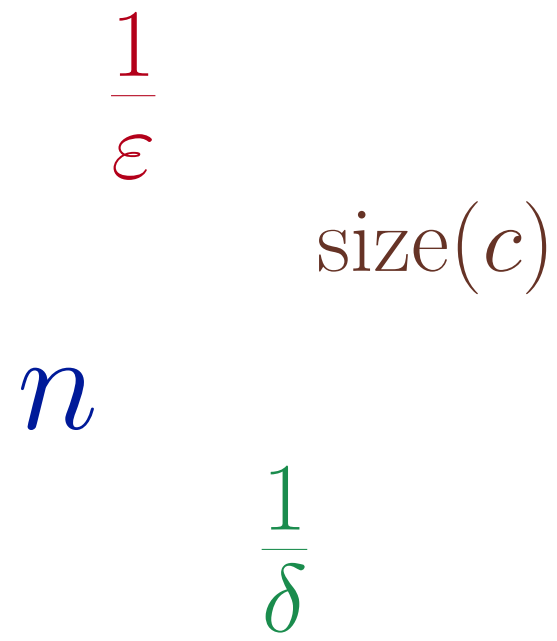
Computational Learning Theory

(Lecture Notes)



Varun Kanade

Contents

Preface

What is computational learning theory?

Machine learning techniques lie at the heart of many technological applications that are used on a daily basis. When using a digital camera, the boxes that appear around faces are produced using a machine learning algorithm. When streaming portals such as BBC iPlayer or Netflix suggest what a user might like to watch next, they are also using machine learning algorithms to provide these recommendations. In fact, more likely than not, any substantial technology that is in use these days has some component that uses machine learning techniques.

The field of (computational) learning theory develops precise mathematical formulations of the more vague notion of *learning from data*. Having precise mathematical formulations allows one to answer questions such as:

(i) What types of functions are easy to learn?

(ii) Are there types of functions that are hard to learn?

(iii) How much data is required to learn a function of a particular type?

(iv) How much computational power is needed to learn certain types of functions?

Positive as well as negative answers to these questions are of great interest. For example, one of the key considerations is to design and analyse learning algorithms that are guaranteed to learn certain types of functions using modest amount of data and reasonable runnning time. For the most part, we will take the view that as long as the resources used can be bounded by a polynomial function of the problem size, the learning algorithm is efficient. Obviously, as is the case in the analysis of algorithms, there may be situations where just being polynomial time may not be considered efficient enough; the existence of polynomial-time learning algorithms is however a good first step in separating easy and hard learning problems. Some of the algorithms we study will not run in polynomial time at all, but they will still be much better than brute force algorithms.

There is a vast body of literature that is often called *Statistical Learning Theory*. To some extent this distinction between *statistical* and *computational* learning theory is rather artificial and we shall make use of several concepts introduced in that theory such as VC dimension and Rademacher complexity. In this course, greater emphasis will be placed on computational considerations. Research in *computational learning theory* has uncovered interesting phenomena such as the existence of certain types of functions that can be learnt if

computational resources are not a consideration, but cannot be learnt in polynomial time. Other examples demonstrate a tradeoff between the amount of data and the algorithmic running time, i.e. the running time of the algorithm can be reduced by using more data. More importantly, placing the question of learning in a clear computational framework allows one to reason about other kinds of (computational) resources such as memory, communication, privacy, etc. that may be a consideration for the learning problem at hand.

Chapter 1

Probably Approximately Correct Learning

Our goal in this chapter is to gradually build up the probably approximately correct (PAC) learning framework while emphasizing the key components of the learning model. We will discuss various model choices in detail; the exercises and some results in later chapters explore the robustness of the PAC learning framework to slight variants of these design choices. As the goal of computational learning theory is to shed light on the phenomenon of automated learning, such robustness is of key importance.

1.1 A Rectangle Learning Game

Let us consider the following rectangle learning game. We are given some points in the Euclidean plane, some of which are labeled positive (+) and others negative (−). Furthermore, we are guaranteed that there is an axis-aligned rectangle such that all the points inside it are labelled positive, while those outside are labelled negative. However, this rectangle itself is not revealed to us. Our goal is to produce a rectangle that is "close" to the *true hidden* rectangle that was used to label the observed data (see Fig. 1.1(a)).

