1

The Probably Approximately Correct Learning Model

1.1 A Rectangle Learning Game

Consider a simple one-player learning game. The object of the game is to learn an unknown axis-aligned rectangle R — that is, a rectangle in the Euclidean plane $\Re^2$ whose sides are parallel with the coordinate axes. We shall call R the **target** rectangle. The player receives information about R only through the following process: every so often, a random point p is chosen in the plane according to some fixed probability distribution $\mathcal{D}$. The player is given the point p together with a label indicating whether p is contained in R (a positive example) or not contained in R (a negative example). Figure 1.1 shows the unknown rectangular region R along with a sample of positive and negative examples.

The goal of the player is to use as few examples as possible, and as little computation as possible, to pick a **hypothesis** rectangle R' which is a close approximation to R. Informally, the player's knowledge of R is tested by picking a new point at random from the same probability distribution $\mathcal{D}$, and checking whether the player can correctly decide whether the point falls inside or outside of R. Formally, we measure the

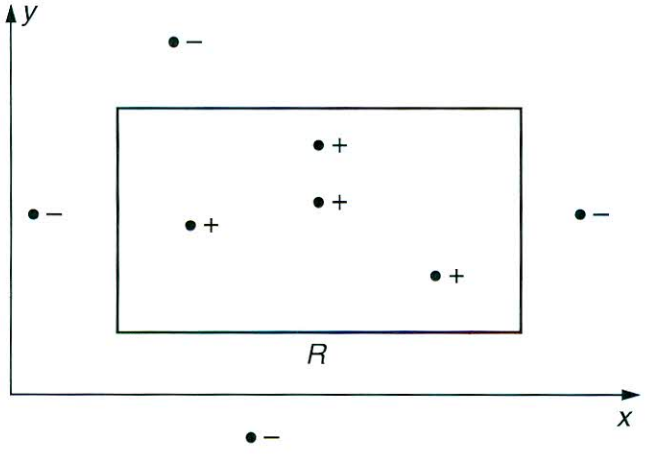


Figure 1.1: *The target rectangle R in the plane along with a sample of positive and negative examples.*

error of R' as the probability that a randomly chosen point from $\mathcal{D}$ falls in the region $R \Delta R'$, where $R \Delta R' = (R - R') \cup (R' - R)$.

To motivate the rectangle learning game, consider a slightly more concrete scenario that can be expressed as an instance of the game. Suppose that we wanted to learn the concept of "men of medium build". Assume that a man is of medium build if his height and weight both lie in some prescribed ranges — for instance, if his height is between five feet six inches and six feet, and his weight is between 150 pounds and 200 pounds. Then each man's build can be represented by a point in the Euclidean plane, and the concept of medium build is represented by an axis-aligned rectangular region of the plane. Thus, during an initial training phase, the learner is told for each new man he meets whether that man is of medium build or not. Over this period, the learner must form some model or hypothesis of the concept of medium build.

Now assume that the learner encounters every man in his city with

equal probability. Even under this assumption, the corresponding points in the plane may not be uniformly distributed (since not all heights and weights are equally likely, and height and weight may be highly dependent quantities), but will instead obey some fixed distribution $\mathcal{D}$ which may be quite difficult to characterize. For this reason, in our learning game, we allow the distribution $\mathcal{D}$ to be arbitrary, but we assume that it is fixed, and that each example is drawn independently from this distribution. (Note that once we allow $\mathcal{D}$ to be arbitrary, we no longer need to assume that the learner encounters every man in his city with equal probability.) To evaluate the hypothesis of the learner, we are simply evaluating its success in classifying the build of men in future encounters, still assuming that men are encountered according to the same probability distribution as during the training phase.

There is a simple and efficient strategy for the player of the rectangle learning game. The strategy is to request a "sufficiently large" number m of random examples, then choose as the hypothesis the axis-aligned rectangle R' which gives the tightest fit to the positive examples (that is, that rectangle with the smallest area that includes all of the positive examples and none of the negative examples). If no positive examples are drawn, then $R' = \emptyset$. Figure 1.2 shows the tightest-fit rectangle defined by the sample shown in Figure 1.1.

We will now show that for any target rectangle R and any distribution $\mathcal{D}$, and for any small values ϵ and δ $(0 < \epsilon, \delta \leq 1/2)$, for a suitably chosen value of the sample size m we can assert that with probability at least $1 - \delta$, the tightest-fit rectangle has error at most ϵ with respect to R and $\mathcal{D}$.

First observe that the tightest-fit rectangle R' is always contained in the target rectangle R (that is, $R' \subseteq R$ and so $R \Delta R' = R - R'$). We can express the difference $R - R'$ as the union of four rectangular strips. For instance, the topmost of these strips, which is shaded and denoted T' in Figure 1.3, is the region above the upper boundary of R' extended to the left and right, but below the upper boundary of R. Note that there is some overlap between these four rectangular strips at the corners. Now

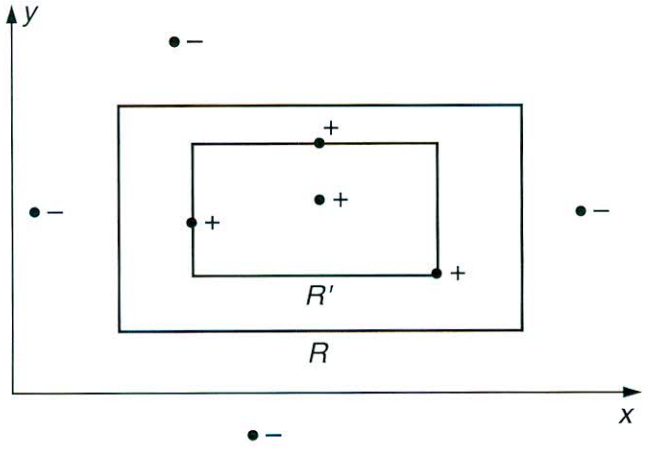


Figure 1.2: *The tightest-fit rectangle R' defined by the sample.*

if we can guarantee that the weight under $\mathcal{D}$ of each strip (that is, the probability with respect to $\mathcal{D}$ of falling in the strip) is at most $\epsilon/4$, then we can conclude that the error of R' is at most $4(\epsilon/4) = \epsilon$. (Here we have erred on the side of pessimism by counting each overlap region twice.)

