually made. Hence we have
Theorem 12.2 For $C$ such that $|C|$ is finite, $\text{opt}(C) \leq \log_2 |C|$.

Before considering an algorithm that often makes fewer mistakes than the halving algorithm, we briefly consider the relation between this on-line learning model and the model of learning with equivalence queries. If both an equivalence query algorithm and a mistake-bound algorithm are applied to learning the same concept (over the same representation class), we have:

$$\frac{\# \text{ equiv. queries for exact id.}}{\# \text{ mistakes}} \leq \frac{\# \text{ mistakes} + 1}{\# \text{ equiv. queries for exact id.}}$$

These inequalities follow from the simple observation that each mistake made in the on-line model serves as a counterexample to an equivalence query. Furthermore, an equivalence query with the correct hypothesis corresponds to a hypothesis for which no additional mistakes will occur.

## 12.5 Standard Optimal Algorithm

Along with the problem that the halving algorithm often requires an exponentially amount of time and space, it is not always an optimal algorithm. After giving some more definitions, we shall describe a modification of the halving algorithm that always performs optimally (although is still not computationally feasible in most cases).

A mistake tree for $C$ over $X$ is defined to be a binary tree each of whose nodes is a non-empty subset of $C$ and each of whose internal nodes is labeled with an $x \in X$ which satisfies:

1. The root node is $C$ (along with a label).

2. Given any internal node $C'$ labeled with $x$, the left child (if present) is $\xi_0(C', x)$ and the right child (if present) is $\xi_1(C', x)$.

A complete $k$-mistake tree is a mistake tree that is a complete binary tree of height $k$. (The height of a binary tree is the number of edges in the longest path from the root to a leaf.) Finally, we define $K(C)$ as the largest integer $k$ such that there exists a complete $k$-mistake tree for the concept class $C$. We shall use the convention that $K(\emptyset) = -1$.

In Figure 12.1 is an example of a complete 2-mistake tree. The concept class used is a simple one; $X = \{0, 1\}^5$ and $C$ consists of the five concepts, $f_i(x_1, \ldots, x_5) = x_i$ for $i = 1, \ldots, 5$.

As we shall show, these trees characterize the number of mistakes made by the optimal learning algorithm.

We now define the standard optimal algorithm (SOA):

Let CONSIST contain all $c \in C$ consistent with all past instances.
Predict 1 if $K(\xi_1(\text{CONSIST}, x)) > K(\xi_0(\text{CONSIST}, x))$.
Predict 0 otherwise.
Figure 12.1: An example of a complete 2-mistake tree for a concept class \( C \) over instance space \( X = \{0,1\}^5 \) where \( C \) consists of the five concepts \( f_i(x_1, \ldots, x_5) = x_i \) for \( i = 1, \ldots, 5 \).

So if a mistake is made the remaining consistent functions have the smaller maximal complete mistake tree. As we shall show this yields an optimal algorithm. However, we first prove the following two lemmas.

**Lemma 12.1** \( \text{opt}(C) \geq K(C) \).

**Proof:** If \( C = \emptyset \) then by definition \( K(C) = -1 \) and the lemma trivially follows. So assume that \( C \neq \emptyset \) and \( k = K(C) \). Given any algorithm \( A \) we show how the adversary can choose a target concept and a sequence of instances such that \( A \) makes at least \( k \) mistakes. If \( k = 0 \) the lemma is trivially satisfied. Otherwise, the adversary chooses the instance that is the root of a complete \( k \)-mistake tree for \( C \). Regardless of \( A \)'s prediction the adversary replies that the prediction is incorrect. The remaining consistent concepts form a complete \((k-1)\)-mistake tree, so by induction we are done.

**Lemma 12.2** Suppose that we run SOA in order to learn a concept in \( C \) where \( x_1, \ldots, x_t \) is the sequences of instances given to SOA. Let \( \text{CONSIST}_i \) be the value of the variable \( \text{CONSIST} \) at the beginning of the \( i \)th trial. Then for any \( k \geq 0 \) and \( i \in \{1, \ldots, t\} \), if \( K(\text{CONSIST}_i) = k \), then SOA will make at most \( k \) mistakes during the trials \( i, \ldots, t \).

**Proof:** This will be a proof by induction on \( k \). For the base case observe that by construction, the target function is always in \( \text{CONSIST}_i \). If \( \text{CONSIST}_i \) has two elements, we can always use an instance on which they differ as the root node of a 1-mistake tree, so \( K(\text{CONSIST}_i) = 0 \) implies that \( \text{CONSIST}_i \) has only the target function. Since \( K(0) = -1 \) (by definition), SOA
cannot make any mistakes when only the target function is left. Hence the base case \( k = 0 \) is proven.

We now prove the lemma for arbitrary \( k > 0 \), assuming it holds for \( k - 1 \). If SOA makes no mistakes during trials \( i, \ldots, t - 1 \) then the lemma is trivially true. So let \( j \) be the first trial among \( i, \ldots, t - 1 \) in which a mistake is made. We now prove by contradiction that \( \xi_0(\text{CONSIST}_j, x_j) \) and \( \xi_1(\text{CONSIST}_j, x_j) \) cannot both be complete \( k \)-mistake trees. Suppose there are \( k \)-mistake trees for both \( \xi_0(\text{CONSIST}_j, x_j) \) and \( \xi_1(\text{CONSIST}_j, x_j) \). Then we can combine these into \((k+1)\)-mistake tree by using \( x_j \) as a root node. But by hypothesis, such a tree cannot exist (\( k \) is the largest size of a mistake tree). Therefore one of \( K(\xi_0(\text{CONSIST}_j, x_j)) \) or \( K(\xi_1(\text{CONSIST}_j, x_j)) \) is less than \( k \). Since SOA always picks the larger of the two, and it made a mistake, then \( K(\text{CONSIST}_{j+1}) \) will be less than \( k \). Using induction only \( k - 1 \) mistakes will be made from this point on. So only \( k \) mistakes can be made completing the inductive step.

Now we are ready to prove the main result of this section.

**Theorem 12.3** \( \text{opt}(C) = M_{\text{SOA}}(C) = K(C) \).

**Proof:** By setting \( i = 1 \) and \( k = K(C) \) in the Lemma 12.2, we get \( M_{\text{SOA}}(C) \leq K(C) \). Furthermore, by Lemma 12.1, \( K(C) \leq \text{opt}(C) \). Combining these two inequalities it follows that \( M_{\text{SOA}}(C) \leq \text{opt}(C) \). Finally, by definition, \( \text{opt}(C) \leq M_{\text{SOA}}(C) \).

Let us now consider some lower bounds on \( \text{opt}(C) \). The Vapnik-Chervonenkis dimension is useful in this respect.

**Theorem 12.4** \( \text{opt}(C) \geq VCD(C) \).

**Proof:** Let \( \{v_1, \ldots, v_k\} \subseteq X \) be any set shattered by \( C \). Then clearly a complete \( k \)-mistake tree can be constructed for \( C \), where all internal nodes at a depth \( i \) are labeled with \( v_{i+1} \). Apply this procedure to \( k = VCD(C) \).

We note however that this is not a tight lower bound. Let \( C \) be the concept class of the instance space \( X = \{1, \ldots, 2^n - 1\} \) where \( C = \{c_i : X \rightarrow \{0, 1\} | c_i(x_j) = 1 \text{ if and only if } j < i\} \). Thus, these are concepts of "all things less than \( i \". Clearly \( VCD(C) = 1 \) since for two concepts \( c_i \) and \( c_j \), the greater of \( \{i, j\} \) cannot be covered by one concept without covering the other. Yet, we can show that \( \text{opt}(C) \geq n \) by constructing a complete \( n \)-mistake tree as follows: label the root node \( 2^n \). Each branch of the tree is the same type but half as big, so by inductively constructing the binary tree, we have a complete \( n \)-mistake tree.

### 12.6 The Linear Threshold Algorithm: WINNOW1

In this section we describe an algorithm that efficiently deals with a large number of irrelevant attributes when learning in a Boolean domain. If desired this algorithm can be implemented within a neural network framework as a simple linear threshold function. This algorithm is similar to the classical perceptron algorithm, except that it uses a multiplicative weight-update scheme that permits it to perform much better than classical perceptron training.
<table>
<thead>
<tr>
<th>prediction</th>
<th>correct</th>
<th>name</th>
<th>update scheme</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>elimination</td>
<td>if ( x_i = 1 ) then set ( w_i = 0 ).</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>promotion</td>
<td>if ( x_i = 1 ) then set ( w_i = \alpha \cdot w_i ).</td>
</tr>
</tbody>
</table>

Figure 12.2: The update scheme used by WINNOW1.

algorithms when many attributes are irrelevant. From empirical evidence it appears that for the perceptron algorithm the number of mistakes grows linearly with the number of irrelevant attributes. Here we shall describe an algorithm for which the number of mistakes only grows logarithmically with the number of irrelevant attributes.

A \textit{linearly separable Boolean function} is a map \( f : \{0,1\}^n \rightarrow \{0,1\} \) such that there exists a hyperplane in \( \mathbb{R}^n \) that separates the inverse images \( f^{-1}(0) \) and \( f^{-1}(1) \) (i.e. the hyperplane separates the point on which the function is 1 from those on which it is 0). An example of a linearly separable function is any monotone disjunction: if \( f(x_1, \ldots, x_n) = x_{i_1} \lor \ldots \lor x_{i_k}, \) then the hyperplane \( x_{i_1} + \ldots + x_{i_k} = 1/2 \) is a separating hyperplane.

We present a limited form, WINNOW1, and will later generalize it. WINNOW1 is a linear threshold algorithm over the Boolean space \( X = \{0,1\}^n \) that is designed for learning monotone disjunctions. There are \( n \) real valued weights \( w_1, \ldots, w_n \) maintained by the algorithm. Initially each weight is set to 1. Also, a real number \( \theta \), called the \textit{threshold}, is utilized. When WINNOW1 receives an instance \( x = (x_1, \ldots, x_n) \), it predicts as follows:

- if \( \sum_{i=1}^{n} w_i x_i > \theta \) then it predicts 1.
- if \( \sum_{i=1}^{n} w_i x_i \leq \theta \) then it predicts 0.

When a mistake is made, the weights with non-zero \( x_i \) are updated as shown in Figure 12.2. Note that the threshold \( \theta \) is never altered. Good values for the parameters \( \theta \) and \( \alpha \) are \( \theta = n/2 \) and \( \alpha = 2 \).

We now present three lemmas which will be used to prove a later theorem. All three have as preconditions that WINNOW1 is run with \( \alpha > 1 \) and \( \theta \geq 1/\alpha \) for the learning of \( k \)-literal monotone disjunctions.

\textbf{Lemma 12.3} Let \( u \) be the number of promotion steps that have occurred in some sequence of trials, and \( v \) the number of elimination steps in the same trials. Then \( v \leq (n/\theta) + (\alpha - 1)u \).

\textbf{Proof:} Consider the sum \( \sum_{i=1}^{n} w_i \). Initially this sum is \( n \) since all of the weights are initially 1. At each promotion, the sum can increase by no more than \( (\alpha - 1)\theta \), since the sum (over all \( x_i \) that are on) must be less than \( \theta \) for a promotion to occur. Similarly, at each elimination step, the sum must be decreased by at least \( \theta \). Hence

\[ 0 \leq \sum_{i=1}^{n} w_i \leq n + \theta(\alpha - 1)u - \theta v. \]
Thus

$$\theta v \leq n + \theta(\alpha - 1)u$$

giving the desired result.