Although the primary purpose of this example is pedagogic, it may be worth providing a scenario where such a (fake) learning problem may be relevant. Suppose that the two dimensions measure the curvature and length of bananas. The points that are labelled positive have *medium* curvature and *medium* length and represent the bananas that would pass "stringent" EU regulations. However, the actual lower and upper limits that "define" *medium* in each dimension are hidden. Thus, we wish to learn some rectangle that will be good enough to predict whether bananas we produce would pass the regulators' tests or not.

Let R be the unknown rectangle used to label the points. We can express the labelling process using a *boolean* function $c_R : \mathbb{R}^2 \to \{+, -\}$, where $c_R(\mathbf{x}) = +$, if $\mathbf{x}$ is inside the rectangle R and $c_R(\mathbf{x}) = -$, otherwise.[1]

[1] We refer to functions whose range has size at most 2 as *boolean* functions. From the point of view of machine learning, the exact values in the range are unimportant. We will frequently use $\{+, -\}$, $\{0, 1\}$ and $\{-1, +1\}$ as the possible options for the range depending on the context (and at times make rather unintuitive transformations between these possibilities).

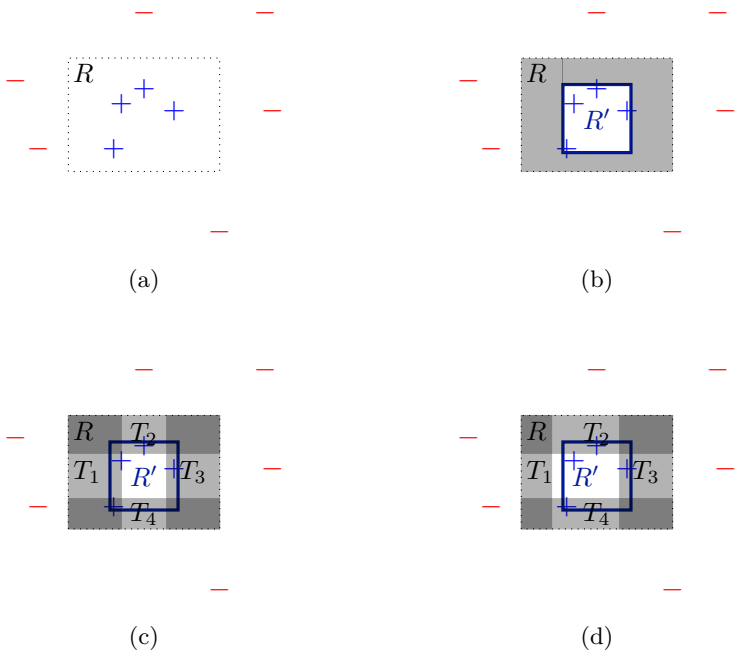


Figure 1.1: (a) Data received for the rectangle learning game. The rectangle R used to generate labels is hidden from the learning algorithm. (b) The *tightest fit* algorithm produces a rectangle R'. (c) & (d) The regions T_1, T_2, T_3 and T_4 contain $\epsilon/4$ mass each under D (for two different distributions D).

Let us consider the following simple algorithm. We consider the *tightest* possible axis-aligned rectangle that can fit all the positively labelled data inside it; let us denote this rectangle by R' (Fig. 1.1(b)). Our prediction function or *hypothesis* $h_{R'} : \mathbb{R}^2 \to \{+, -\}$ is the following: if $\mathbf{x} \in R'$, $h_{R'}(\mathbf{x}) = +$, else $h_{R'}(\mathbf{x}) = -$.[2] Let us consider the following questions:

- Have we learnt the function c_R ?
- How good is our prediction function $h_{R'}$?

Let R denote the *true* rectangle that actually defines the labelling function c_R. Since we've chosen the tightest possible fit, the rectangle R' must be entirely contained inside R. Consider the shaded region shown in Fig. 1.1(b). For any point $\mathbf{x}$ that is in this shaded region, it must be that $h_{R'}(\mathbf{x}) = -$, while $c_R(\mathbf{x}) = +$. In other words, our prediction function $h_{R'}$ would make errors on all of these points. If we had to make predictions on points that mostly lie in this region our hypothesis would be quite bad. This raises an important point that the data that is used to learn a hypothesis should be similar to the data on which the hypothesis will be tested. We will now formalise this notion.

Let D be a probability distribution over $\mathbb{R}^2$; in the ensuing discussion, we will assume that D can be expressed using a density function that is defined

[2]For the sake of concreteness, let us say that points on the sides are considered to be inside the rectangle.

