7

Reducibility in PAC Learning

From the positive results of Chapters 1 and 2 and the hardness results of Chapter 6, we now have examples, for natural and nontrivial concept classes, of both efficient PAC learning and inherent unpredictability. Although we have certainly identified some powerful methods for obtaining both kinds of results — for instance, the method of finding an Occam algorithm for a concept class in order to show that it is efficiently PAC learnable, and the method of showing that a concept class can compute iterated products in order to demonstrate its inherent unpredictability — we still lack a framework that allows us to compare the relative complexity of PAC learning concept classes whose actual status in the PAC model is uncertain.

In this chapter, we develop a notion of *reducibility* for learning in the PAC model. In order to choose a notion of reducibility that is meaningful we must first state our goals. Informally, we are of course interested in a notion of reducibility that preserves efficient PAC learnability. Thus if a concept class $\mathcal{C}$ "reduces" to a concept class $\mathcal{C}'$, and $\mathcal{C}'$ is efficiently PAC learnable, then it should follow that $\mathcal{C}$ is also efficiently PAC learnable.

There will be at least three uses for the reducibility we develop. First, if $\mathcal{C}$ reduces to $\mathcal{C}'$, and we already have an efficient learning algorithm for $\mathcal{C}'$, then the reduction immediately yields an efficient learning algorithm

for $\mathcal{C}$. Recall that it was exactly this method that provided an efficient algorithm for learning kCNF from our efficient algorithm for learning boolean conjunctions (1CNF) in Chapter 1. Second, we can give evidence for the intractability of learning a concept class $\mathcal{C}'$ by showing that another concept class $\mathcal{C}$, believed to be hard to learn, reduces to $\mathcal{C}'$. Third, if $\mathcal{C} \supset \mathcal{C}'$, but we do not know if either of $\mathcal{C}$ and $\mathcal{C}'$ is efficiently PAC learnable, a reduction of $\mathcal{C}$ to $\mathcal{C}'$ at least proves that learning the subclass $\mathcal{C}'$ is no easier than learning $\mathcal{C}$.

We begin with a motivating example falling in the final category. While the PAC learnability of general disjunctive normal form (DNF) formulae remains unresolved so far, we use a simple reduction to demonstrate that the monotone version of the problem is not easier than the unrestricted version.

7.1 Reducing DNF to Monotone DNF

We have informally discussed DNF formulae at many points throughout our studies. Formally, a general **disjunctive normal form (DNF) formula** over $\{0,1\}^n$ is an expression of the form $c = T_1 \vee T_2 \vee \cdots \vee T_m$, where each term T_i is a conjunction of literals over the boolean variables $x_1, \ldots, x_n$. Since each term can be represented using at most $O(n)$ bits, we define $size(c) = mn$. Because a learning algorithm is always allowed time polynomial in n, it is fair to think of the dependence on $size(c)$ as allowing the learning algorithm to also have time polynomial in the number of terms m.

If we let $\mathcal{C}_n$ be the class of all DNF formulae over $\{0,1\}^n$, note that $\mathcal{C}_n$ actually contains a representation of every possible boolean function over $\{0,1\}^n$; however, the PAC learning problem is nevertheless "fair" in principle, because we measure the complexity of a function by its DNF representation size. Thus the learning algorithm is provided with more computation for more complex target functions.

Recall that in Chapter 1 we studied the severely restricted subclass of DNF formulae in which the number of terms was bounded by a fixed constant k (thus, $size(c) = kn$). We called such a formula a k-term DNF formula, and we proved that PAC learning such formulae is hard if the hypothesis class used is also k-term DNF formulae, but can be done efficiently if k-CNF formulae is used as the hypothesis class. However, since this solution required time exponential in k, it is inapplicable to the general problem, where the number of terms is a parameter. On the other hand, the inherent unpredictability methods of Chapter 6 also seem inapplicable, since DNF formulae do not appear up to the task of efficiently computing iterated products. In short, the efficient PAC learnability of general DNF formulae remains one of the most important open problems in the PAC model. Our modest goal here is to use a reduction to dismiss one possible source for the apparent difficulty of this problem — namely, the fact that the target formulae are allowed to have both negated and unnegated variables.