Let us analyze the weight of the top strip T'. Define T to be the rectangular strip along the inside top of R which encloses *exactly* weight $\epsilon/4$ under $\mathcal{D}$ (thus, we sweep the top edge of R downwards until we have swept out weight $\epsilon/4$; see Figure 1.3). Clearly, T' has weight exceeding $\epsilon/4$ under $\mathcal{D}$ if and only if T' includes T (which it does not in Figure 1.3). Furthermore, T' includes T if and only if no point in T appears in the sample S — since if S does contain a point $p \in T$, this point has a positive label since it is contained in R, and then by definition of the tightest fit, the hypothesis rectangle R' must extend upwards into T to cover p.

By the definition of T, the probability that a single draw from the distribution $\mathcal{D}$ misses the region T is exactly $1 - \epsilon/4$. Therefore the

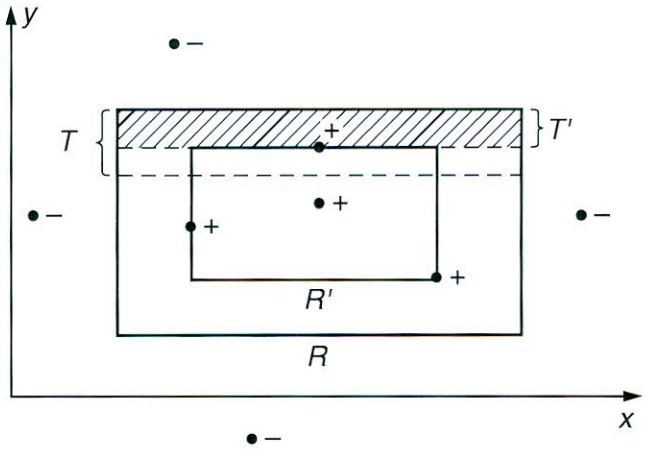


Figure 1.3: *Analysis of the error contributed by the top shaded strip T'. The strip T has weight exactly $\epsilon/4$ under $\mathcal{D}$.*

probability that m independent draws from $\mathcal{D}$ all miss the region T is exactly $(1 - \epsilon/4)^m$. Here we are using the fact that the probability of a conjunction of independent events is simply the product of the probabilities of the individual events. The same analysis holds for the other three rectangular regions of $R - R'$, so by the **union bound**, the probability that any of the four strips of $R - R'$ has weight greater than $\epsilon/4$ is at most $4(1 - \epsilon/4)^m$. By the union bound, we mean the fact that if A and B are any two events (that is, subsets of a probability space), then

$$\mathbf{Pr}[A \cup B] \leq \mathbf{Pr}[A] + \mathbf{Pr}[B].$$

Thus, the probability that one of the four error strips has weight exceeding $\epsilon/4$ is at most four times the probability that a fixed error strip has weight exceeding $\epsilon/4$.

Provided that we choose m to satisfy $4(1 - \epsilon/4)^m \leq \delta$, then with probability $1 - \delta$ over the m random examples, the weight of the error

region $R - R'$ will be bounded by ϵ, as claimed. Using the inequality

$$(1 - x) \leq e^{-x}$$

(which we shall appeal to frequently in our studies) we see that any value of m satisfying $4e^{-\epsilon m/4} \leq \delta$ also satisfies the previous condition. Dividing by 4 and taking natural logarithms of both sides gives $-\epsilon m/4 \leq \ln(\delta/4)$, or equivalently $m \geq (4/\epsilon)\ln(4/\delta)$.

In summary, provided our tightest-fit algorithm takes a sample of at least $(4/\epsilon)\ln(4/\delta)$ examples to form its hypothesis rectangle R', we can assert that with probability at least $1 - \delta$, R' will misclassify a new point (drawn according to the same distribution from which the sample was chosen) with probability at most ϵ.

A few brief comments are appropriate. First, note that the analysis really does hold for any fixed probability distribution. We only needed the independence of successive points to obtain our bound. Second, the sample size bound behaves as we might expect, in that as we increase our demands on the hypothesis rectangle — that is, as we ask for greater **accuracy** by decreasing ϵ or greater **confidence** by decreasing δ — our algorithm requires more examples to meet those demands. Finally, the algorithm we have analyzed is efficient: the required sample size is a slowly growing function of $1/\epsilon$ and $1/\delta$ (linear and logarithmic, respectively), and once the sample is given, the computation of the tightest-fit hypothesis can be carried out rapidly.

1.2 A General Model

In this section, we introduce the model of learning that will be the central object for most of our study: the **Probably Approximately Correct** or **PAC** model of learning. There are a number of features of the rectangle learning game and its solution that are essential to the PAC model, and bear highlighting before we dive into the general definitions.

- The goal of the learning game is to learn an unknown target set, but the target set is not arbitrary. Instead, there is a known and rather strong constraint on the target set — it is a rectangle in the plane whose sides are parallel to the axes.

- Learning occurs in a probabilistic setting. Examples of the target rectangle are drawn randomly in the plane according to a fixed probability distribution which is unknown and unconstrained.

- The hypothesis of the learner is evaluated relative to the *same* probabilistic setting in which the training takes place, and we allow hypotheses that are only approximations to the target. The tightest-fit strategy might not find the target rectangle exactly, but will find one with only a small probability of disagreement with the target.

- We are interested in a solution that is efficient: not many examples are required to obtain small error with high confidence, and we can process those examples rapidly.

We wish to state a general model of learning from examples that shares and formalizes the properties we have listed. We begin by developing and motivating the necessary definitions.

1.2.1 Definition of the PAC Model

Let X be a set called the **instance space**. We think of X as being a set of encodings of instances or objects in the learner's world. In our rectangle game, the instance space X was simply the set of all points in the Euclidean plane $\Re^2$. As another example, in a character recognition application, the instance space might consist of all 2-dimensional arrays of binary pixels of a given width and height.

A **concept** over X is just a subset $c \subseteq X$ of the instance space. In the rectangle game, the concepts were axis-aligned rectangular regions.

Continuing our character recognition example, a natural concept might be the set of all pixel arrays that are representations of the letter "A" (assuming that every pixel array either represents an "A", or fails to represent an "A").

A concept can thus be thought of as the set of all instances that positively exemplify some simple or interesting rule. We can equivalently define a concept to be a boolean mapping $c : X \to \{0,1\}$, with $c(x) = 1$ indicating that x is a positive example of c and $c(x) = 0$ indicating that x is a negative example. For this reason, we also sometimes call X the **input space**.