Lemma 12.4 For all $i$, $w_i \leq \alpha \theta$ (after any number of trials).

Proof: Since $\theta \geq 1/\alpha$, for all $i$, initially $w_i \leq \alpha \theta$. We now proceed by induction on the number of steps. Note that $w_i$ is only increased by a promotion step when $x_i = 1$ and $\sum_{i=1}^{n} w_ix_i \leq \theta$. These conditions can only occur together if $w_i \leq \theta$ prior to promotion. Thus $w_i \leq \alpha \theta$ after the promotion step. So after any step the claim is true, hence it is always true.

Lemma 12.5 After $u$ promotion steps and any number of eliminations, there exists an $i$ such that $\log_\alpha w_i \geq u/k$.

Proof: Let the $k$-literal disjunction we are learning be of the form

$$f(x_1, \ldots, x_n) = x_{i_1} \lor \ldots \lor x_{i_k}$$

Let the set $R = \{i_1, \ldots, i_k\}$, and consider the product $P = \prod_{j \in R} w_j$. Since $f = 0$ if and only if $x_j = 0$ for all $j \in R$, and elimination occurs when $f = 0$, elimination cannot affect the product $P$ at all. Also, at each promotion step, $P$ is increased by at least a factor of $\alpha$, since at least one of the $x_i$s was on for $i \in R$. At first we have $P = 1$. After $u$ promotions (and any number of eliminations), $P \geq \alpha^u$. Taking logs of both sides, we have $\sum_{i \in R} \log_\alpha w_i \geq u$. Since $|R| = k$, for some $i \in R$ we must have that $\log_\alpha w_i \geq u/k$.

We are finally ready to prove an upperbound on the number of mistakes made by WINNOW1 when learning a $k$-literal monotone disjunction.

Theorem 12.5 For the learning of $k$-literal monotone disjunctions, if WINNOW1 is run with $\alpha > 1$ and $\theta \geq 1/\alpha$, then the total number of mistakes is at most $\alpha k(\log_\alpha \theta + 1) + n/\theta$.

Proof: The total number of mistakes is clearly the number of promotion steps plus the number of elimination steps $(u + v)$. Lemmas 12.4 and 12.4 yield the following bound on $v$:

$$u/k \leq \log_\alpha w_i \leq \log_\alpha \alpha \theta = \log_\alpha \theta + 1$$

So

$$u \leq k(\log_\alpha \theta + 1)$$

Plugging in this value of $u$ into the inequality of Lemma 12.3, we obtain the following bound on $v$:

$$v \leq (n/\theta) + (\alpha - 1)k(\log_\alpha \theta + 1)$$

Adding the two bounds together gives the desired result.

Observe that WINNOW1 need not have prior knowledge of $k$ although the number of mistakes will depend on $k$. Littlestone's paper also discusses how to optimally choose $\theta$ and $\alpha$ if an upperbound on $k$ is known a priori.

Noticing that $k$-literal monotone disjunctions are 1-term monotone $k$-DNF formulas, we state (without proof) the following lower bound on the VC dimension and thus on the number of mistakes made in the on-line model. See Littlestone's paper for the proof.

90
Lemma 12.6 For $1 \leq k \leq n$ and $1 \leq l \leq \binom{n}{k}$, let $C$ be the class of $l$-term monotone $k$-DNF formulas. Let $m$ be any integer with $k \leq m \leq n$ with $\binom{m}{k} \geq l$. Then $VCD(C) \geq kl[\log_2(n/m)]$.

We now apply this theorem to two special cases. If $l = 1$ and $m = k$ then we get that $VCD(C) \geq k[\log_2 n/k]$ where $C$ contains all conjunctions of at most $k$ variables chosen from $n$ variables. Likewise, if $k = 1$ and $m = l$ then $VCD(C) \geq l[\log_2 n/l]$ where $C$ contains all disjunctions of at most $l$ variables chosen from $n$ variables.

12.7 Extensions: WINNOW2

Let $X = \{0, 1\}^n$ be the instance space, and $0 < \delta \leq 1$. We define $F(X, \delta)$ to be the set of all functions $X \rightarrow \{0, 1\}$ with the following property: for each $f \in F(X, \delta)$ there exist $\mu_1, \ldots, \mu_n \geq 0$ such that for all instances $x = (x_1, \ldots, x_n) \in X$,

$$\sum_{i=1}^{n} \mu_i x_i \geq 1 \iff x(1) = 1$$

and

$$\sum_{i=1}^{n} \mu_i x_i < 1 - \delta \iff x(1) = 0$$

That is, the inverse images of 0 and 1 are separated by at least $\delta$.

An example of such a class of functions would be the $r$-of-$k$ threshold functions. These functions are defined by selecting $k$ variables which are “important”. Then, $f(x) = 1$ whenever $r$ or more of these $k$ significant variables are on. Let the selected variables be $x_{i_1}, \ldots, x_{i_k}$.

$$f(x) = 1 \iff x_{i_1} + \ldots + x_{i_k} \geq r$$

Thus,

$$f(x) = 1 \iff (1/r)x_{i_1} + \ldots + (1/r)x_{i_k} \geq 1$$

Also,

$$f(x) = 0 \iff x_{i_1} + \ldots + x_{i_k} \leq r - 1$$

Thus,

$$f(x) = 0 \iff (1/r)x_{i_1} + \ldots + (1/r)x_{i_k} \leq 1 - (1/r)$$

Hence $r$-of-$k$ threshold functions are contained in $F(\{0, 1\}^n, 1/r)$.

We describe the algorithm WINNOW2 here. It is basically the same as WINNOW1, but updates made in response to mistakes is slightly different: instead of eliminating weights entirely, they are divided by $\alpha$. One could call this altered step a demotion. See Figure 12.3 for the update scheme used by WINNOW2.

A theorem similar to the one for WINNOW1 exists to give an upper bound on the number of mistakes that WINNOW2 will make. We state without proof the theorem here. For a proof see Littlestone’s paper.
<table>
<thead>
<tr>
<th>prediction</th>
<th>correct</th>
<th>name</th>
<th>update scheme</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>demotion</td>
<td>if $x_i = 1$ then set $w_i = w_i/\alpha$.</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>promotion</td>
<td>if $x_i = 1$ then set $w_i = w_i \cdot \alpha$.</td>
</tr>
</tbody>
</table>

Figure 12.3: The update scheme used by WINNOW2.

**Theorem 12.6** Let $0 < \delta \leq 1$, and the target function be in $F(X, \delta)$ for $X = \{0, 1\}^n$. If the appropriate $\mu_1, \ldots, \mu_n$ are chosen so that the target function satisfies the definition for $F(X, \delta)$, and WINNOW2 is run with $\alpha = 1 + \delta/2$ and $\theta \geq 1$ and the algorithm gets its instances from $X$, then the number of mistakes is bounded above by

$$\frac{8}{\delta^2} \frac{n}{\theta} + \left( \frac{5}{\delta} + \frac{14 \ln \theta}{\delta^2} \right) \sum_{i=1}^{n} \mu_i$$

For $r$-of-$k$ threshold functions we have $\delta = 1/r$ and $\sum \mu_i = k/r$. So setting $\theta = n$ yields a mistake bound of $8r^2 + 5k + 14kr \ln n$.

### 12.8 Transformations to Other Concept Classes

Suppose we had a learnable concept class, and wanted to learn a second class. If morphisms existed which map instances between the two concept spaces, and mapped from the answers of one to the answers of the second, then it would not seem to be difficult to learn the second class by simply pretending to be learning the first. This is the basic idea behind these transformations.

Let the instance space of the derived algorithm be $X_1$, and that of the original algorithm be $X_2$. The transformations will take the form of $T_1 : X_1 \rightarrow X_2$ and $T_p : \{0, 1\} \rightarrow \{0, 1\}$. The mapping $T_1$ maps between the instance spaces, and $T_p$, the map between predictions, is either the identity function or the function which interchanges 0 and 1. Let $A_1$ be the algorithm desired to learn concepts over $X_1$, and let $A_2$ be the algorithm provided that learns concepts over $X_2$. When $A_1$ gets an instance $x \in X_1$, it sends the instance $T_i(x)$ to $A_2$. Then $A_2$ generates a prediction $y$. Next, $A_2$ sends this prediction to $A_1$ which outputs $T_p(y)$. Finally, any reinforcements are passed directly to $A_2$.

So suppose we have an algorithm $A_2$, and we want to derive an algorithm to learn $C_1$. We need not only a $C_2$ that $A_2$ can learn, but also two mappings $T_i$, $T_p$ such that for all $g \in C_1$, there exists $f \in C_2$ such that $T_p \circ f \circ T_i = g$.

**Theorem 12.7** Suppose we are given maps $T_i : X_1 \rightarrow X_2$ and $T_p : \{0, 1\} \rightarrow \{0, 1\}$, an original algorithm $A$ that takes instances from $X_2$, and a derived algorithm $B$ (defined as above). Suppose also that we have a function $g : X_1 \rightarrow \{0, 1\}$ that we want $B$ to learn. If $f : X_2 \rightarrow \{0, 1\}$ is a function that $A$ can learn with some bounded number of mistakes such that $T_p \circ f \circ T_i = g$, then $B$ will learn $g$ with at most $MA(f)$ mistakes.
Proof: This proof is trivial. Basically, \( B \) can only make a mistake when \( A \) has. □

We now go through several example transformations.

Example 12.1 Arbitrary Disjunctions

We first consider learning arbitrary disjunctions, using WINNOW1 as \( A_1 \). We can determine the correct signs (negated variable or not) for all variables by finding a single negative example. Predict positive until we get a negative example (one mistake at most). Let \( (z_1, \ldots, z_n) \) be the negative example. All future instances are sent to WINNOW1 using

\[
T_i(x_1, \ldots, x_n) = (x_1 \oplus z_1, \ldots, x_n \oplus z_n).
\]

Such a mapping makes sense, since if \( z_i \) is off in the negative example, it cannot be a negated variable, and vice versa, so only the negated variables are negated by the mapping. (What is done to irrelevant variables does not matter.) Finally, let \( T_p \) be the identity function. It is easily shown that the conditions of Theorem 12.7 are satisfied. So the mistake bound for learning non-monotone disjunctions with this technique is just one more than the corresponding mistake bound for learning monotone disjunctions.

Example 12.2 Arbitrary Conjunctions

The example of arbitrary disjunctions can be extended to learning \( k \)-literal monotone conjunctions. Let \( A_2 \) be the algorithm described in the above example, let \( T_i = (1 - x_1, \ldots, 1 - x_n) \), and let \( T_p(r) = 1 - r \). Again, using DeMorgan’s law it is easily verified that the conditions of Theorem 12.7 hold. Thus, the number of mistakes will be bounded by \( 2k \log_2 n + 2 \).