A **monotone DNF formula** over $\{0,1\}^n$ is simply a disjunction $c' = T_1 \vee T_2 \vee \cdots \vee T_m$ in which each T_i is a conjunction over the boolean variables $x_1, \ldots, x_n$ (but *not* their negations). Thus, the difference between monotone DNF and general DNF is that we forbid negated variables in the monotone case. Obviously, unlike for general DNF, it is not the case that every boolean function over $\{0,1\}^n$ can be represented as a monotone DNF formula. Could it be the case that there is an efficient PAC learning algorithm for monotone DNF formulae, yet general DNF formulae are inherently unpredictable?

The answer is no. Suppose we had an efficient learning algorithm L' for PAC learning monotone DNF formulae using some polynomially evaluatable hypothesis class $\mathcal{H}'$. We now show that L' can actually be used as a subroutine in an efficient PAC learning algorithm L for general DNF formulae.

Let us consider a small example. Suppose that we have a general DNF formula over the variables $x_1, \ldots, x_6$, say $c = (x_1 \wedge \overline{x}_5 \wedge x_6) \vee (\overline{x}_1 \wedge x_2 \wedge x_4)$. By introducing "new" variables $y_1, \ldots, y_6$ but always assigning $y_i = \overline{x}_i$,

we may also write c as $(x_1 \wedge y_5 \wedge x_6) \vee (y_1 \wedge x_2 \wedge x_4)$. Now this is a monotone formula over the expanded variable set $x_1, \ldots, x_6, y_1, \ldots, y_6$; let us use c' to denote this monotone representation of c. Note that $size(c')$ is not too much larger than $size(c)$.

Suppose now we are given the positive example $\langle 010110, 1 \rangle$ of c over the original variables x_i. Then the *expanded instance* 010110 101001, which is the original instance followed by its bitwise complement, is a positive example of the monotone formula c' over the expanded variable set consisting of the x_i and the y_i. More generally, it is easy to verify that if $\langle a, c(a) \rangle$ is any example of c, then $\langle a \cdot comp(a), c(a) \rangle$ is always a correct example of c', where $comp(a)$ is the bitwise complement of a and $\cdot$ denotes string concatenation. It is crucial to note that this transformation of the instances $a \rightarrow a \cdot comp(a)$ is independent of the actual target formula c, and can be efficiently computed from a.

Now given access to the examples oracle $EX(c, \mathcal{D})$ for a target general DNF formula c over $\{0,1\}^n$, our algorithm L will simply simulate the algorithm L' for the monotone case. Each time L' requests a random example, L will take a random labeled example $\langle a, c(a) \rangle$ of c from $EX(c, \mathcal{D})$, and give the transformed example $\langle a \cdot comp(a), c(a) \rangle$ of length $2n$ to L'. Since the examples given to L' are perfectly consistent with the monotone formula c', algorithm L' will then produce some polynomially evaluatable boolean function h' over the $2n$ variables $x_1, \ldots, x_n, y_1, \ldots, y_n$ that is accurate with respect to the distribution $\mathcal{D}'$ induced on the transformed examples by the simulation. Note that $\mathcal{D}'$ may be quite different from $\mathcal{D}$. For instance, if $\mathcal{D}$ was the uniform distribution on $\{0,1\}^n$, $\mathcal{D}'$ will not be the uniform distribution on $\{0,1\}^{2n}$, but the uniform distribution on pairs $a \cdot a' \in \{0,1\}^{2n}$ where $a, a' \in \{0,1\}^n$ and $a' = comp(a)$.

The hypothesis h of L will then be given by $h(a) = h'(a \cdot comp(a))$. It is easy to see that $error_{\mathcal{D}}(h) = error_{\mathcal{D}'}(h')$, because $h(a) \neq c(a)$ if and only if $h'(a \cdot comp(a)) \neq c'(a \cdot comp(a))$, and a has exactly the same weight under $\mathcal{D}$ that $a \cdot comp(a)$ has under $\mathcal{D}'$. Also, since L' runs in time polynomial in $size(c')$, and we have already pointed out that $size(c')$ is not much larger than $size(c)$, L runs in time polynomial in $size(c)$, and

also in time polynomial in $2n$.

We have shown:

Theorem 7.1 *If the representation class of general DNF formulae is efficiently PAC learnable, then the representation class of monotone DNF formulae is efficiently PAC learnable.*

7.2 A General Method for Reducibility

We have just given a simple example of a reduction of one learning problem to another. We now give a general definition of this notion.