A **concept class** $\mathcal{C}$ over X is a collection of concepts over X. In the rectangle game, the target rectangle was chosen from the class $\mathcal{C}$ of all axis-aligned rectangles. Ideally, we are interested in concept classes that are sufficiently expressive for fairly general knowledge representation. As an example in a logic-based setting, suppose we have a set $x_1, \ldots, x_n$ of n boolean variables, and let X be the set of all assignments to these variables (that is, $X = \{0,1\}^n$). Suppose we consider concepts c over $\{0,1\}^n$ whose positive examples are exactly the satisfying assignments of some boolean formulae f_c over $x_1, \ldots, x_n$. Then we might define an interesting concept class $\mathcal{C}$ by considering only those boolean formulae f_c that meet some natural syntactic constraints, such as being in disjunctive normal form (DNF) and having a small number of terms.

In our model, a learning algorithm will have access to positive and negative examples of an unknown **target concept** c, chosen from a known concept class $\mathcal{C}$. The learning algorithm will be judged by its ability to identify a hypothesis concept that can accurately classify instances as positive or negative examples of c. Before specifying the learning protocol further, it is important to note that in our model, learning algorithms "know" the target class $\mathcal{C}$, in the sense that the designer of the learning algorithm is guaranteed that the target concept will be chosen from $\mathcal{C}$ (but must design the algorithm to work for any $c \in \mathcal{C}$).

Let $\mathcal{D}$ be any fixed probability distribution over the instance space X.

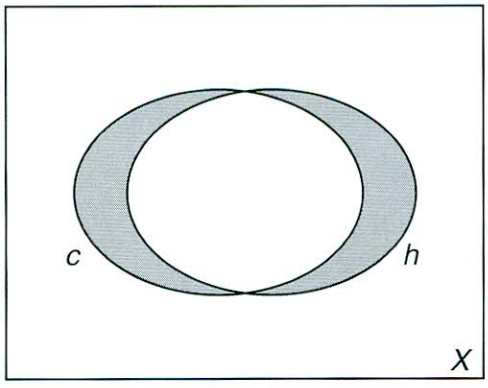


Figure 1.4: *Venn diagram of two concepts, with symmetric difference shaded.*

We will refer to $\mathcal{D}$ as the **target distribution**. If h is any concept over X, then the distribution $\mathcal{D}$ provides a natural measure of **error** between h and the target concept c: namely, we define

$$error(h) = \mathbf{Pr}_{x \in \mathcal{D}}[c(x) \neq h(x)].$$

Here we regard the concepts c and h as boolean functions, and we have introduced a notational convention that we shall use frequently: the subscript $x \in \mathcal{D}$ to $\mathbf{Pr}[\cdot]$ indicates that the probability is taken with respect to the random draw of x according to $\mathcal{D}$. Note that $error(h)$ has an implicit dependence on c and $\mathcal{D}$ that we will usually omit for brevity when no confusion will result.

A useful alternative way to view $error(h)$ is represented in Figure 1.4. Here we view the concepts c and h as sets rather than as functions, and we have drawn an abstract Venn diagram showing the positive examples of c and h, which of course lie within the entire instance space X. Then $error(h)$ is simply the probability with respect to $\mathcal{D}$ that an instance is drawn falling in the shaded region.

Let $EX(c, \mathcal{D})$ be a procedure (we will sometimes call it an *oracle*) that

runs in unit time, and on each call returns a labeled example $\langle x, c(x)\rangle$, where x is drawn randomly and independently according to $\mathcal{D}$. A learning algorithm will have access to this oracle when learning the target concept $c \in \mathcal{C}$. Ideally, the learning algorithm will satisfy three properties:

- The number of calls to $EX(c, \mathcal{D})$ is small, in the sense that it is bounded by a fixed polynomial in some parameters to be specified shortly.
- The amount of computation performed is small.
- The algorithm outputs a **hypothesis concept** h such that $error(h)$ is small.

Note that the number of calls made by a learning algorithm to $EX(c, \mathcal{D})$ is bounded by the running time of the learning algorithm.

We are now ready to give the definition of Probably Approximately Correct learning. We designate it as our preliminary definition, since we shall soon make some important additions to it.

Definition 1 *(The PAC Model, Preliminary Definition) Let $\mathcal{C}$ be a concept class over X. We say that $\mathcal{C}$ is* **PAC learnable** *if there exists an algorithm L with the following property: for every concept $c \in \mathcal{C}$, for every distribution $\mathcal{D}$ on X, and for all $0 < \epsilon < 1/2$ and $0 < \delta < 1/2$, if L is given access to $EX(c, \mathcal{D})$ and inputs ϵ and δ, then with probability at least $1 - \delta$, L outputs a hypothesis concept $h \in \mathcal{C}$ satisfying $error(h) \leq \epsilon$. This probability is taken over the random examples drawn by calls to $EX(c, \mathcal{D})$, and any internal randomization of L.*

If L runs in time polynomial in $1/\epsilon$ and $1/\delta$, we say that $\mathcal{C}$ is **efficiently** *PAC learnable. We will sometimes refer to the input ϵ as the* **error parameter**, *and the input δ as the* **confidence parameter**.

The hypothesis $h \in C$ of the PAC learning algorithm is thus "approximately correct" with high probability, hence the name Probably Approximately Correct learning.

Two important comments regarding the PAC learning model are now in order. First, the error and confidence parameters ϵ and δ control the two types of failure to which a learning algorithm in the PAC model is inevitably susceptible. The error parameter ϵ is necessary since, for example, there may be only a negligible probability that a small random sample will distinguish between two competing hypotheses that differ on only one improbable point in the instance space. The confidence parameter δ is necessary since the learning algorithm may occasionally be extremely unlucky, and draw a terribly "unrepresentative" sample of the target concept — for instance, a sample consisting only of repeated draws of the same instance despite the fact that the distribution is spread evenly over all instances. The best we can hope for is that the probability of both types of failure can be made arbitrarily small at a modest cost.

Second, notice that we demand that a PAC learning algorithm perform well with respect to any distribution $\mathcal{D}$. This strong requirement is moderated by the fact that we only evaluate the hypothesis of the learning algorithm with respect to the same distribution $\mathcal{D}$. For example, in the rectangle learning game discussed earlier, this means that if the distribution gives negligible weight to some parts of the Euclidean plane, then the learner does not have to be very careful in learning the boundary of the target rectangle in that region.

Definition 1, then, is our tentative definition of PAC learning, which will be the model forming the bulk of our studies. As previously mentioned, we shall make a couple of important refinements to this definition before we begin the serious investigation. Before doing so, however, we pause to note that we have already proven our first result in this model. Recall that our algorithm for the rectangle learning game required the ability to store real numbers and perform basic operations on them, such as comparisons. In the following theorem, and throughout our study, whenever necessary we will assume a model of computation that allows

storage of a single real number in a single memory location, and that charges one unit of computation time for a basic arithmetic operation (addition, multiplication or division) on two real numbers.