Example 12.3 \( k \)-DNF for fixed \( k \)

As another example, consider learning \( k \)-DNF for a fixed \( k \). We notice that this class is more difficult, since it is not a linearly separable class. Let the original algorithm be WINNOW1 over \( k \)-literal disjunctions. Let \( n_1 \) be the number of variables in the derived concept class, and let \( n_2 \), the number of variables over which the original algorithm is run, be set to \( n_2 = \sum_{i=0}^{k} 2^i \binom{n}{i} \). Then we have that

\[
T_i(x) = (c_1(x), \ldots, c_{n_2}(x))
\]

where \( x = (x_1, \ldots, x_{n_1}) \) and the \( c_i \)'s range over all possible conjunctions which could be terms in a \( k \)-DNF formula. Notice that given any \( k \)-DNF we can represent it as such. If such a \( g \) is given we can construct an \( f \) in the derived space which is a disjunction of all the variables whose corresponding terms are in the expansion of \( g \).

By expanding the summation for \( n_2 \), we can see that \( n_2 \leq (2n)^k + 1 \). Since the original algorithm will be learning an \( l \)-literal monotone disjunction, we will make \( O(l \log n^k) = \)
$O(\log n)$ mistakes. Compare this to the algorithm Valiant presented which can be forced to make $\binom{n}{k} - l$ mistakes in the worst case.

By using the VC dimension, we can find a lower bound on the mistake bound. Taking $m = \lceil kl^{1/k} \rceil$, we have

$$\binom{m}{k} \geq \frac{m^k}{k^k} \geq 1$$

Thus a lower bound would be given, when $kl^{1/k} \leq n$, by:

$$kl \left\lfloor \log_2 \frac{n}{[k/l^{1/k}]} \right\rfloor$$

If, however, $l$ is known, an even better bound of $4l + 2kl \log_2 \left( \frac{2n}{l^{1/k}} \right)$ can be achieved.
13.1 Introduction

In this lecture we consider the problem of learning an unknown regular set using membership and equivalence queries. The material presented in this lecture comes from the paper, “Learning Regular Sets from Queries and Counterexamples,” by Dana Angluin [2]. Since deterministic finite-state acceptors (dfa’s) recognize the same languages as regular sets, this is equivalent to the problem of learning an unknown dfa. We will express our hypothesis in the form of a dfa by giving the initial state, final (accepting) states and transition function. Our goal will be to find a dfa with the minimum number of states that recognizes the unknown regular set.

Gold [15] has shown that finding a minimum dfa consistent with an arbitrary set of positive and negative examples is NP-hard. The learning algorithm has the advantage of being able to select the examples for the membership queries, thus the set of examples used to help construct the dfa is not arbitrary. To be certain we are not asking for too much with membership and equivalence queries, we would like both types of queries to be computable in time polynomial in the number of states in a minimum dfa recognizing the regular set and polynomial in the length of the hypothesis dfa. The teacher has access to the minimum dfa, thus a membership query can be answered by simply tracing the dfa using the given string. In addition, there is a polynomial time algorithm for testing the equivalence of two dfa’s which returns a counterexample if the dfa’s are not equivalent.

13.2 The Learning Algorithm

In the following discussion we use the notation $L^*$ to denote the learning algorithm, $U$ to denote the unknown regular set to be learned, and $A$ to denote the alphabet of the regular language. (The alphabet is known to the learner.)

The learning algorithm develops a hypothesis by using membership queries to determine whether or not certain strings are in the set $U$. The results of these queries are stored in an observation table maintained by the learner. Periodically the learner will construct a hypothesis dfa from the observation table and perform an equivalence query to see if the hypothesis is correct. If the hypothesis is not correct then the counterexample will be used to modify the observation table. Next we describe the structure of the observation table. Later we will see how the hypothesis dfa is constructed from the table.
The observation table represents a mapping $T$ from a set of finite strings to $\{0,1\}$. The function $T$ is such that $T(u) = 1$ if and only if $u \in U$. The strings in the table are formed by concatenating an element from the set $S \cup S \cdot A$ with an element from the set $E$, where $S$ is a nonempty finite prefix-closed set of strings and $E$ is a nonempty finite suffix-closed set of strings. $S \cdot A = \{ s \cdot a : s \in S, a \in A \}$, where $\cdot$ is the concatenation operator. The table can be represented by a two-dimensional array with rows labeled by elements of $S \cup S \cdot A$ and columns labeled by elements of $E$. The entry in row $s \in S \cup S \cdot A$ and column $e \in E$ contains $T(s \cdot e)$. The observation table will be denoted $(S,E,T)$.

In the initial observation table, $S = E = \{ \lambda \}$, where $\lambda$ represents the empty string. This table has one column and $1 + |A|$ rows. For example, if $A = \{ a, b \}$ and the regular language $U = \{ u : u \text{ contains an even number of both } a \text{'}s \text{ and } b \text{'}s \}$ then Table 13.1 is the initial table. The double horizontal line separates the rows labeled with elements of $S$ from the rows labeled with elements of $S \cdot A$. We will denote the row of the table labeled by $s \in S \cup S \cdot A$ by $\text{row}(s)$. In the example, $\text{row}(a) = (0)$.

We define two properties of an observation table. An observation table is closed if for all $t \in S \cdot A$, there exists an $s \in S$ such that $\text{row}(t) = \text{row}(s)$. An observation table is consistent if whenever $s_1, s_2 \in S$ satisfy $\text{row}(s_1) = \text{row}(s_2)$, then for all $a \in A$, $\text{row}(s_1 \cdot a) = \text{row}(s_2 \cdot a)$. Table 13.1 is not closed since $s \in S \cdot A$ and $\text{row}(a) = (0)$, but there is no $s \in S$ such that $\text{row}(s) = (0)$. The table is consistent.

We now define the dfa, denoted $M(S,E,T)$, corresponding to the closed, consistent observation table $(S,E,T)$. (The observation table must be closed and consistent otherwise the dfa is undefined.) $M(S,E,T)$ is the acceptor over alphabet $A$, with state set $Q$, initial state $q_0$, accepting state set $F$ and transition function $\delta$ where:

\[
Q = \{ \text{row}(s) : s \in S \} \\
q_0 = \text{row}(\lambda) \\
F = \{ \text{row}(s) : s \in S, T(s) = 1 \} \\
\delta(\text{row}(s), a) = \text{row}(s \cdot a)
\]

We can show that $M(S,E,T)$ is a well defined acceptor. First, it is always true that $\lambda \in S$ since $S$ is nonempty and prefix-closed. Thus $q_0 = \text{row}(\lambda)$ is well defined. Second, $F = \{ \text{row}(s) : s \in S, T(s) = 1 \}$ is well defined. For $F$ to be ill defined there must exist $s_1, s_2 \in S$ such that $\text{row}(s_1) = \text{row}(s_2)$ but $T(s_1) \neq T(s_2)$. Since $E$ is nonempty and suffix-closed, $\lambda \in E$. Thus $\text{row}(s_1)$ contains $T(s_1 \cdot \lambda) = T(s_1)$, and $\text{row}(s_2)$ contains $T(s_2 \cdot \lambda) = T(s_2)$. Since $\text{row}(s_1) = \text{row}(s_2)$, it must be true that $T(s_1) = T(s_2)$. Thus, $F$ is well defined. Third, $\delta(\text{row}(s), a) = \text{row}(s \cdot a)$ is well defined. There are two ways for $\delta$ to be ill defined. First,
there could exist \( s_1, s_2 \in S \) such that \( \text{row}(s_1) = \text{row}(s_2) \) but \( \text{row}(s_1 \cdot a) \neq \text{row}(s_2 \cdot a) \). Second, it could be that \( \text{row}(s \cdot a) \) is not in \( Q \). Because the table is consistent, the first condition cannot occur. Because the table is closed, the second condition cannot occur. Thus \( \delta \) is well defined.

We now give the learning algorithm \( L^* \), which maintains an observation table \( (S, E, T) \). The table is modified to reflect responses to membership and equivalence queries. The hypothesis dfa posed by each equivalence query is the dfa \( M(S, E, T) \) corresponding to the current observation table.

**Algorithm \( L^* \)**

Initialize \( S \) and \( E \) to \( \{\lambda\} \).
Ask membership queries for \( \lambda \) and \( a, \forall a \in A \).
Construct the initial observation table \( (S, E, T) \).
Repeat:
   While \( (S, E, T) \) is not closed or not consistent:
   If \( (S, E, T) \) is not consistent,
      find \( s_1, s_2 \in S, a \in A, e \in E \) such that \( \text{row}(s_1) = \text{row}(s_2) \)
      and \( T(s_1 \cdot a \cdot e) \neq T(s_2 \cdot a \cdot e) \),
      add \( a \cdot e \) to \( E \),
      extend \( T \) to \( (S \cup S \cdot A) \cdot E \) using membership queries.
   If \( (S, E, T) \) is not closed,
      find \( s_1 \in S, a \in A \) such that \( \text{row}(s_1 \cdot a) \neq \text{row}(s) \forall s \in S \),
      add \( s_1 \cdot a \) to \( S \),
      extend \( T \) to \( (S \cup S \cdot A) \cdot E \) using membership queries.
Perform an equivalence query with \( M = M(S, E, T) \).
If answer is "no" with counterexample \( t \),
   add \( t \) and its prefixes to \( S \),
   extend \( T \) to \( (S \cup S \cdot A) \cdot E \) using membership queries.
Until answer is "yes" from equivalence query.

We note that the runtime of the algorithm depends upon the length of the longest counterexample, since the table must include each counterexample \( t \) and all its prefixes. The algorithm will never remove a row or column from the table. The only adjustments to the table are the addition of rows and columns. Also, the algorithm must test the closure and consistency of the current observation table. The closure is easy to check; the consistency is a bit more time consuming, but not too difficult. Before analyzing the correctness of the algorithm we consider an example execution of \( L^* \).

### 13.3 An Example

97
\begin{center}
\begin{tabular}{|c|c|}
\hline
$T_1$ & $\lambda$ \\
\hline
$\lambda$ & 1 \\
\hline
$a$ & 0 \\
\hline
$b$ & 0 \\
\hline
\end{tabular}
\end{center}

Table 13.2: Initial observation table.

\begin{center}
\begin{tabular}{|c|c|}
\hline
$T_2$ & $\lambda$ \\
\hline
$\lambda$ & 1 \\
\hline
$a$ & 0 \\
\hline
$b$ & 0 \\
\hline
$aa$ & 1 \\
\hline
$ab$ & 0 \\
\hline
\end{tabular}
\end{center}

Table 13.3: Second observation table.

In this section we trace the execution of $L^*$ as it learns the regular set $U$ defined in the previous section. That is, $U = \{u : u \text{ contains an even number of both } a\text{'s and } b\text{'s}\}$. The alphabet $A = \{a, b\}$. The initial table was given in the previous section and is repeated as Table 13.2. To distinguish the versions of the observation table as the algorithm progresses, we label the $i^{th}$ table in the upper left corner with $T_i$.

We noted earlier that this table is consistent, but not closed since $a \in S \cup A$ but $\text{row}(a) = 0 \neq \text{row}(s)$ for any $s \in S$. Accordingly, $L^*$ moves $a$ to $S$, extends $S \cup A$ to include extensions of $a$ and fills in the new entries in the first column of the table. Table 13.3 is the second observation table.

This table is closed and consistent, so the algorithm poses an equivalence query with the dfa $M(S, E, T)$. This dfa has two states, $q_0 = \text{row}(\lambda)$ and $q_1 = \text{row}(a)$. The starting state is $q_0$, the accepting state is $q_0$ and the transition function is as shown in Table 13.4. To distinguish the various hypothesis dfa's, we label the transition function corresponding to observation table $i$ with $\delta_i$.