Definition 16 *We say that the concept class C over instance space X* **PAC-reduces** *to the concept class C' over instance space X' if the following conditions are met:*

- *(Efficient Instance Transformation) There exists a mapping $G : X \to X'$ and a polynomial $p(\cdot)$ such that for every n and every $x \in X_n$, $G(x) \in X'_{p(n)}$, and G is computable in polynomial time. Thus, G maps instances in X of length n to instances in X' of length $p(n)$, and can be efficiently computed.*

- *(Existence of Image Concept) There exists a polynomial $q(\cdot)$ such that for every concept $c \in C_n$, there is a concept $c' \in C'_{p(n)}$ with the property that $size(c') \leq q(size(c))$, and for all $x \in X_n$, $c(x) = 1$ if and only if $c'(G(x)) = 1$. Thus, for any concept $c \in C$ there is a concept $c' \in C'$ that is not much larger than c, and whose behavior on the transformed instances exactly mirrors that of c on the original instances.*

Note that while we insist the instance transformation be efficiently computable, there is no such demand on the mapping from c to c'; we only ask for its existence. Thus, it may be intractable (or even impossible) to compute the representation of c' from the representation of c.

Under this formalization, our reduction of DNF to monotone DNF was $G(a) = a{\cdot}comp(a)$ (thus, instances of length n were mapped to instances of length $2n$), and c' was just the monotone DNF formula obtained by replacing each occurrence of $\overline{x}_i$ in c with the variable y_i.

The basic property we require of our reducibility is established by the following theorem.

Theorem 7.2 *Let $\mathcal{C}$ and $\mathcal{C}'$ be concept classes. Then if $\mathcal{C}$ PAC-reduces to $\mathcal{C}'$, and $\mathcal{C}'$ is efficiently PAC learnable, $\mathcal{C}$ is efficiently PAC learnable.*

Proof: Given a learning algorithm L' for $\mathcal{C}'$, we use L' to learn $\mathcal{C}$ in the obvious way: given a random example $\langle x, c(x)\rangle$ of an unknown target concept $c \in \mathcal{C}$, we compute the labeled example $\langle G(x), c(x)\rangle$ and give it to L'. If the instances $x \in X$ are drawn according to $\mathcal{D}$, then the instances $G(x) \in X'$ are drawn according to some induced distribution $\mathcal{D}'$. Although we do not know the target concept c, our definition of reduction guarantees that the computed examples $\langle G(x), c(x)\rangle$ are consistent with some $c' \in \mathcal{C}'$, and thus L' will output a hypothesis h' in time polynomial in $size(c')$ (and thus polynomial in $size(c)$) that has error at most ϵ with respect to $\mathcal{D}'$. Our hypothesis for c becomes $h(x) = h'(G(x))$, which is easily seen to have at most ϵ error with respect to $\mathcal{D}$. □(Theorem 7.2)

Another useful way of stating Theorem 7.2 is to say that if $\mathcal{C}$ PAC-reduces to $\mathcal{C}'$, and $\mathcal{C}$ is inherently unpredictable then $\mathcal{C}'$ is inherently unpredictable. It is this view of our reducibility we use in the next section.

7.3 Reducing Boolean Formulae to Finite Automata

In this section, we derive the main result of this chapter, which is that the class of boolean formulae PAC-reduces to the class of deterministic finite automata. We shall show this in two parts. First, we show that the class of log-space Turing machines PAC-reduces to finite automata. Then we show that the class of boolean formulae PAC-reduces to log-space Turing machines. The main result then follows from the transitivity of our reducibility, which is established in Exercise 7.1.

There is a minor technicality involved with defining concept classes represented by finite automata and Turing machines, because we normally think of these devices as accepting strings of a possibly infinite number of different lengths, while we have been thinking of a concept as being defined only over instances of some fixed length n. For the purposes of this chapter, however, it will suffice to define our concept classes by restricting our attention to the behavior of a finite automaton or Turing machine on inputs of a single common length.

Thus, consider the concept class $\mathcal{C}$ in which there is a constant k (we will implicitly choose k as large as necessary in our analysis) such that every concept $c \in \mathcal{C}_n$ over $\{0,1\}^n$ can be evaluated by a Turing machine T_c that uses only $k \log n$ work space (thus, we assume that T_c has a read-only input tape and a separate read/write work tape). Thus, for every $c \in \mathcal{C}_n$ and every $a \in \{0,1\}^n$, $T_c(a) = c(a)$. We call this the representation class of **log-space Turing machines**, and we define $size(c)$ to be the number of states in the finite control of T_c. Similarly, let $\mathcal{C}'$ be the concept class in which each $c' \in \mathcal{C}'_n$ over $\{0,1\}^n$ can be evaluated by a deterministic finite automata $M_{c'}$; thus for any $a \in \{0,1\}^n$, $c'(a) = 1$ if and only if $M_{c'}$ accepts a. We call this the representation class of **deterministic finite automata**, and we define $size(c')$ to be the number of states in $M_{c'}$. We now show that $\mathcal{C}$ PAC-reduces to $\mathcal{C}'$.