Theorem 1.1 *The concept class of axis-aligned rectangles over the Euclidean plane $\Re^2$ is efficiently PAC learnable.*

1.2.2 Representation Size and Instance Dimension

An important issue was swept under the rug in our definition of PAC learning. This is the fundamental distinction between a concept (which is just a set or a boolean function) and its representation (which is a symbolic encoding of that set or function). Consider a class of concepts defined by the satisfying assignments of boolean formulae. A concept from this class — that is, the set of satisfying assignments for some boolean formula f — can be represented by the formula f, by a truth table, or by another boolean formula f' that is logically equivalent to f. Although all of these are representations of the same underlying concept, they may differ radically in representational size.

For instance, it is not hard to prove that for all n, the boolean parity function $f(x_1, \ldots, x_n) = x_1 \oplus \cdots \oplus x_n$ (where $\oplus$ denotes the exclusive-or operation) can be computed by a circuit of $\wedge$, $\vee$ and $\neg$ gates whose size is bounded by a fixed polynomial in n, but to represent this same function as a disjunctive normal form (abbreviated DNF) formula requires size exponential in n. As another example, in high-dimensional Euclidean space $\Re^n$, we may choose to represent a convex polytope either by specifying its vertices, or by specifying linear equations for its faces, and these two representation schemes can differ exponentially in size.

In each of these examples, we are fixing some representation scheme — that is, a precise method for encoding concepts — and then examining

the size of the encoding for various concepts. Other natural representation schemes that the reader may be familiar with include decision trees and neural networks. As with boolean formulae, in these representation schemes there is an obvious mapping from the representation (a decision tree or a neural network) to the set or boolean function that is being represented. There is also a natural measure of the size of a given representation in the scheme (for instance, the number of nodes in the decision tree or the number of weights in a neural network).

Since a PAC learning algorithm only sees examples of the functional (that is, input-output) behavior of the target concept, it has absolutely no information about which, if any, of the many possible representations is actually being used to represent the target concept in reality. However, it matters greatly which representation the algorithm chooses for its hypothesis, since the time to write this representation down is obviously a lower bound on the running time of the algorithm.

Formally speaking, a **representation scheme** for a concept class $\mathcal{C}$ is a function $\mathcal{R} : \Sigma^* \to \mathcal{C}$, where Σ is a finite alphabet of symbols. (In cases where we need to use real numbers to represent concepts, such as axis-aligned rectangles, we allow $\mathcal{R} : (\Sigma \cup \Re)^* \to \mathcal{C}$.) We call any string $\sigma \in \Sigma^*$ such that $\mathcal{R}(\sigma) = c$ a **representation** of c (under $\mathcal{R}$). Note that there may be many representations of a concept c under the representation scheme $\mathcal{R}$.

To capture the notion of representation size, we assume that associated with $\mathcal{R}$ there is a mapping $size : \Sigma^* \to N$ that assigns a natural number $size(h)$ to each representation $h \in \Sigma^*$. Note that we allow $size(\cdot)$ to be any such mapping; results obtained under a particular definition for $size(\cdot)$ will be meaningful only if this definition is natural. Perhaps the most realistic setting, however, is that in which $\Sigma = \{0, 1\}$ (thus, we have a binary encoding of concepts) and we define $size(h)$ to be the length of h in bits. (For representations using real numbers, it is often natural to charge one unit of size for each real number.) Although we will use other definitions of size when binary representations are inconvenient, our definition of $size(\cdot)$ will always be within a polynomial factor

of the binary string length definition. For example, we can define the size of a decision tree to be the number of nodes in the tree, which is always within a polynomial factor of the length of the binary string needed to encode the tree in any reasonable encoding method.

So far our notion of size is applicable only to representations (that is, to strings $h \in \Sigma^*$). We would like to extend this definition to measure the size of a target concept $c \in \mathcal{C}$. Since the learning algorithm has access only to the input-output behavior of c, in the worst case it must assume that the simplest possible mechanism is generating this behavior. Thus, we define $size(c)$ to be $size(c) = \min_{\mathcal{R}(\sigma)=c}\{size(\sigma)\}$. In other words, $size(c)$ is the size of the smallest representation of the concept c in the underlying representation scheme $\mathcal{R}$. Intuitively, the larger $size(c)$ is, the more "complex" the concept c is with respect to the chosen representation scheme. Thus it is natural to modify our notion of learning to allow more computation time for learning more complex concepts, and we shall do this shortly.

For a concept class $\mathcal{C}$, we shall refer to the **representation class** $\mathcal{C}$ to indicate that we have in mind some fixed representation scheme $\mathcal{R}$ for $\mathcal{C}$. In fact, we will usually *define* the concept classes we study by their representation scheme. For instance, we will shortly examine the concept class in which each concept is the set of satisfying assignments of some conjunction of boolean variables. Thus, each concept can be represented by a list of the variables in the associated conjunction.

It is often convenient to also introduce some notion of size or dimension for the elements of the instance space. For example, if the instance space X_n is the n-dimensional Euclidean space $\Re^n$, then each example is specified by n real numbers, and so it is natural to say that the size of the examples is n. The same comments apply to the instance space $X_n = \{0,1\}^n$. It turns out that these are the only two instance spaces that we will ever need to consider in our studies, and in the spirit of asymptotic analysis we will want to regard the instance space dimension n as a parameter of the learning problem (for example, to allow us to study the problem of learning axis-aligned rectangles in $\Re^n$ in time poly-

nomial in n). Now if we let $\mathcal{C}_n$ be the class of concepts over X_n, and write $X = \cup_{n\geq 1} X_n$ and $\mathcal{C} = \cup_{n\geq 1}\mathcal{C}_n$, then X and $\mathcal{C}$ define an infinite family of learning problems of increasing dimension.

To incorporate the notions of target concept size and instance space dimension into our model, we make the following refined definition of PAC learning:

Definition 2 *(The PAC Model, Modified Definition) Let $\mathcal{C}_n$ be a representation class over X_n (where X_n is either $\{0,1\}^n$ or n-dimensional Euclidean space $\Re^n$), and let $X = \cup_{n\geq 1} X_n$ and $\mathcal{C} = \cup_{n\geq 1}\mathcal{C}_n$. The modified definition of PAC learning is the same as the preliminary definition (Definition 1), except that now we allow the learning algorithm time polynomial in n and $size(c)$ (as well as $1/\epsilon$ and $1/\delta$ as before) when learning a target concept $c \in \mathcal{C}_n$.*

Since in our studies X_n will always be either $\{0,1\}^n$ or n-dimensional Euclidean space, the value n is implicit in the instances returned by $EX(c,\mathcal{D})$. We assume that the learner is provided with the value $size(c)$ as an input. (However, see Exercise 1.5.)