This dfa does not recognize $U$ thus the equivalence query returns a counterexample $t$. Let us assume the counterexample is $t = bb$. It is easy to see that $t \in U$, but $t$ is not accepted by the hypothesis dfa. Accordingly, $L^*$ adds $bb$ and all its prefixes to $S$ and adds the extensions of $bb$ to $S \cdot A$. $L^*$ then fills in the new entries in the first column. Table 13.5 is the resulting observation table.

The third observation table is closed, but not consistent, since $\text{row}(a) = \text{row}(b)$ but

\begin{center}
\begin{tabular}{|c|c|c|}
\hline
$\delta_2$ & $a$ & $b$ \\
\hline
$q_0$ & $q_1$ & $q_1$ \\
\hline
$q_1$ & $q_0$ & $q_1$ \\
\hline
\end{tabular}
\end{center}

Table 13.4: Dfa for second observation table.
Table 13.5: Third observation table.

<table>
<thead>
<tr>
<th>$T_3$</th>
<th>$\lambda$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\lambda$</td>
<td>1</td>
</tr>
<tr>
<td>$a$</td>
<td>0</td>
</tr>
<tr>
<td>$b$</td>
<td>0</td>
</tr>
<tr>
<td>$bb$</td>
<td>1</td>
</tr>
<tr>
<td>$aa$</td>
<td>1</td>
</tr>
<tr>
<td>$ab$</td>
<td>0</td>
</tr>
<tr>
<td>$ba$</td>
<td>0</td>
</tr>
<tr>
<td>$bba$</td>
<td>0</td>
</tr>
<tr>
<td>$bbb$</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 13.6: Fourth observation table.

<table>
<thead>
<tr>
<th>$T_4$</th>
<th>$\lambda$</th>
<th>$a$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\lambda$</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>$a$</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>$b$</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$bb$</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>$aa$</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>$ab$</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$ba$</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$bba$</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>$bbb$</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
Table 13.7: Dfa for fourth observation table.

<table>
<thead>
<tr>
<th>$\delta_4$</th>
<th>$a$</th>
<th>$b$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$q_0$</td>
<td>$q_1$</td>
<td>$q_2$</td>
</tr>
<tr>
<td>$q_1$</td>
<td>$q_0$</td>
<td>$q_2$</td>
</tr>
<tr>
<td>$q_2$</td>
<td>$q_2$</td>
<td>$q_0$</td>
</tr>
</tbody>
</table>

Table 13.8: Fifth observation table.

<table>
<thead>
<tr>
<th>$T_5$</th>
<th>$\lambda$</th>
<th>$a$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\lambda$</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>$a$</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>$b$</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$bb$</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>$ab$</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$abb$</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>$aa$</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>$ba$</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$bba$</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>$bbb$</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$aba$</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$abba$</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>$abbb$</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

row($aa$) $\neq$ row($ba$). The algorithm adds $a$ to $E$ and fills in the values of $T$ for the new column. Table 13.6 is the result.

Table 13.6 is a closed, consistent observation table. The algorithm constructs the dfa $M(S, E, T)$, which has three states, $q_0 = \text{row}(\lambda)$, $q_1 = \text{row}(a)$ and $q_2 = \text{row}(b)$. The starting state is $q_0$, the accepting state is $q_0$ and the transition function is as shown in Table 13.7.

This dfa is not quite correct, thus the equivalence query returns a counterexample $t$. Let us assume the counterexample is $t = abb$, which is not in $U$ but is accepted by the dfa. The algorithm adds $abb$ and all its prefixes and extensions to the observation table and fills in the values of $T$. Table 13.8 shows the result.

Table 13.8 is closed, but not consistent since row($b$) = row($ab$) but row($bb$) $\neq$ row($abb$). The algorithm adds $b$ to $E$ and fills in the values of $T$ for this new column. The result is Table 13.9.

This table is closed and consistent, so the algorithm constructs $M(S, E, T)$. This dfa has four states, $q_0 = \text{row}(\lambda)$, $q_1 = \text{row}(a)$, $q_2 = \text{row}(b)$ and $q_3 = \text{row}(ab)$. The starting state is $q_0$, the accepting state is $q_0$ and the transition function is as shown in Table 13.10.

This dfa accepts exactly the language $U$, thus the equivalence query returns “yes” and the algorithm halts. This is the minimum size dfa to recognize the language consisting of all strings with an even number of $a$'s and an even number of $b$'s.
<table>
<thead>
<tr>
<th>$T_6$</th>
<th>$\lambda$</th>
<th>$a$</th>
<th>$b$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\lambda$</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$a$</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>$b$</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>$bb$</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$ab$</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$abb$</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>$aa$</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$ba$</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$bba$</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>$bbb$</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>$aba$</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>$abba$</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$abbb$</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 13.9: Sixth observation table.

<table>
<thead>
<tr>
<th>$\delta_6$</th>
<th>$a$</th>
<th>$b$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$q_0$</td>
<td>$q_1$</td>
<td>$q_2$</td>
</tr>
<tr>
<td>$q_1$</td>
<td>$q_0$</td>
<td>$q_3$</td>
</tr>
<tr>
<td>$q_2$</td>
<td>$q_3$</td>
<td>$q_0$</td>
</tr>
<tr>
<td>$q_3$</td>
<td>$q_2$</td>
<td>$q_1$</td>
</tr>
</tbody>
</table>

Table 13.10: Dfa for sixth observation table.
13.4 Algorithm Analysis

There are three requirements of the algorithm which we address in this section. First, we show that the algorithm is correct, that is, $L^*$ finds a minimum dfa to recognize $U$. In addressing this issue we assume the algorithm terminates and show that, assuming termination, the algorithm finds a minimum dfa to recognize $U$. This leads to the second issue, showing that the algorithm terminates. Third, we show that the time complexity of the algorithm is polynomial in the number of states in the minimum dfa and polynomial in the length of the longest counterexample. We consider these requirements in the next three subsections.

13.4.1 Correctness of $L^*$

Assuming $L^*$ terminates, the claim that $L^*$ produces a dfa that recognizes $U$ is trivial. The condition for termination is that the equivalence query returns “yes”, indicating that the dfa recognizes $U$. The claim that $L^*$ produces a minimum dfa to recognize $U$ is more complicated. The key to this claim is the following theorem about the acceptor $M(S, E, T)$ constructed from a closed, consistent observation table $(S, E, T)$.

**Theorem 13.1** If $(S, E, T)$ is a closed, consistent observation table, then the acceptor $M(S, E, T)$ is consistent with the finite function $T$. Any other acceptor consistent with $T$ but inequivalent to $M(S, E, T)$ must have more states.

**Proof:** This theorem is proven by several lemmas.

**Lemma 13.1** Assume that $(S, E, T)$ is a closed, consistent observation table. For the acceptor $M(S, E, T)$ and every $s \in (S \cup S \cdot A)$, $\delta(q_0, s) = \text{row}(s)$.

**Proof:** The proof is by induction on the length of $s$.

The base case is $|s| = 0$ implying that $s = \lambda$. By definition, $q_0 = \text{row}(\lambda)$, thus

$$\delta(q_0, s) = \delta(\text{row}(\lambda), \lambda) = \text{row}(\lambda) = \text{row}(s).$$

For the induction step we assume the lemma holds for $|s| \leq k$ and show it holds for strings of length $k + 1$. Let $t \in (S \cup S \cdot A)$ with $|t| = k + 1$. Clearly, $t = s \cdot a$ for some string $s$ of length $k$ and some $a \in A$. By the construction of the observation table, $s \in S$. Thus,

$$\delta(q_0, t) = \delta(q_0, s \cdot a),$$

$$= \delta(\delta(q_0, s), a),$$

$$= \delta(\text{row}(s), a),$$

by induction hypothesis,

$$= \text{row}(s \cdot a),$$

by definition of $\delta$,

$$= \text{row}(t).$$

**Lemma 13.2** Assume that $(S, E, T)$ is a closed, consistent observation table. The acceptor $M(S, E, T)$ is consistent with the finite function $T$. That is, for every $s \in (S \cup S \cdot A)$ and $e \in E$, $\delta(q_0, s \cdot e) \in F$ if and only if $T(s \cdot e) = 1$.

102
**Proof:** The proof is by induction on the length of $e$.

In the base case, $|e| = 0$, thus $e = \lambda$. By the preceding lemma, $\delta(q_0, s \cdot e) = \text{row}(s)$. If $s \in S$ then by the definition of $F$, $\text{row}(s) \in F$ if and only if $T(s) = 1$. If $s \in S \cup A$ then since the table is closed, $\text{row}(s) = \text{row}(s_1)$ for some $s_1 \in S$. Now $\text{row}(s_1) \in F$ if and only if $T(s_1) = 1$, which is true if and only if $T(s) = 1$, since $\text{row}(s) = \text{row}(s_1)$.

In the induction step we assume the lemma holds for all $e$ with $|e| \leq k$. Let $e \in E$ with $|e| = k + 1$. Since $E$ is suffix closed, $e = a \cdot e_1$ for some $a \in A$ and some $e_1 \in E$ of length $k$. Let $s$ be any element of $(S \cup S \cdot A)$. Because the observation table is closed, there exists a string $s_1 \in S$ such that $\text{row}(s) = \text{row}(s_1)$. Then,

\[
\delta(q_0, s \cdot e) = \delta(\delta(q_0, s), a \cdot e_1), \\
= \delta(\text{row}(s), a \cdot e_1), \quad \text{by preceding lemma}, \\
= \delta(\text{row}(s_1), a \cdot e_1), \quad \text{since row}(s) = \text{row}(s_1), \\
= \delta(\delta(\text{row}(s_1), a), e_1), \\
= \delta(\delta(s_1, a), e_1), \quad \text{by definition of } \delta, \\
= \delta(\delta(q_0, s_1 \cdot a), e_1), \quad \text{by preceding lemma}, \\
= \delta(q_0, s_1 \cdot a \cdot e_1).
\]

By the induction hypothesis on $e_1$, $\delta(q_0, s_1 \cdot a \cdot e)$ is in $F$ if and only if $T(s_1 \cdot a \cdot e) = 1$. Since $\text{row}(s) = \text{row}(s_1)$ and $a \cdot e_1 = e$ is in $E$, $T(s_1 \cdot a \cdot e) = T(s \cdot a \cdot e) = T(s \cdot e)$. Therefore, $\delta(q_0, s \cdot e) \in F$ if and only if $T(s \cdot e) = 1$, as claimed by the lemma.

---

**Lemma 13.3** Assume that $(S, E, T)$ is a closed, consistent observation table. Suppose $M(S, E, T)$ has $n$ states. If $M' = (Q', q_0', F', \delta')$ is any dfa consistent with $T$ that has $n$ or fewer states, then $M'$ is isomorphic to $M(S, E, T)$.

**Proof:** The proof consists of exhibiting an isomorphism.

For each $q' \in Q'$, define $\text{row}(q')$ to be the function $f$ from $E$ to $\{0, 1\}$ such that $f(e) = 1$ if and only if $\delta'(q', e) \in F'$.