Theorem 7.3 *The class of log-space Turing machines PAC-reduces to the class of deterministic finite automata.*

Proof: We describe for each $k \log n$-space Turing machine $T = T_c$ (computing some concept c in $\mathcal{C}_n$) a small DFA $M = M_c$ that will simulate T on appropriately transformed instances $G(a)$. Intuitively (and we will flesh out the details momentarily), M must overcome two handicaps in order to simulate T. The first is that T has logarithmic work space but a DFA has no explicit memory. This is easily compensated for by encoding all the $2^{k \log n} = n^k$ possible work tape contents of T in the state diagram of M, and can be done using only n^k additional size overhead in M. The second handicap is that T can move its read-only input head either right or left on the input tape, while a DFA must proceed forward (to the right) through its input at every transition. This can be overcome with the help of the instance transformation G. For any input $a \in \{0,1\}^n$ to T, $G(a)$ will simply replicate a many times: thus $G(a) = aa \cdots a$. If at any point T moves left on the input a, then M will simply move $n-1$ symbols forward on $G(a)$, arriving in the next copy of a but one symbol to the left of its position in the former copy, thus affecting a move to the left. This requires a $\log n$ bit counter, which can also be encoded in the state diagram of M using only polynomially many states. The number of copies $p(n)$ of a that must be given in $G(a)$ is clearly bounded by the running time of T (since this bounds the number of possible input head moves by T), which is polynomial.

To see this is more detail, consider constructing a directed graph G_T based on the description of T. Each node of G_T is labeled by a tuple (s, σ, i) where s is a state of the finite control of T, σ is a binary string of length $k \log n$, which we interpret as the work tape contents of T, and $1 \leq i \leq n$ is interpreted as an index indicating the head position of T on the input tape. Then we draw a directed edge, labeled by the bit $b \in \{0,1\}$, from the node (s, σ, i) to the node $(s', \sigma', i+1)$ if and only if T, when in state s with work tape contents σ and input head position i, on reading a b from the input would move the input head right and go to state s' with work tape contents σ'. (Note that this can only happen

if σ and σ' differ only in the single bit at the head position.) We will additionally label this directed edge by an R to indicate a move to the right on the input. Similarly, we will label an edge from a node with input head index i to a node with input head index $i-1$ by an L.

Now G_T is "almost" a DFA simulating T, if we allow the traversal of a transition labeled R or L to move the input head of G_T right or left, respectively. But it is easy to see that we can replace each R transition by a finite automata that simply reads through the next $n+1$ input bits of $G(a)$, and each L transition by a one that reads through the next $n-1$ bits of $G(a)$. The resulting graph is exactly a finite automata whose behavior on $G(a)$ is the same as T on a. Note that the size of this automata is polynomial in n and polynomial in the number of states in the finite control of T. □(Theorem 7.3)

We now reduce boolean formulae to log-space Turing machines to complete the sequence of reductions.

Theorem 7.4 *The class of boolean formulae PAC-reduces to the class of log-space Turing machines.*

Proof: We show that for any boolean formula f over $\{0,1\}^n$, there is a log-space Turing machine T_f, with a number of finite control states that is polynomial in $size(f)$, that on input a computes $f(a)$ (thus, the instance transformation $G(a)$ is simply the identity transformation). We will actually prove the stronger result that there is a single log-space Turing machine T that takes as input a boolean formula f and an assignment a, and computes $f(a)$; the desired machine T_f can be obtained by fixing the formula input of T to be f. (Thus, T is universal for the class of boolean formulae.)