We emphasize that while the target concept may have many possible representations in the chosen scheme, we only allow the learning algorithm time polynomial in the size of the smallest such representation. This provides a worst-case guarantee over the possible representations of c, and is consistent with the fact that the learning algorithm has no idea which representation is being used for c, having only functional information about c.

Finally, we note that for several concept classes the natural definition of $size(c)$ is already bounded by a polynomial in n, and thus we really seek an algorithm running in time polynomial in just n. For instance, if we look at the representation class of all DNF formulae with at most 3 terms, any such formula has length at most $3n$, so polynomial dependence

on the size of the target formula is the same as polynomial dependence on n.

1.3 Learning Boolean Conjunctions

We now give our second result in the PAC model, showing that **conjunctions of boolean literals** are efficiently PAC learnable. Here the instance space is $X_n = \{0,1\}^n$. Each $a \in X_n$ is interpreted as an assignment to the n boolean variables $x_1, \ldots, x_n$, and we use the notation a_i to indicate the ith bit of a. Let the representation class $\mathcal{C}_n$ be the class of all conjunctions of literals over $x_1, \ldots, x_n$ (a **literal** is either a variable x_i or its negation $\overline{x}_i$). Thus the conjunction $x_1 \wedge \overline{x}_3 \wedge x_4$ represents the set $\{a \in \{0,1\}^n : a_1 = 1, a_3 = 0, a_4 = 1\}$. It is natural to define the size of a conjunction to be the number of literals in that conjunction. Then clearly $size(c) \leq 2n$ for any conjunction $c \in \mathcal{C}_n$. (We also note that a standard binary encoding of any conjunction $c \in \mathcal{C}_n$ has length $O(n \log n)$.) Thus for this problem, we seek an algorithm that runs in time polynomial in n, $1/\epsilon$ and $1/\delta$.

Theorem 1.2 *The representation class of conjunctions of boolean literals is efficiently PAC learnable.*

Proof: The algorithm we propose begins with the hypothesis conjunction

$$h = x_1 \wedge \overline{x}_1 \wedge \cdots \wedge x_n \wedge \overline{x}_n.$$

Note that initially h has no satisfying assignments. The algorithm simply ignores any negative examples returned by $EX(c, \mathcal{D})$. Let $\langle a, 1 \rangle$ be a positive example returned by $EX(c, \mathcal{D})$. In response to such a positive example, our algorithm updates h as follows: for each i, if $a_i = 0$, we delete x_i from h, and if $a_i = 1$, we delete $\overline{x}_i$ from h. Thus, our algorithm deletes any literal that "contradicts" the positive data.

For the analysis, note that the set of literals appearing in h at any time always contains the set of literals appearing in the target concept c. This is because we begin with h containing all literals, and a literal is only deleted from h when it is set to 0 in a positive example; such a literal clearly cannot appear in c. The fact that the literals of h always include those of c implies that h will never err on a negative example of c (that is, h is more specific than c).

Thus, consider a literal z that occurs in h but not in c. Then z causes h to err only on those positive examples of c in which $z = 0$; also note that it is exactly such positive examples that would have caused our algorithm to delete z from h. Let $p(z)$ denote the total probability of such instances under the distribution $\mathcal{D}$, that is,

$$p(z) = \mathbf{Pr}_{a \in \mathcal{D}}[c(a) = 1 \wedge z \text{ is } 0 \text{ in } a].$$

Since every error of h can be "blamed" on at least one literal z of h, by the union bound we have $error(h) \leq \sum_{z \in h} p(z)$. We say that a literal is *bad* if $p(z) \geq \epsilon/2n$. If h contains no bad literals, then $error(h) \leq \sum_{z \in h} p(z) \leq 2n(\epsilon/2n) = \epsilon$. We now upper bound the probability that a bad literal will appear in h.

For any *fixed* bad literal z, the probability that this literal is not deleted from h after m calls of our algorithm to $EX(c, \mathcal{D})$ is at most $(1 - \epsilon/2n)^m$, because the probability the literal z is deleted by a single call to $EX(c, \mathcal{D})$ is $p(z)$ (which is at least $\epsilon/2n$ for a bad literal). From this we may conclude that the probability that there is *some* bad literal that is not deleted from h after m calls is at most $2n(1 - \epsilon/2n)^m$, where we have used the union bound over the $2n$ possible literals.

Thus to complete our analysis we simply need to solve for the value of m satisfying $2n(1 - \epsilon/2n)^m \leq \delta$, where $1 - \delta$ is the desired confidence. Using the inequality $1 - x \leq e^{-x}$, it suffices to pick m such that $2ne^{-m\epsilon/2n} \leq \delta$, which yields $m \geq (2n/\epsilon)(\ln(2n) + \ln(1/\delta))$.

Thus, if our algorithm takes at least this number of examples, then with probability at least $1 - \delta$ the resulting conjunction h will have error

at most ϵ with respect to c and $\mathcal{D}$. Since the algorithm takes linear time to process each example, the running time is bounded by mn, and hence is bounded by a polynomial in n, $1/\epsilon$ and $1/\delta$, as required. □(Theorem 1.2)

1.4 Intractability of Learning 3-Term DNF Formulae

We next show that a slight generalization of the representation class of boolean conjunctions results in an intractable PAC learning problem. More precisely, we show that the class of disjunctions of three boolean conjunctions (known as **3-term disjunctive normal form (DNF) formulae**) is not efficiently PAC learnable unless every problem in NP can be efficiently solved in a worst-case sense by a randomized algorithm — that is, unless for every language A in NP there is a randomized algorithm taking as input any string α and a parameter $\delta \in [0,1]$, and that with probability at least $1-\delta$ correctly determines whether $\alpha \in A$ in time polynomial in the length of α and $1/\delta$. The probability here is taken only with respect to the coin flips of the randomized algorithm. In technical language, our hardness result for 3-term DNF is based on the widely believed assumption that $RP \neq NP$.