Since $M'$ is consistent with $T$, for each $s \in (S \cup S \cdot A)$ and each $e \in E$, $\delta'(q_0', s \cdot e) \in F'$ if and only if $T(s \cdot e) = 1$. Also, since $\delta'(q_0', s \cdot e) = \delta'(\delta'(q_0', s), e)$ we know that $\delta'(\delta'(q_0', s), e) \in F'$ if and only if $T(s \cdot e) = 1$. Therefore, $\text{row}(\delta'(q_0', s)) = \text{row}(s)$ in $M(S, E, T)$. As $s$ ranges over all of $S$, $\text{row}(\delta'(q_0', s))$ ranges over all elements of $Q$, implying that $M'$ must have at least $n$ states. Since the statement of the lemma presumed that $M'$ had $n$ or fewer states we can conclude that $M'$ has exactly $n$ states.

Thus, for each $s \in S$ there is a a unique $q' \in Q'$ for which $\text{row}(s) = \text{row}(q')$, namely, $\delta'(q_0', s)$. We can define a one-to-one and onto mapping between the states of $M(S, E, T)$ and the states $Q'$ of $M'$. Specifically, for each $s \in S$, we define $\phi(\text{row}(s)) = \delta'(q_0', s)$. It remains to be shown that this mapping takes $q_0$ to $q_0'$, that it preserves the transition function and that it takes $F$ to $F'$. Each of these can be verified in a straightforward way using the definitions above. The details can be found in the paper by Angluin [2].

\[103\]
Lemma 2 and Lemma 3 together prove the two parts of Theorem 1. Namely, Lemma 2 shows that $M(S, E, T)$ is consistent with $T$ and Lemma 3 shows that any dfa consistent with $T$ is either isomorphic to $M(S, E, T)$ or has more states. Thus $M(S, E, T)$ is a smallest dfa consistent with $T$.

13.4.2 Termination of $L^*$

In the previous subsection we ignored the question of whether or not $L^*$ will terminate and concentrated on showing that if it terminates, the output will be correct. In this subsection we address the issue of termination.

To see that the algorithm will terminate we need the following lemma. At first glance the lemma appears similar to the last lemma of the previous section. The key difference is that the following lemma does not assume that the observation table is closed and consistent. It is important to consider this more general case because as the algorithm proceeds, there will be times when the observation table is not closed and consistent.

Lemma 13.4 Let $(S, E, T)$ be an observation table. Let $n$ denote the number of different values of row($s$) for $s \in S$. (Note that if $(S, E, T)$ is closed and consistent, then this will be the number of states in $M(S, E, T)$.) Any dfa consistent with $T$ must have at least $n$ states.

Proof:

Let $M = (Q, \delta, q_0, F)$ be a dfa consistent with $T$. Define $f(s) = \delta(q_0, s)$ for every $s \in S$. In other words, $f(s)$ is the final state of $M$ when run with input $s$. Suppose $s_1$ and $s_2$ are elements of $S$ such that row($s_1$) $\neq$ row($s_2$). Then there exists an $e \in E$ such that $T(s_1 \cdot e) \neq T(s_2 \cdot e)$. Since $M$ is consistent with $T$, exactly one of $\delta(q_0, s_1 \cdot e)$ and $\delta(q_0, s_2 \cdot e)$ is in $F$. Thus, $\delta(q_0, s_1 \cdot e)$ and $\delta(q_0, s_2 \cdot e)$ must be distinct states, implying that $f(s_1 \cdot e) \neq f(s_2 \cdot e)$. Since there are $n$ different values of row($s$), $f(s)$ must take on at least $n$ distinct values. Thus $M$ has at least $n$ states.

Let $n$ be the number of states in a minimum dfa $M_U$ for the unknown language $U$. To prove termination we show that the number of distinct values of row($s$) for $s \in S$ increases monotonically up to $n$ as $L^*$ runs.

First, consider what happens when a string is added to $E$ because the table is not consistent. The number of distinct values of row($s$) must increase by at least one. Two previously equal values, row($s_1$) and row($s_2$), are no longer equal after $E$ is augmented. Any two unequal values will remain unequal.

Second, consider what happens when a string $s_1 \cdot a$ is added to $S$ because the table is not closed. By definition, row($s_1 \cdot a$) differs from row($s$) for all $s \in S$. Thus the number of distinct values of row($s$) is increased by at least one.

From these two situations we can conclude that the total number of operations of either type over the entire run of $L^*$ is at most $n - 1$. (There is initially one value of row($s$) and there cannot be more than $n$.) Thus the algorithm will enter the while loop at most $n - 1$ times. This means that $L^*$ always eventually finds a closed, consistent observation table $(S, E, T)$ and makes an equivalence query with $M(S, E, T)$.
We must show that the number of equivalence queries is limited (i.e., that the algorithm
does not get stuck in the repeat loop). If an equivalence query $M(S, E, T)$ is incorrect
with counterexample $t$, then by Theorem 1, $M_U$ must have at least one more state than
$M(S, E, T)$. Furthermore, $L^*$ must eventually make another equivalence query $M(S', E', T')$
which is consistent with $T$ (since $T'$ extends $T$) and also classifies $t$ the same as $M_U$ (since
t $t \in S'$ and $\lambda \in E$). This implies that $M(S', E', T')$ is inequivalent to $M(S, E, T)$ and thus
has at least one more state that $M(S, E, T)$.

From this we conclude that $L^*$ can make at most $n - 1$ incorrect equivalence queries,
since the number of states in the successive queries is monotonically increasing from one
and cannot exceed $n - 1$. Since $L^*$ will eventually make another equivalence query, it will
terminate with a correct query.

13.4.3 Runtime complexity of $L^*$

The runtime of the algorithm depends in part of the length of the longest counterexample.
We let $m$ be the length of the longest counterexample and analyze the runtime in terms of
$m$ and $n$, the number of states in a minimum dfa for the unknown language. In addition,
we let $k$ denote the cardinality of the alphabet $A$.

First we determine the space needed by the observation table. Initially, $|S| = |E| = 1$. Each
time $(S, E, T)$ is discovered to be not closed, $|S| \rightarrow |S| + 1$. Each time $(S, E, T)$ is
discovered to be not consistent, $|E| \rightarrow |E| + 1$. For each counterexample of length at most
$m$, at most $m$ strings are added to $S$.

From this and the analysis of the termination of $L^*$ we see that $|E| \leq n$ and for all $e \in E$,
$|e| \leq n - 1$. Also, $|S| \leq n + m(n - 1)$. The first term results because the observation table
may be discovered to be not closed at most $n - 1$ times. The second term results because
there may be at most $n - 1$ counterexamples, each of which adds at most $m$ strings to $S$.

The maximum length of any string in $S$ is increased by one each time the table is found not
to be closed. Thus for all $s \in S$, $|s| \leq m + n - 1$.

Thus, the table size, $|(S \cup S \cdot A) \cdot E|$ is at most

$$(k + 1)(n + m(n - 1))n = O(mn^2).$$

The maximum length of any string in $(S \cup S \cdot A) \cdot E$ is at most

$$(m + n - 1) + 1 + n - 1 = m + 2n - 1 = O(m + n).$$

Thus the observation table takes space $O(m^2n^2 + mn^3)$.

Now we determine the time needed for the computation performed by $L^*$. Checking if the
observation table is closed and consistent can be done in time polynomial in the size of the
table. This is done at most $n - 1$ times. Adding a string to $S$ or $E$ requires at most $O(mn)$
membership queries of strings of length at most $O(m + n)$. The total number of membership
queries is $O(mn^2)$. Given a closed and consistent table, $M(S, E, T)$ can be constructed in
time polynomial in the size of the table. This must be done at most $n - 1$ times. Thus the
computation time is polynomial.
Finally, note that if the counterexamples are always of the minimum possible length then $m \leq n$ and all results are polynomial in $n$. The algorithm which tests for the equivalence of two DFA's can identify a counterexample of minimum length. Also, the bound on the number of membership queries was improved to $O(kn^2 + n \log m)$ by Rivest and Schapire [36].
14.1 Introduction

In these notes we more carefully study the relationship between the Vapnik-Chervonenkis Dimension and the sample complexity of PAC learning. The material presented here comes from the paper "Results on Learnability and the Vapnik-Chervonenkis Dimension," by Nathan Linial, Yishay Mansour, and Ronald Rivest [28].

Recall in Topic 7 that we studied a result of Blumer, et al. [12] stating that a concept class $C$ which is non-trivial and well-behaved is PAC learnable if and only if the VC dimension of $C$ is finite. Here we demonstrate that by using dynamic sampling versus static sampling that one can PAC learn some concept classes with infinite VC dimension. In other words, the Blumer, Ehrenfeucht, Haussler and Warmuth result assumes that the learner asks for a single sample of a given size and must then processes these examples. If instead the learner can divide the learning session into stages where in each stage the learner asks for some examples and then performs some processing, then it is possible to learn any concept class that can be written as the countable union

$$C = C_1 \cup C_2 \cup \cdots$$

where each concept class $C_d$ has VC-dimension at most $d$.

In these notes we shall say that a concept class $C$ is PAC learnable if there exists a learning algorithm that outputs a hypothesis meeting the PAC criterion. We say a concept class $C$ is polynomially PAC learnable if it uses time and sample complexity that are polynomial in $\frac{1}{\epsilon}$ and $\frac{1}{\delta}$.

All previous PAC algorithms that we have studied assume a static sampling model in which the learning algorithm must draw all examples before any computation is performed. In addition to using a static sampling model, typically PAC learning algorithms are consistent in that the concept $C$ they return agrees with the classification of each example of the sample.

Using this notation, we will restate the (Blumer et al. [12]) result as below.

**Theorem 14.1** A concept class $C$ is PAC learnable with static sampling if and only if $C$ has finite VC-dimension. Furthermore the bounds of the sample complexity are:

$$m(\epsilon, \delta) = O\left(\frac{1}{\epsilon} \ln \frac{1}{\delta} + \frac{d}{\epsilon} \ln \frac{1}{\epsilon}\right)$$

and

$$m(\epsilon, \delta) = \Omega\left(\frac{1}{\epsilon} \ln \frac{1}{\delta} + \frac{d}{\epsilon}\right)$$
Based on these bounds, Blumer et al. also showed many concept classes to be polynomially PAC learnable. More generally, if $C$ is a class with finite VC dimension and there exists a polynomial time algorithm to find a concept in $C$ that is consistent with a given sample, then $C$ is polynomially PAC learnable.

### 14.2 Dynamic Sampling

Now we consider the notion of dynamic sampling, in which the number of examples examined increases with the complexity of target concept. It turns out that dynamic sampling does indeed enrich the class of PAC learnable concepts, compared to static sampling. By this method, one can establish the learnability of various concept class with an infinite Vapnik-Chervonenkis dimension.

When using dynamic sampling, the PAC learning algorithm alternates between drawing examples and doing computations. A stage of the PAC learning algorithm consists of drawing a set examples and performing the subsequent computations. Thus a static sampling algorithm is equivalent to a one-stage dynamic sampling algorithm. We note that PAC learning a class with infinite VC dimension may require an unbounded number of stages.

### 14.3 Learning Enumerable Concept Classes

We first define an enumerable concept class and then prove that any such class is PAC learnable using dynamic sampling. This illustrates the power of dynamic sampling relative to static sampling, since an enumerable class of concepts may have infinite VC dimension.

Let $C = \{C_1, C_2, \ldots, \}$ be a recursively enumerable concept class, such that for each $C_i$, membership in $C_i$ is decidable. Note that $C$ may have infinite VC-dimension, for example, let $N$ be the set of natural numbers, and $C$ be the set of all finite subsets of $N$.

Now we give the algorithm for PAC learning $C$.