Recall that a boolean formula can be thought of as a circuit whose underlying graph is a tree (see Figure 6.1 in Chapter 6 and the accompanying text). Let us label each node in this tree with a unique natural

number that we call the *name* of the node. Assume without loss of generality that the formula f that is input to T is encoded as a list of items representing the tree circuit for f. Each item consists of a label indicating the name of a node in the binary tree for computing f (this label requires at most $O(\log size(f))$ bits, where $size(f)$ is the number of gates in the tree for f), a couple of bits indicating the gate type ($\wedge, \vee$ or $\neg$), and the labels of the left and right children of this gate. Now to compute $f(a)$, T conducts a depth-first search of the tree using the item list. To keep track of the search, T only needs to store the label of the current gate, and a few bits indicating the current "direction" of the search (that is, whether we arrived at the current gate g from the parent of g, the left child of g, or the right child of g). We also only ever need to store a single bit v indicating the value of the computation so far. For instance, if we are currently at an $\vee$ gate that we arrived at from the left child, and the value of the subfunction computed by the subtree rooted at the left child was $v = 1$, then there is no need to explore the right child of this gate; we can simply continue back up the tree and maintain the value $v = 1$. On the other hand, if $v = 0$ then we must explore the right subtree but we can overwrite the value of v, since the left subtree evaluated to 0 and thus cannot make the current $\vee$ evaluate to 1. The value of v returned from the right subtree will become the value for the current $\vee$ node. Similarly, if the current gate is an $\wedge$ gate and we returned from the left child with $v = 0$, we can simply continue up the tree with this value, bypassing the right subtree. Otherwise, we explore the right subtree and overwrite v.

□(Theorem 7.4)

From Theorems 7.3 and 7.4 and the transitivity of our reducibility (see Exercise 7.1), we immediately obtain:

Corollary 7.5 *The class of boolean formulae PAC-reduces to the class of deterministic finite automata.*

Thus, drawing on the results of Chapter 6, we obtain:

Theorem 7.6 *Under the discrete cube root assumption, the representation class of deterministic finite automata is inherently unpredictable.*

In light of this negative result, in the next chapter we will investigate a natural model of learning that provides the learner more power than in the PAC model, and obtain an efficient learning algorithm for finite automata.

7.4 Exercises

7.1. Prove that our reducibility for PAC learning is transitive. Thus, for any concept classes $\mathcal{C}_1, \mathcal{C}_2$ and $\mathcal{C}_2$, if $\mathcal{C}_1$ PAC-reduces to $\mathcal{C}_2$ and $\mathcal{C}_2$ PAC-reduces to $\mathcal{C}_3$, then $\mathcal{C}_1$ PAC-reduces to $\mathcal{C}_3$.

7.2. The concept class of **halfspaces** in $\Re^n$ is defined as follows: each concept is defined by a vector $\vec{u} \in \Re^n$ of unit length. An input $\vec{x} \in \Re^n$ is a positive example of $\vec{u}$ if and only if $\vec{u} \cdot \vec{x} \equiv \sum_{i=1}^{n} u_i \cdot x_i \geq 0$. In the the concept class of **exclusive-or of two halfspaces**, each concept is defined by a pair $(\vec{u}, \vec{v})$ of unit vectors in $\Re^n$. An input $\vec{x} \in \Re^n$ is a positive example of $(\vec{u}, \vec{v})$ if either $\vec{u} \cdot \vec{x} \geq 0$ and $\vec{v} \cdot \vec{x} < 0$, or $\vec{u} \cdot \vec{x} < 0$ and $\vec{v} \cdot \vec{x} \geq 0$; otherwise, $\vec{x}$ is a negative example.

Show that the class of exclusive-or of halfspaces PAC-reduces to the the class of halfspaces.

7.3. A **read-once DNF formulae** over $\{0,1\}^n$ is a disjunction $c = T_1 \vee T_2 \vee \cdots \vee T_m$ (where each T_i is a conjunction of literals over the boolean variables $x_1, \ldots, x_n$) in which every variable is restricted to appear at most once (whether negated or unnegated). Show that the representation class of general DNF formulae in which each formula over $\{0,1\}^n$ has at most $p(n)$ terms, for some fixed polynomial $p(\cdot)$, PAC-reduces to the representation class of read-once DNF formulae.

7.5 Bibliographic Notes

The general definition of PAC-reducibility was developed by Pitt and Warmuth [73], who also give the reduction of boolean formulae to finite automata and many other interesting reductions. The reduction of general DNF to monotone DNF is due to Kearns et al. [58]. Such reductions are now a standard tool of computational learning theory; the paper of Long and Warmuth [68] gives examples for geometric concept classes. Exercise 7.2 is due to M. Warmuth and L.G. Valiant.