The representation class $\mathcal{C}_n$ of 3-term DNF formulae is the set of all disjunctions $T_1 \vee T_2 \vee T_3$, where each T_i is a conjunction of literals over the boolean variables $x_1, \ldots, x_n$. We define the size of such a representation to be sum of the number of literals appearing in each term (which is always bounded by a fixed polynomial in the length of the bit string needed to represent the 3-term DNF in a standard encoding). Then $size(c) \leq 6n$ for any concept $c \in \mathcal{C}_n$ because there are at most $2n$ literals in each of the three terms. Thus, an efficient learning algorithm for this problem is required to run in time polynomial in n, $1/\epsilon$ and $1/\delta$.

Theorem 1.3 *If $RP \neq NP$, the representation class of 3-term DNF*

formulae is not efficiently PAC learnable.

Proof: The high-level idea of the proof is to reduce an *NP*-complete language A (to be specified shortly) to the problem of PAC learning 3-term DNF formulae. More precisely, the reduction will efficiently map any string α, for which we wish to determine membership in A, to a set S_α of labeled examples. The cardinality $|S_\alpha|$ will be bounded by a polynomial in the string length $|\alpha|$. We will show that given a PAC learning algorithm L for 3-term DNF formulae, we can run L on S_α, in a manner to be described, to determine (with high probability) if α belongs to A or not.

The key property we desire of the mapping of α to S_α is that $\alpha \in A$ if and only if S_α is **consistent** with some concept $c \in \mathcal{C}$. The notion of a concept being consistent with a sample will recur frequently in our studies.

Definition 3 *Let $S = \{\langle x_1, b_1\rangle, \ldots, \langle x_m, b_m\rangle\}$ be any labeled set of instances, where each $x_i \in X$ and each $b_i \in \{0,1\}$. Let c be a concept over X. Then we say that c is* **consistent** *with S (or equivalently, S is consistent with c) if for all $1 \leq i \leq m$, $c(x_i) = b_i$.*

Before detailing our choice for the *NP*-complete language A and the mapping of α to S_α, just suppose for now that we have managed to arrange things so that $\alpha \in A$ if and only if S_α is consistent with some concept in $\mathcal{C}$. We now show how a PAC learning algorithm L for $\mathcal{C}$ can be used to *determine* if there exists a concept in $\mathcal{C}$ that is consistent with S_α (and thus whether $\alpha \in A$) with high probability. This is achieved by the following general method: we set the error parameter $\epsilon = 1/(2|S_\alpha|)$ (where $|S_\alpha|$ denotes the number of labeled pairs in S_α), and answer each request of L for a random labeled example by choosing a pair $\langle x_i, b_i\rangle$ uniformly at random from S_α. Note that if there is a concept $c \in \mathcal{C}$ consistent with S_α, then this simulation emulates the oracle $EX(c, \mathcal{D})$, where $\mathcal{D}$ is uniform over the (multiset of) instances appearing in S_α. In

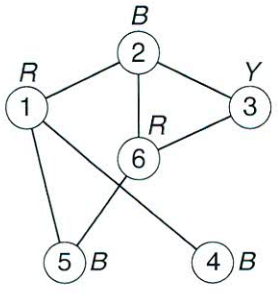
this case, by our choice of ϵ, we have guaranteed that any hypothesis h with error less that ϵ must in fact be consistent with S_α, for if h errs on even a single example in S_α, its error with respect to c and $\mathcal{D}$ is at least $1/|S_\alpha| = 2\epsilon > \epsilon$. On the other hand, if there is no concept in $\mathcal{C}$ consistent with S_α, L cannot possibly find one. Thus we can simply check the output of L for consistency with S_α to determine with confidence $1 - \delta$ if there exists a consistent concept in $\mathcal{C}$.

Combined with the assumed mapping of a string α to a set S_α, we thus can determine (with probability at least $1 - \delta$) the membership of α in A by simulating the PAC learning algorithm on S_α. This general method of using a PAC learning algorithm to determine the existence of a concept that is consistent with a labeled sample is quite common in the computational learning theory literature, and the main effort comes in choosing the right *NP*-complete language A, and finding the desired mapping from instances α of A to sets of labeled examples S_α, which we now undertake.

To demonstrate the intractability of learning 3-term DNF formulae, the *NP*-complete language A that we shall use is Graph 3-Coloring:

The Graph 3-Coloring Problem. Given as input an undirected graph $G = (V, E)$ with vertex set $V = \{1, \ldots, n\}$ and edge set $E \subseteq V \times V$, determine if there is an assignment of a color to each element of V such that at most 3 different colors are used, and for every edge $(i, j) \in E$, vertex i and vertex j are assigned different colors.

We now describe the desired mapping from an instance $G = (V, E)$ of Graph 3-Coloring to a set S_G of labeled examples. S_G will consist of a set S_G^+ of positively labeled examples and a set S_G^- of negatively labeled examples, so $S_G = S_G^+ \cup S_G^-$. For each $1 \leq i \leq n$, S_G^+ will contain the labeled example $\langle v(i), 1 \rangle$, where $v(i) \in \{0, 1\}^n$ is the vector with a 0 in the ith position and 1's everywhere else. These examples intuitively encode the vertices of G. For each edge $(i, j) \in E$, the set S_G^- will contain the labeled example $\langle e(i, j), 0 \rangle$, where $e(i, j) \in \{0, 1\}^n$ is the vector with 0's in the ith and jth positions, and 1's everywhere else. Figure 1.5 shows



S_G^+	S_G^-
<011111,1>	<001111,0>
<101111,1>	<011011,0>
<110111,1>	<011101,0>
<111011,1>	<100111,0>
<111101,1>	<101110,0>
<111110,1>	<110110,0>
	<111100,0>

$T_R = x_2 \wedge x_3 \wedge x_4 \wedge x_5$

$T_B = x_1 \wedge x_3 \wedge x_6$

$T_Y = x_1 \wedge x_2 \wedge x_4 \wedge x_5 \wedge x_6$

Figure 1.5: *A graph G with a legal 3-coloring, the associated sample, and the terms defined by the coloring.*

an example of a graph G along with the resulting sets S_G^+ and S_G^-. The figure also shows a legal 3-coloring of G, with R, B and Y denoting red, blue and yellow.

We now argue that G is 3-colorable if and only if S_G is consistent with some 3-term DNF formula. First, suppose G is 3-colorable and fix a 3-coloring of G. Let R be the set of all vertices colored red, and let T_R be the conjunction of all variables in $x_1, \ldots, x_n$ whose index does *not* appear in R (see Figure 1.5). Then for each $i \in R$, $v(i)$ must satisfy T_R because the variable x_i does not appear in T_R. Furthermore, no $e(i,j) \in S_G^-$ can satisfy T_R because since both i and j cannot be colored red, one of x_i and x_j must appear in T_R. We can define terms that are satisfied by the non-blue and non-yellow $v(i)$ in a similar fashion, with no negative examples being accepted by any term.