**Algorithm Enumerable-Learner**

1. Let $i=1$.

2. Draw enough examples so that the total number $m_i$ of examples drawn so far is at least $\frac{1}{\epsilon} \ln \frac{2d}{\delta}$.

3. If $C_i$ is consistent with all examples seen so far then output $C_i$. Otherwise increase $i$ by 1 and return to step 2.

**Theorem 14.2** *Enumerable-Learner can PAC learn any enumerable concept class $C$.*

**Proof:** Let $T$ be the target concept. Recall the concept $C_i$ is “$\epsilon$-bad” if $P(T \oplus C_i) > \epsilon$. The probability that an $\epsilon$-bad concept $C_i$ is output is at most $(1 - \epsilon)^{m_i}$. For $m_i \geq \frac{1}{\epsilon} \ln \frac{2d}{\delta}$,
\((1 - \epsilon)^{m_i} < \frac{\delta \cdot 6}{i^2 \pi^2}\) holds.

To show this, just note that

\[
m_i \ln(1 - \epsilon) < -\epsilon m_i
\]

Now substitute the lower bounds for \(m_i\), we get

\[
m_i \ln(1 - \epsilon) < \ln \frac{\delta}{2i^2} < \ln \frac{\delta}{2i^2} + \ln \frac{12}{\pi^2} < \ln \left(\frac{\delta \cdot 6}{i^2 \pi^2}\right)
\]

therefore

\[
(1 - \epsilon)^{m_i} < \frac{\delta \cdot 6}{i^2 \pi^2}
\]

Since

\[
\sum_{i=1}^{\infty} \frac{1}{i^2} = \frac{\pi^2}{6}
\]

the probability that Enumerable-Learner outputs an \(\epsilon\)-bad concept is at most

\[
\frac{6 \cdot \delta}{\pi^2 \sum_{i=1}^{\infty} \frac{i}{i^2}} = \delta.
\]

\[\blacksquare\]

### 14.4 Learning Decomposable Concept Classes

The result just described does not handle the uncountable concept classes. To learn such classes we introduce the notion of a decomposable concept class.

**Definition 14.1** A concept class \(C\) is decomposable if it can be written as a countable union, \(C = C_1 \cup C_2 \cup \ldots\), where each concept class \(C_d\) has VC dimension at most \(d\).

In many cases, this decomposition can be done in such a way that \(C_i \subseteq C_{i+1}\) for all \(i\), and those concepts in \(C_d - C_{d-1}\) can naturally be said to have size \(d\). For example, if each concept is represented by a binary string, we might let \(C_d\) be the set of concepts whose binary encoding has at most \(d - 1\) bits. If \(X = [0, 1]\) and \(C\) is finite unions of subintervals of \(X\), then \(C_d\) is the set of concepts which are the union of at most \(d/2\) subintervals of \([0,1]\). In other cases, the "natural" size measure might be polynomially related to \(d_i\); the results can be easily extended to these cases.

Before describing the PAC learning algorithm for decomposable classes we need the following definitions.

**Definition 14.2** For target concept \(T \in C\), the size of the target concept \(T\) is defined as \(\text{size}(T) = \min\{d \mid T \in C_d\}\).
Now we want the complexity of learning algorithm to be polynomial in \( \text{size}(T) \), as well as \( \frac{1}{\varepsilon}, \frac{1}{T} \). Furthermore, we wish to have the PAC learning algorithm determine \( \text{size}(T) \) itself.

**Definition 14.3** A concept is uniformly decomposable if it is decomposable and there exists an algorithm \( A \), which given \( d \) and a sample can produce a concept \( c \in C_d \) consistent with the sample or else output "none" if no such concept exists. If \( A \) runs in time polynomial in \( d \) and the number of examples we say that \( C \) is polynomially uniformly decomposable.

**Theorem 14.3** Any uniformly decomposable concept class is PAC learnable using dynamic sampling. If \( C \) is polynomially uniformly decomposable, then the time and sample complexity are polynomial in the size of target concept. (The sample complexity polynomial in either case.)

**Proof:** First of all, we give the algorithm Uniformly-Decomposable-Learner, which can PAC learn any uniformly decomposable concept class.

**Algorithm Uniformly-Decomposable-Learner**

1. Let \( d=1 \).

2. Draw enough examples so that the total number \( m_d \) of examples drawn so far is at least \( \max \left( \frac{4}{\varepsilon} \ln \frac{8d^2}{\delta}, \frac{8d}{\varepsilon} \ln \frac{13}{\varepsilon} \right) \).

3. If there is a \( C' \in C_d \) which is consistent with all examples seen so far, then output \( C \). Otherwise increment \( d \) by 1 and return to step 2.

Now we prove the above algorithm will PAC learn all the uniformly decomposable concept classes.

The number of examples at each stage, \( m_d \), is chosen by the requirement from Blumer, et al. [12] with the probability that an \( \varepsilon \)-bad concept is output is at most \( \frac{\delta}{4d^2} \). Summing up \( \frac{\delta}{4d^2} \) over all \( d \), we get that the probability of an \( \varepsilon \)-bad concept to be output is at most \( \delta \).

Now we show that the algorithm will halt. Since at each stage \( d \) is incremented by 1, when \( d \) reaches \( \text{size}(T) \) (after \( \text{size}(T) \) steps), there is a concept in \( C_d \) (namely \( T \)), which is consistent with all the examples. At this point, algorithm will halt. Further more, the number of examples seen by the algorithm is polynomial in \( \text{size}(T) \). For the case that the concept classes that polynomially uniformly decomposable, the running of the algorithm is polynomial in \( \text{size}(T) \) as well.

As an illustration of the power of these techniques, the following classes are PAC learnable, even though they are uncountable and have infinite VC dimension.

**Theorem 14.4** The concept class \( C_{FI} \), defined as set of finite union of subintervals of \([0,1]\) is PAC learnable.
Proof: Decompose the concept class such that each $C_i$ includes concepts with at most $\frac{i}{2}$ subintervals (only defined for $i$ even). To show $C_{FI}$ is polynomially uniformly decomposable, we need to exhibit an algorithm, $A_{FI}$ that given a sample and index $d$, finds a concept in $C_d$ consistent with the samples (if one exists) or replies “none” (if such a concept does not exist).

The algorithm is based upon the observation that number of alternations (switches from “$+$” to “$-$” or “$-$” to “$+$”) in the sample is at most twice number of subintervals in target. (Technically to make this work, we have to assume that there is a “$-$” at the points 0 and 1. Another alternative is to look at the number of “blocks” of positive examples, where any two blocks are separated by at least 1 negative example.)

So if number of alternatives is greater than $d$, $A_{FI}$ outputs “none”, else it outputs a concept with minimal number of subintervals that is consistent with the sample.

Clearly concept with minimum number of subintervals that is consistent with an example is easy to find. Therefore we know that $C_{FI}$ is polynomially PAC learnable.

We also note that the concept class $C_{PR}$, the set of regions in $\mathbb{R}^2$, defined by an inequality of the form $y \leq f(x)$, where $f$ is any polynomial of finite degree with real coefficients, is PAC learnable.

### 14.5 Reducing the Number of Stages

In this section we briefly explore ways to reduce the number of stages in a PAC algorithm which uses dynamic sampling. Observe that in Uniformly-Decomposable-Learner the number of stages may be as large as $n = \text{size}(T)$. We now show that this can be improved for the concept class $C_{FI}$.

**Theorem 14.5** To PAC learn the concept class $C_{FI}$ the number of stages required is $\mathcal{O}(\lg \lg n)$.

Proof: In each stage update the value of $d$ to $d^2$ rather than $d + 1$. (Note that the decomposition of $C_{FI}$ is such that if $i < j$ then $C_i \subset C_j$.)

Furthermore, it can be shown that this bound for $C_{FI}$ is tight.

**Theorem 14.6** Any algorithm that PAC learns $C_{FI}$, with respect to the uniform distribution using a number of examples that is bounded by a polynomial in the number $n$ of subintervals of the target concept requires at least $\Omega(\log \log n)$ stages.

See the Linial, Mansour, Rivest paper for the proof of this result. Finally, we show that not every concept class of infinite VC dimension requires an unbounded number of stages.

**Theorem 14.7** Let $C_N$ denote the concept class of all subsets of the natural numbers. Then $C_N$ can be PAC learned with a two-stage learning algorithm.
Proof: In the first stage we draw a sample of size \((2/\epsilon)\log(2/\delta)\). Let \(M\) denote the largest integer appearing in this sample. With probability at least \(1 - \delta/2\) the probability associated with integers greater than \(M\) is at most \(\epsilon/2\). In the second stage we consider the induced problem of learning the restriction of the target concept to the natural numbers at most \(M\). This reduces the problem to one having a finite VC-dimension (i.e., \(M\)), which can be solved with a static sampling algorithm with parameters \(\epsilon/2\) and \(\delta/2\).

A simple generalization of this argument applies in a straightforward manner whenever the instance space is countable.
15.1 Introduction

Earlier in the course, we saw the different types of noises (errors) that can affect the training instances. The worst case is when these errors are generated by an adversary with unbounded computational resources, and knowledge of the function being learned, the probability distribution on the examples, and the internal state of the learning algorithm. The goal of the adversary is to foil the learning algorithm. This type of noise is known as malicious noise. With a fixed probability $\beta$ ($0 \leq \beta < 1$), the adversary gets to pick the instance and label.

After defining the required terms and notions, we shall obtain the upper and lower bounds on the noise rate that can be tolerated by any learning algorithm for a specified concept class $C$. Then, we will show the relationship between learning with errors and traditional complexity theory. This will be done by giving approximation-preserving reductions from standard optimization problems to natural learning problems, and vice-versa. The results obtained here also apply to learning under random misclassification noise covered earlier. The material presented in the lecture comes from the paper “Learning in the Presence of Malicious Errors,” by Michael Kearns and Ming Li [25]. Portions of the notes are taken from this paper.

15.2 Definitions and Notation

We shall be considering the 2-button model. For a fixed target concept $c \in C$, target distributions $D^+$ and $D^-$, and fixed $0 \leq \beta < 1$, we define two oracles with malicious errors as follows: in oracle $POS_M^\beta$, with probability $1 - \beta$, a point $x \in \text{pos}(c)$ is drawn according to $D^+$ and returned, and with probability $\beta$, an arbitrary point is returned and in oracle $NEG_M^\beta$, with probability $1 - \beta$, a point $x \in \text{neg}(c)$ is drawn according to $D^-$ and returned, and with probability $\beta$, an arbitrary point is returned.

$E_M(C)$, the optimal malicious error rate for a class of functions $C$, is the largest value of $\beta$ that can be tolerated by any learning algorithm (not necessarily polynomial time) for $C$. We will obtain upper bounds on this. $E_M(C)$ is the largest error rate tolerated by a polynomial time learning algorithm for $C$. We will obtain lower bounds on this by giving efficient learning algorithms.

A concept class $C$ over $X$ is distinct if there exist concepts $c_1, c_2 \in C$ and points $u, v, w, x \in X$ satisfying $u \in c_1, u \not\in c_2, v \in c_1, v \not\in c_2, w \not\in c_1, w \in c_2$, and $x \not\in c_1, x \not\in c_2$.  

113
15.3 Upper Bounds on $E_M(\mathcal{C})$

In this section, we will give a theorem bounding $E_M(\mathcal{C})$.