For the other direction, suppose that the formula $T_R \vee T_B \vee T_Y$ is consistent with S_G. Define a coloring of G as follows: the color of vertex i is red if $v(i)$ satisfies T_R, blue if $v(i)$ satisfies T_B, and yellow if $v(i)$ satisfies T_Y (we break ties arbitrarily if $v(i)$ satisfies more than one term). Since

the formula is consistent with S_G, every $v(i)$ must satisfy some term, and so every vertex must be assigned a color by this process. We now argue that it is a legal 3-coloring. To see this, note that if i and j $(i \neq j)$ are assigned the same color (say red), then both $v(i)$ and $v(j)$ satisfy T_R. Since the ith bit of $v(i)$ is 0 and the ith bit of $v(j)$ is 1, it follows that neither x_i nor $\overline{x}_i$ can appear in T_R. Since $v(j)$ and $e(i,j)$ differ only in their ith bits, if $v(j)$ satisfies T_R then so does $e(i,j)$, implying $e(i,j) \notin S_G^-$ and hence $(i,j) \notin E$. □(Theorem 1.3)

Thus, we see that 3-term DNF formulae are not efficiently PAC learnable under the assumption that *NP*-complete problems cannot be solved with high probability by a probabilistic polynomial-time algorithm (technically, under the assumption $RP \neq NP$). With some more elaborate technical gymnastics, the same statement can in fact be made for 2-term DNF formulae, and for k-term DNF formulae for any constant $k \geq 2$.

However, note that our reduction relied critically on our demand in the definition of PAC learning that the learning algorithm output a hypothesis from the same representation class from which the target formula is drawn — we used each term of the hypothesis 3-term formula to define a color class in the graph. In the next section we shall see that this demand is in fact necessary for this intractability result, since its removal permits an efficient learning algorithm for this same class. This will motivate our final modification of the definition of PAC learning.

1.5 Using 3-CNF Formulae to Avoid Intractability

We conclude this chapter by showing that if we allow the learning algorithm to output a more expressive hypothesis representation, then the class of 3-term DNF formulae *is* efficiently PAC learnable. In combination with Theorem 1.3, this motivates our final modification to the definition of PAC learning.

We can use the fact that for boolean algebra, $\vee$ distributes over $\wedge$ (that is, $(u \wedge v) \vee (w \wedge x) = (u \vee w) \wedge (u \vee x) \wedge (v \vee w) \wedge (v \vee x)$ for boolean variables u, v, w, x) to represent any 3-term DNF formula over $x_1, \ldots, x_n$ by an equivalent conjunctive normal form (CNF) formulae over $x_1, \ldots, x_n$ in which each clause contains at most 3 literals (we will call such formulae **3-CNF formulae**):

$$T_1 \vee T_2 \vee T_3 \equiv \bigwedge_{u \in T_1, v \in T_2, w \in T_3} (u \vee v \vee w).$$

Here the conjunction is over all clauses choosing one literal from each term.

We can *reduce* the problem of PAC learning 3-CNF formulae to the problem of PAC learning conjunctions, for which we already have an efficient algorithm. The high-level idea is as follows: given an oracle for random examples of an unknown 3-CNF formula, there is a simple and efficient method by which we can transform each positive or negative example into a corresponding positive or negative example of an unknown conjunction (over a larger set of variables). We then give the transformed examples to the learning algorithm for conjunctions that we have already described in Section 1.3. The hypothesis output by the learning algorithm for conjunctions can then be transformed into a good hypothesis for the unknown 3-CNF formula.

To describe the desired transformation of examples, we regard a 3-CNF formula as a conjunction over a new and larger variable set. For every triple of literals u, v, w over the original variable set $x_1, \ldots, x_n$, the new variable set contains a variable $y_{u,v,w}$ whose value is defined by $y_{u,v,w} = u \vee v \vee w$. Note that when $u = v = w$, then $y_{u,v,w} = u$, so all of the original variables are present in the new set. Also, note that the number of new variables $y_{u,v,w}$ is $(2n)^3 = O(n^3)$.

Thus for any assignment $a \in \{0,1\}^n$ to the original variables $x_1, \ldots, x_n$, we can in time $O(n^3)$ compute the corresponding assignment a' to the new variables $\{y_{u,v,w}\}$. Furthermore, it should be clear that any 3-CNF formula c over $x_1, \ldots, x_n$ is equivalent to a simple conjunction c' over the

new variables (just replace any clause $(u \vee v \vee w)$ by an occurrence of the new variable $y_{u,v,w}$). Thus, we can run our algorithm for conjunctions from Section 1.3, expanding each assignment to $x_1, \ldots, x_n$ that is a positive example of the unknown 3-CNF formula into an assignment for the $y_{u,v,w}$, and giving this expanded assignment to the algorithm as a positive example of an unknown conjunction over the $y_{u,v,w}$. We can then convert the resulting hypothesis conjunction h' over the $y_{u,v,w}$ back to a 3-CNF h in the obvious way, by expanding an occurrence of the variable $y_{u,v,w}$ to the clause $(u \vee v \vee w)$.

Formally, we must argue that if c and $\mathcal{D}$ are the target 3-CNF formula and distribution over $\{0,1\}^n$, and c' and $\mathcal{D}'$ are the corresponding conjunction over the $y_{u,v,w}$ and induced distribution over assignments a' to the $y_{u,v,w}$, then if h' has error less than ϵ with respect to c' and $\mathcal{D}'$, h has error less than ϵ with respect to c and $\mathcal{D}$. This is most easily seen by noting that our transformation of instances is one-to-one: if a_1 is mapped to a_1' and a_2 is mapped to a_2', then $a_1 \neq a_2$ implies $a_1' \neq a_2'$. Thus each vector a' on which h' differs from c' has a unique preimage a on which h differs from c, and the weight of a under $\mathcal{D}$ is exactly that of a' under $\mathcal{D}'$. It is worth noting, however, that our reduction is exploiting the fact that our conjunctions learning algorithm works for any distribution $\mathcal{D}$, as the distribution is "distorted" by the transformation. For example, even if $\mathcal{D}$ was the uniform distribution over $\{0,1\}^n$, $\mathcal{D}'$ would not be uniform over the transformed assignments a'.

We have just given an example of a **reduction** between two learning problems. A general notion of reducibility in PAC learning will be formalized and studied in Chapter 7.

We have proven:

Theorem 1.4 *The representation class of 3-CNF formulae is efficiently PAC learnable.*

Thus, because we have already shown that any 3-term DNF formula

can be written as a 3-CNF formula, we can PAC learn 3-term DNF formulae if we allow the hypothesis to be represented as a 3-CNF formula, but not if we insist that it be represented as a 3-term DNF formula! The same statement holds for any constant $k \geq 2$ for k-term DNF formulae and k-CNF formulae. This demonstrates an important principle that often appears in learning theory: even for a fixed concept class from which target concepts are chosen, the choice of hypothesis representation can sometimes mean the difference between efficient algorithms and intractability. The specific cause of intractability here is worth noting: the problem of just predicting the classification of new examples of a 3-term DNF formula is tractable (we can use a 3-CNF formula for this purpose), but expressing the prediction rule in a particular form (namely, 3-term DNF formulae) is hard.

This state of affairs motivates us to generalize our basic definition one more time, to allow the learning algorithm to use a more expressive hypothesis representation than is strictly required to represent the target concept. After all, we would not have wanted to close the book on the learnability of 3-term DNF formulae after our initial intractability result just because we were constrained by an artificial definition that insisted that learning algorithms use some particular hypothesis representation. Thus our final modification to the definition of PAC learning lets the hypothesis representation used be a parameter of the PAC learning problem.

Definition 4 *(The PAC Model, Final Definition) If C is a concept class over X and $\mathcal{H}$ is a representation class over X, we will say that C* **is (efficiently) PAC learnable using** *$\mathcal{H}$ if our basic definition of PAC learning (Definition 2) is met by an algorithm that is now allowed to output a hypothesis from $\mathcal{H}$. Here we are implicitly assuming that $\mathcal{H}$ is at least as expressive as C, and so there is a representation in $\mathcal{H}$ of every function in C. We will refer to $\mathcal{H}$ as the* **hypothesis class** *of the PAC learning algorithm.*

While for the reasons already discussed we do not want to place un-

necessary restrictions on $\mathcal{H}$, neither do we want to leave $\mathcal{H}$ entirely unconstrained. In particular, it would be senseless to study a model of learning in which the learning algorithm is constrained to run in polynomial time, but the hypotheses output by this learning algorithm could not even be evaluated in polynomial time. This motivates the following definition.

Definition 5 *We say that the representation class $\mathcal{H}$ is* **polynomially evaluatable** *if there is an algorithm that on input any instance $x \in X_n$ and any representation $h \in \mathcal{H}_n$, outputs the value $h(x)$ in time polynomial in n and size(h).*

Throughout our study, we will always be implicitly assuming that PAC learning algorithms use polynomially evaluatable hypothesis classes. Using our new language, our original definition was for PAC learning $\mathcal{C}$ using $\mathcal{C}$, and now we shall simply say that $\mathcal{C}$ is **efficiently PAC learnable** to mean that $\mathcal{C}$ is efficiently PAC learnable using $\mathcal{H}$ for some polynomially evaluatable hypothesis class $\mathcal{H}$.

The main results of this chapter are summarized in our new language by the following theorem.

Theorem 1.5 *The representation class of 1-term DNF formulae (conjunctions) is efficiently PAC learnable using 1-term DNF formulae. For any constant $k \geq 2$, the representation class of k-term DNF formulae is not efficiently PAC learnable using k-term DNF formulae (unless $RP = NP$), but is efficiently PAC learnable using k-CNF formulae.*

1.6 Exercises

1.1. Generalize the algorithm for the rectangle learning game to prove that if $\mathcal{C}_n$ is the class of all axis-aligned hyperrectangles in n-dimensional Euclidean space $\Re^n$, then $\mathcal{C}$ is efficiently PAC learnable.

1.2. Let $f(\cdot)$ be an integer-valued function, and assume that there does not exist a randomized algorithm taking as input a graph G and a parameter $0 < \delta \leq 1$ that runs in time polynomial in $1/\delta$ and the size of G, and that with probability at least $1 - \delta$ outputs "no" if G is not k-colorable and outputs an $f(k)$-coloring of G otherwise. Then show that for some $k \geq 3$, k-term DNF formulae are not efficiently PAC learnable using $f(k)$-term DNF formulae.

1.3. Consider the following **two-oracle** variant of the PAC model: when $c \in \mathcal{C}$ is the target concept, there are separate and arbitrary distributions $\mathcal{D}_c^+$ over only the positive examples of c and $\mathcal{D}_c^-$ over only the negative examples of c. The learning algorithm now has access to two oracles $EX(c, \mathcal{D}_c^+)$ and $EX(c, \mathcal{D}_c^-)$ that return a random positive example or a random negative example in unit time. For error parameter ϵ, the learning algorithm must find a hypothesis satisfying $\mathbf{Pr}_{x \in \mathcal{D}_c^+}[h(x) = 0] \leq \epsilon$ and $\mathbf{Pr}_{x \in \mathcal{D}_c^-}[h(x) = 1] \leq \epsilon$. Thus, the learning algorithm may now explicitly request either a positive or negative example, but must find a hypothesis with small error on both distributions.

Let $\mathcal{C}$ be any concept class and $\mathcal{H}$ be any hypothesis class. Let h_0 and h_1 be representations of the identically 0 and identically 1 functions, respectively. Prove that $\mathcal{C}$ is efficiently PAC learnable using $\mathcal{H}$ in the original one-oracle model if and only if $\mathcal{C}$ is efficiently PAC learnable using $\mathcal{H} \cup \{h_0, h_1\}$ in the two-oracle model.

1.4. Let $\mathcal{C}$ be any concept class and $\mathcal{H}$ be any hypothesis class. Let h_0 and h_1 be representations of the identically 0 and identically 1 functions, respectively. Show that if there is a randomized algorithm for efficiently PAC learning $\mathcal{C}$ using $\mathcal{H}$, then there is a deterministic algorithm for efficiently PAC learning $\mathcal{C}$ using $\mathcal{H} \cup \{h_0, h_1\}$.

1.5. In Definition 2, we modified the PAC model to allow the learning algorithm time polynomial in n and $size(c)$, and also provided the value $size(c)$ as input. Prove that this input is actually unnecessary: if there is an efficient PAC learning algorithm for $\mathcal{C}$ that is given $size(c)$ as input, then there is an efficient PAC learning algorithm for $\mathcal{C}$ that is not given