**Theorem 15.1** For a distinct concept class $\mathcal{C}$, $E_M(\mathcal{C}) < \frac{\epsilon}{1+\epsilon}$.

**Proof:** We will prove the theorem by using a technique called as the *method of induced distributions*: two or more concepts $\{c_i\} \subseteq \mathcal{C}$ are chosen, such that if $c_i$ is the target concept, then for all $i \neq j$, $c_j$ is $\epsilon$-bad. Then adversaries are given for generating errors such that regardless of which $c_i$ is the target concept, the behavior of the oracle $POS_M^\delta$ is identical, and similarly for the oracle $NEG_M^\delta$, thus making it impossible for the learning algorithm to distinguish the true target concept.

Coming to our theorem, we know that $\mathcal{C}$ is distinct. Let $c_1, c_2 \in \mathcal{C}$ and $u, v, w, x \in X$ as per the definition of distinct concept classes. Now consider the following table which gives the actual distribution $D_i$, the adversary’s choice in the case of noise and the induced distribution $I_i$ for target concept $c_i$.

<table>
<thead>
<tr>
<th>Actual distribution</th>
<th>Adversary’s choice if noise</th>
<th>Induced distribution</th>
</tr>
</thead>
<tbody>
<tr>
<td>$D_1(u) = \epsilon$</td>
<td>$D_1(v) = 1 - \epsilon$</td>
<td>$I_1(u) = (1 - \beta)\epsilon$ (\beta = \frac{\epsilon}{1+\epsilon})</td>
</tr>
<tr>
<td>$D_2(w) = \epsilon$</td>
<td>$D_2(x) = 1 - \epsilon$</td>
<td>$I_2(u) = \beta$ (\beta = \frac{\epsilon}{1+\epsilon})</td>
</tr>
<tr>
<td>$D_3(w) = \epsilon$</td>
<td>$D_3(v) = 1 - \epsilon$</td>
<td>$I_3(u) = (1 - \beta)\epsilon$ (\beta = \frac{\epsilon}{1+\epsilon})</td>
</tr>
<tr>
<td>$D_4(u) = \epsilon$</td>
<td>$D_4(x) = 1 - \epsilon$</td>
<td>$I_4(u) = (1 - \beta)\epsilon$ (\beta = \frac{\epsilon}{1+\epsilon})</td>
</tr>
</tbody>
</table>

From the table, we see that if $\beta = (1 - \beta)\epsilon$, then the induced distributions are identical. Also, $\beta = (1 - \beta)\epsilon \Rightarrow \beta = \frac{\epsilon}{1+\epsilon}$. Note that if $\beta > \frac{\epsilon}{1+\epsilon}$, then the adversary can choose to draw from the actual distribution some of the time, so that the effective error rate is $\frac{\epsilon}{1+\epsilon}$.

Thus we see that the learning algorithm can only handle $\beta < \frac{\epsilon}{1+\epsilon}$. Note that this result holds regardless of the time or sample complexity of the learning algorithms for $\mathcal{C}$.

Since the bound for Theorem 15.1 holds even for algorithms with unbounded computational resources, the best we can hope for distinct $\mathcal{C}$ is $E_M(\mathcal{C}) = \Theta(\epsilon)$. Monomials, $k$-DNF and symmetric functions are all distinct concept classes.

We will now consider the hardness results for algorithms that learn from only positive or only negative examples. First, we need a definition.

A concept class $\mathcal{C}$ is *positive $t$-spittable* if there exist concepts $c_1, \ldots, c_t \in \mathcal{C}$ and points $u_1, \ldots, u_t, v \in X$ such that $c_j$ includes $v$ and all the $u_i$'s except $u_j$. In other words, $\forall i, j$ $u_i \in c_j$ if $i \neq j$ and $\forall i$ $v \in c_i$.

114
As an example, monomials on $n$ variables are positive $n$-splittable. Let $c_1 = x_1, \ldots, c_n = x_n$ and let $u_1 = 011\ldots11, u_2 = 101\ldots11, \ldots, u_n = 111\ldots10, v = 111\ldots11$. Also, if $C$ is a concept class with \text{VCD}(C) = d$, then $C$ is both positive and negative $d$-splittable.

**Theorem 15.2** For positive $t$-splittable $C$, any algorithm calling on $\text{POS}_d^\theta$ can tolerate an error rate $\beta \leq \frac{\epsilon}{t-1}$ regardless of time or sample complexity.

**Proof:** The theorem is proved using the method of induced distributions. Since $C$ is positive $t$-splittable, let $c_1, \ldots, c_t \in C$ and $u_1, \ldots, u_t, v \in X$ be as in the definition of positive $t$-splittable. Let $c_j$ be the target concept. Consider the following actual distribution:

$$
D_j^+(v) = 1 - \epsilon \\
D_j^+(u_i) = \frac{\epsilon}{t-1} \quad \forall i \neq j \\
D_j^+(u_j) = 1
$$

In case of noise, the adversary chooses $u_j$. Thus, the induced distribution is

$$
I_j^+(v) = (1 - \beta)(1 - \epsilon) \\
I_j^+(u_i) = (1 - \beta)\left(\frac{\epsilon}{t-1}\right) \quad \forall i \neq j \\
I_j^+(u_j) = \beta
$$

If $\beta = (1 - \beta)\left(\frac{\epsilon}{t-1}\right)$, then the induced distributions $I_j^+$ are identical for all $u_i$. Solving for $\beta$, we get

$$
\beta = \frac{\epsilon}{1 + \frac{\epsilon}{t-1}} = \frac{\epsilon}{t - 1 + \epsilon} \leq \frac{\epsilon}{t - 1}
$$

Thus we see that the learning algorithm can only tolerate an error rate of $\beta < \frac{\epsilon}{t-1}$. ■

The above theorem leads to the following corollaries.

**Corollary 15.1** A learning algorithm for monomials (using only positive or only negative examples) can tolerate an error rate of $\beta < \frac{\epsilon}{n-1}$ where $n$ is the number of variables in the monomial.

**Corollary 15.2** A learning algorithm for $k$-DNF (using only positive or only negative examples) can tolerate an error rate of $\beta < \frac{c_0 \epsilon}{n^k}$ where $c_0$ is a constant $> 0$.

**Corollary 15.3** A learning algorithm for symmetric functions (using only positive or only negative examples) can tolerate an error rate of $\beta < \frac{\epsilon}{n-1}$ where $n$ is the number of variables in the symmetric functions.
The above corollaries agree with some earlier work by Valiant [44]. Valiant proved the following two theorems.

**Theorem 15.3** Monomials are learnable using calls to $POS^β_M$ for $β < \frac{αc}{n}$ for constant $c_0$.

The learning algorithm for the above is slightly different from the algorithm for learning monomials in a noise-free environment. In this case, variable $x_i$ is deleted from the current hypothesis only if $x_i = 0$ in at least $\frac{αc}{n}$ of the examples seen so far.

**Theorem 15.4** $k$-DNF is learnable using calls to $NEG^β_M$ for $β < \frac{αc}{n^k}$ for constant $c_0$.

## 15.4 Generalization of Occam’s Razor

In the previous section, we saw the limitations on the error rate that a learning algorithm can tolerate, even if it had unbounded computational resources. In this section, we will concentrate on the largest error rate that a polynomial time learning algorithm can tolerate. We will give a technique for building robust algorithms which is a generalization of Occam’s Razor.

Let $A$ be an algorithm accessing $POS^β_M$ and $NEG^β_M$, and taking inputs $ε > 0, δ < 1$. Suppose that for target concept $c ∈ C$ and $0 ≤ β < \frac{ε}{4}$, $A$ makes $m$ calls to $POS^β_M$ and receives examples $u_1, \ldots, u_m$, and $m$ calls to $NEG^β_M$ and receives examples $v_1, \ldots, v_m$, and outputs $h_A \in H$ satisfying with probability at least $1 − δ$:

\[
\begin{align*}
(i) \quad & \frac{\{u_i : u_i \not\in h_A\}}{m} \leq \frac{ε}{2} \\
(ii) \quad & \frac{\{v_i : v_i \in h_A\}}{m} \leq \frac{ε}{2}
\end{align*}
\]

Thus, with high probability, $h_A$ agrees with a fraction of at least $1 − \frac{ε}{2}$ of the sample. Such an algorithm $A$ will be called a $β$-robust Occam Algorithm for $C$.

**Theorem 15.5** Let $β < \frac{ε}{4}$, and let $A$ be a $β$-robust Occam Algorithm for $C$ outputting $h_A \in H$. Then $A$ is a PAC-learning algorithm for $C$ for $m = O \left( \frac{1}{ε^3} \ln \frac{1}{δ} + \frac{1}{ε} \ln |H| \right)$.

**Proof:** We will show that if condition $(i)$ holds for $A$, then $ε^+(h_A) < ε$, and similarly, of condition $(ii)$ holds, then $ε^-(h_A) < ε$.

Suppose $h \in H$ is $ε$-bad, i.e. $ε^+(h) ≥ ε$. Then, the probability that $h$ agrees with an example $x$ from $POS^β_M$ is at most

\[(1 − β)(1 − ε) + β ≤ 1 − \frac{3ε}{4}\]

for $β < \frac{ε}{4}$. Thus, the probability that $h$ agrees with at least a fraction $1 − \frac{ε}{2}$ of the $m$ positive examples is

\[LE \left( \frac{3ε}{4}, m, \frac{ε}{2} m \right) \leq e^{-mc/24}\]

116
using Chernoff bounds. Thus, the probability that some $\varepsilon$-bad $h$ agrees with a fraction $1 - \frac{\varepsilon}{2}$ of the $m$ positive examples is at most

$$|H|e^{-me/24}.$$ 

We want this to be at most $\frac{\delta}{2}$. Thus, by solving

$$|H|e^{-me/24} \leq \frac{\delta}{2}$$

for $m$, we get

$$m \geq \frac{24}{\varepsilon} \left( \ln |H| + \ln \frac{2}{\delta} \right)$$

On the other hand, if $c$ is the target concept, then the probability that $c$ does not agree with a fraction $1 - \frac{\varepsilon}{2}$ of the $m$ positive examples is

$$GE \left( \frac{\varepsilon}{4}, m, \frac{\varepsilon}{2} m \right) \leq \frac{\delta}{2}$$

for $\beta \leq \frac{\varepsilon}{4}$ and $m$ from above. Observe that $m = O \left( \frac{1}{\varepsilon} \ln \frac{1}{\delta} + \frac{1}{\varepsilon} \ln |H| \right)$. Also, for such an $m$, any hypothesis agreeing with a fraction $1 - \frac{\varepsilon}{2}$ of a sample of size $m$ from $POS^\beta_M$ must have $e^+(h) < \varepsilon$ with high probability.

The above theorem can be used to prove the correctness of the following learning algorithms by Valiant [44].

**Theorem 15.6** There is a polynomial time learning algorithm $A$ for monomials which can learn from positive examples and tolerate an error rate of $\Theta \left( \frac{\varepsilon}{n} \right)$ where $n$ is the number of variables in the monomial.

**Theorem 15.7** There is a polynomial time learning algorithm $A$ for $k$-DNF which can learn from negative examples and tolerate an error rate of $\Theta \left( \frac{\varepsilon}{n^k} \right)$ where $n$ is the number of variables in the $k$-DNF formula.

### 15.5 Using Positive and Negative Examples to Improve Learning Algorithms

In this section, we show that by using both positive and negative examples, we can improve the noise rate that can be handled by the learning algorithm.

**Theorem 15.8** Let $C$ be a polynomial time learnable concept class in the error-free model by algorithm $A$ with sample complexity $s_A(\varepsilon, \delta, n)$ and let $s = s_A(\frac{\delta}{8}, \frac{1}{2}, n)$. Then, we can learn $C$ in polynomial time with an error rate of $\beta = \Omega \left( \min \left( \frac{\varepsilon}{8}, \frac{\ln s}{s} \right) \right)$. 

117
Proof: Assume \( \beta < \frac{\ln s}{s} \). Run algorithm \( A \) using oracles \( POS^R_M \) and \( NEG^R_M \) with accuracy set to \( \frac{\xi}{s} \) and confidence set to \( \frac{1}{2} \). The probability that no errors occur during this run of \( A \) is

\[
(1 - \beta)^s \geq \left(1 - \frac{\ln s}{s}\right)^s = \left(1 - \frac{\ln s}{s}\right)^{\frac{s}{\ln s}} \ln s
\]

Recall that

\[ e^x = O(1 + x) \]

Putting \( x = -\frac{\ln s}{s} \), we get:

\[ e^{-\frac{\ln s}{s}} = O\left(1 - \frac{\ln s}{s}\right) \]

which implies

\[ e^{-1} = \left(e^{-\frac{\ln s}{s}}\right)^{\frac{s}{\ln s}} = O\left(\left(1 - \frac{\ln s}{s}\right)^{\frac{s}{\ln s}}\right) \]

Thus, we get

\[
(1 - \beta)^s \geq O\left(e^{-\ln s}\right) = O\left(\frac{1}{s}\right)
\]

Let a successful run be one that has no errors. Using Chernoff bounds, it can be shown that for \( r = O\left(s \ln \frac{1}{\delta}\right) \), the probability of no successful runs in \( r \) tries is at most \( \delta \).

Thus, by running algorithm \( A \) \( r \) times, we get hypotheses \( h_1, \ldots, h_r \), one of which is \( \frac{\xi}{s} \)-good. By doing our standard hypothesis testing, we can find this hypothesis. Note that if a hypothesis is \( \frac{\xi}{s} \)-good, then regardless of what the adversary does, we get an error rate of at most \( (1 - \beta)\frac{\xi}{s} + \beta \leq \frac{\xi}{s} \) for the range of \( \beta \) we are assuming. Thus, we can use the standard hypothesis testing.

Earlier, we saw that polynomial time algorithms for learning monomials using only positive examples can tolerate an error rate of \( \Theta\left(\frac{\xi}{n}\right) \). Applying the above theorem improves this bound by a log factor as can be seen in the following corollary.

Corollary 15.4 There is a polynomial time learning algorithm \( A \) for monomials which can learn from both positive and negative examples and tolerate an error rate of \( \Omega\left(\frac{\xi}{n} \ln \frac{n}{\epsilon}\right) \) where \( n \) is the number of variables in the monomial.

Also, in an earlier topic, we saw that any learning algorithm \( A \) for a concept class \( C \) must have sample complexity

\[
s_A(\epsilon, \delta) = \Omega\left(\frac{1}{\epsilon} \ln \frac{1}{\delta} + \frac{d'}{\epsilon}\right)
\]
where \( d = \text{vcd}(\mathcal{C}) \). Applying Theorem 15.8, we get an algorithm for \( \mathcal{C} \) that can tolerate a malicious error rate of \( c_0 \left( \ln \frac{\delta}{\varepsilon} \right)^{\frac{1}{\delta}} \) for a constant \( c_0 \). Note that Corollary 15.4 follows from this more general result since the VC dimension of monomials is \( \Theta(n) \).

### 15.6 Relationship between Learning Monomials with Errors and Set Cover

In this section, we will generalize the Set Cover problem to Partial Cover and show its relationship to learning monomials with errors. We first define the partial cover problem.

**Problem:** PC (Partial Cover)

**Instance:** Finite sets \( S_1, \ldots, S_n \), (without loss of generality, assume \( \bigcup_{i=1}^{n} S_i = \{1, \ldots, m\} \)), positive real costs \( c_1, \ldots, c_n \), and a positive fraction \( 0 < p \leq 1 \).

**Output:** \( J' \subseteq \{1, \ldots, n\} \) such that \( |\bigcup_{j \in J'} S_j| \geq pm \) and \( \text{PCost}(J') = \sum_{j \in J'} c_j \) is minimized.

The Partial Cover Problem is clearly NP-complete, since it contains set cover as a special case \((p = 1)\). We now give the greedy approximation algorithm for Partial Cover.

Let \( J = \{1, \ldots, n\} \) and \( T = \{1, \ldots, m\} \).

**Algorithm \( G \) for the Partial Cover Problem:**

1. Set \( J^* = 0, m = |\bigcup_{j \in J^*} S_j| \).

2. Let \( q = pm - |\bigcup_{j \in J^*} S_j| \). \( \forall j \notin J^* \), if \( |S_j| > q \), delete any \( |S_j| - q \) elements from \( S_j \).

3. If \( |\bigcup_{j \in J^*} S_j| \geq pm \) then stop: \( J^* \) is a partial cover. Otherwise find a \( k \) maximizing the ratio \( \frac{|S_k|}{c_k} \).

4. Add \( k \) to \( J^* \), and replace each \( S_j \) by \( S_j - S_k \). Return to Step 1.

It can be shown that \( G \) gives a cover which is within \( \Theta(\log m) \) of the optimal. The following theorem demonstrates that an approximation algorithm for a covering problem can be used to obtain a robust algorithm for monomials. See the Kearns and Li paper for both of these proofs.

**Theorem 15.9** There is a polynomial time learning algorithm \( A \) for monomials of length at most \( l \) which can tolerate an error rate of \( \frac{c_0}{l} \left( \frac{1}{\log \frac{1}{\varepsilon} n} \right) \) where \( n \) is the number of variables in the monomial and \( c_0 \) is a positive constant.

Now we shall show that by approaching the best known upper bound of \( \varepsilon \) on \( \text{E}_M(M_n) \), where \( M_n \) is the class of monomials over \( n \) variables, we are actually improving the best known approximation algorithms for set cover.
Theorem 15.10 Suppose $A$ is a polynomial time learning algorithm for learning monomials using monomials, and can tolerate a malicious error rate of $\beta = \frac{e}{r(n)}$, then there is a polynomial time algorithm $A'$ for set cover that finds a weighted cover with cost at most $4r(n)$ times the optimal cost, where $n$ is the number of sets.

Proof: We will give a reduction showing how algorithm $A$ can be used as a subroutine to obtain the desired set cover approximation algorithm by associating the weighted set covers with monomials. Let $J_{\text{opt}} \in \{1, \ldots, n\}$ be an optimal cover of $T = \{1, \ldots, m\}$, and let $PCost(J_{\text{opt}}) = C_{PC_{\text{opt}}}$. Each cover $\{i_1, \ldots, i_j\}$ is associated with monomial $x_{i_1} \wedge \cdots \wedge x_{i_j}$. Let $M_{\text{opt}}$ be the monomial corresponding to the optimal cover $J_{\text{opt}}$. The basic idea is for $A'$ to run $A$ with $M_{\text{opt}}$ as the target concept. We then force the output of algorithm $A$ to be a monomial with error close to $M_{\text{opt}}$. This monomial will correspond to a cover that is close to optimal.

We will use $NEG_M^\beta$ to force the output of $A$ to correspond to a cover. To achieve this goal we put a uniform distribution over the following set of $m$ negative examples. For each $i \in T$, define $\overline{e}_i$ to be an $n$-bit vector whose $j$th bit is 0 if and only if $i \in S_j$. For example, let $m = 5$ and $S_1 = \{1, 3, 5\}$, $S_2 = \{1, 2\}$, $S_3 = \{1, 4, 5\}$, and $S_4 = \{1, 3\}$. Then $\overline{e}_1 = 0000$, $\overline{e}_2 = 1011$, $\overline{e}_3 = 0110$, $\overline{e}_4 = 1101$, and $\overline{e}_5 = 0101$. Let $E = \cup_{i \in T} \overline{e}_i$. Observe that $J' = \{i_1, \ldots, i_j\}$ is a cover of $T$ if and only if every vector in $E$ is a negative example of the corresponding monomial. Thus since $J_{\text{opt}}$ is a cover, $M_{\text{opt}}$ is consistent with all the negative examples. Finally, we let $\epsilon < \frac{1}{m}$. Note that $\frac{1}{\epsilon}$ is polynomial in the size of the set cover instance. Since we force $A$ to output a monomial consistent with the sample, $A'$ outputs a cover. Note that at this point we have used the assumption that $A$ learns monomials by monomials because we do not have a natural mapping between non-monomials and possible covers.

Next we use $POS_M^\beta$ to force the cost of the cover to be low. First, without loss of generality, we assume the costs are scaled so that $\sum_{i=1}^n c_i \leq \frac{e}{r(n)} < 1$. Next we define the induced distribution $I^+$ on the positive examples. We will then show that this distribution can be obtained. For each $1 \leq i \leq n$, let $\overline{p}_i$ be the $n$-bit vector that is 1 everywhere except in the $i$th position. Now $I^+$ places a probability of $c_i$ on all $\overline{p}_i$, and probability $1 - \sum_{i=1}^n c_i$ on $\overline{I}^+$. The probability that a monomial $x_{i_1} \wedge \cdots \wedge x_{i_j}$ disagrees with a positive example is $\sum_{i=1}^n c_i = \text{COST}(J)$ where $J = \{i_1, \ldots, i_j\}$ is the corresponding cover.

Thus the smallest possible rate of disagreements with $I^+$ is $C_{PC_{\text{opt}}}$ and is achieved by $M_{\text{opt}}$. Since $C_{PC_{\text{opt}}} \leq \sum_{i=1}^n c_i \leq \frac{e}{r(n)} = \beta$, this distribution can be induced by a malicious oracle.

A good hypothesis output by algorithm $A$ must have a rate of disagreement of at most $2\epsilon$ with $M_{\text{opt}}$ on $I^+$. One factor of $\epsilon$ comes from the disagreement with $D^+$, and the other factor comes from the error $\beta < \epsilon$ induced by the malicious oracle. Finally, we do not know the actual value of $C_{PC_{\text{opt}}}$. So, we repeatedly divide $\epsilon$ by 2 and run $A$ with the new value of $\epsilon$, but without changing $I^+$. This forces $A$ to output better and better monomials, until it reaches a point where $C_{PC_{\text{opt}}} < \frac{e}{r(n)} \leq 2C_{PC_{\text{opt}}}$. At that point, the hypothesis output will have error $< 2\epsilon < 4r(n)\epsilon$.

The only problem with this occurs when $C_{PC_{\text{opt}}}$ is extremely small causing exponential number of iterations of $A$. This can be avoided by first using the greedy approximation.
algorithm $G$ and discarding all the sets whose cost is higher than the greedy cover. This ensures that in the new (smaller) instance, all the costs are polynomially related (within a multiplicative $\log m$ factor), and thus we can maintain the polynomial running time. ■

If in the above theorem, $r(n) = o(\log n)$, then $A$ would imply a significant improvement in the existing approximation for set cover. As for our learning problem, the only other alternative (besides improving on the best known set covering result) is to use a hypothesis space larger than just monomials.