While no knowledge of machine learning is required to complete this course, it would of course be helpful to make connections with the techniques and terminology in machine learning. This discussion appears in coloured boxes and can be safely ignored for those uninterested in applied machine learning.

In machine learning, one makes the distinction between the *training* and *test* datasets; when done correctly the *empirical error* on the test dataset would give an unbiased estimate of what we refer to as error in Eqn. (1.1). In machine learning it is also common to use a *validation set*; this is often done because multiple models are trained on the training set (possibly because of hyperparameters) and one of them needs to be picked. Picking them using their performance on the training set may result in overfitting. In this course, we will adopt the convention that model selection is also part of the learning algorithm and not make a distinction between the training and validation sets. (For example, see Exercise 1.3.)

over all of $\mathbb{R}^2$ and is continuous.[3] The *training data* consists of m points that are drawn independently according to D and then labelled according to the function c_R. We will define the error of a hypothesis $h_{R'}$ with respect to the target function c_R and distribution D as follows:

$$\text{err}(h_{R'}; c_R, D) = \mathbb{P}_{\mathbf{x}\sim D}\left[h_{R'}(\mathbf{x}) \neq c_R(\mathbf{x})\right] \tag{1.1}$$

Whenever the target c_R and distribution D are clear from context, we will simply refer to this as $\text{err}(h_{R'})$.

We will now show that in fact our algorithm outputs an $h_{R'}$ that is quite good, in the sense that given any $\epsilon > 0$ as the target error, with high probability (at least $1 - \delta$), given a sufficiently large training sample, it will output $h_{R'}$ such that $\text{err}(h_{R'}; c_R, D) \leq \epsilon$. Consider four rectangular strips T_1, T_2, T_3, T_4 that are chosen along the sides of the rectangle R (and lying inside R) such that the probability that a random point drawn according to D lands in some T_i is exactly $\epsilon/4$.[4] Note that some of these strips overlap, e.g. T_1 and T_2 (see Fig. 1.1(c)). The probability that a point drawn randomly according to D lies in the set $T_1 \cup T_2 \cup T_3 \cup T_4$ is at most ϵ (a fact that can be proved formally using the union bound (cf. Appendix A.1)). If we can guarantee that the training data of m points contains at least one point from each of T_1, T_2, T_3 and T_4, then the tightest fit rectangle R' will be such that $R \setminus R' \subset T_1 \cup T_2 \cup T_3 \cup T_4$, and as a consequence, $\text{err}(h_{R'}; c_R, D) \leq \epsilon$. This is shown in Fig. 1.1(c); note that if even one of the T_i do not contain any point in the data, this may cause a problem, in the sense that the region of disagreement between R and R' may have probability mass greater than ϵ (see Fig. 1.1(d)).

Let A_1 be the event that when m points are drawn independently according to D, none of them lies in T_1. Similarly, define the events A_2, A_3, A_4 for

[3] This assumption is not required; in the exercises you are asked to show how the assumption can be removed.

[4] Assuming that the distribution D can be expressed using a continuous density function that is defined over all of $\mathbb{R}^2$, such strips always exist. Otherwise, the algorithm is still correct, however, the analysis is slightly more tedious and is left as Exercise 1.1.

T_2, T_3, T_4. Consider the event $\mathcal{E} = A_1 \cup A_2 \cup A_3 \cup A_4$. If $\mathcal{E}$ does not occur, then we have already argued that $\text{err}(h_{R'}; c_R, D) \leq \epsilon$. We will use the union bound to bound $\mathbb{P}[\mathcal{E}]$ (cf. Appendix A.1). To begin, let us compute $\mathbb{P}[A_1]$. The probability that a single point drawn according to D does not land in T_1 is exactly $1-\epsilon/4$; so the probability that after m independent draws from D, none of the points are in T_1 is $\left(1 - \frac{\epsilon}{4}\right)^m$. By a similar argument, $\mathbb{P}[A_i] = \left(1 - \frac{\epsilon}{4}\right)^m$ for $i = 1, \ldots, 4$. Thus, we have

$$\begin{aligned}
\mathbb{P}[\mathcal{E}] &\leq \sum_{i=1}^{4} \mathbb{P}[A_i] && \text{The Union Bound (A.1).} \\
&= 4\left(1 - \frac{\epsilon}{4}\right)^m \\
&\leq 4\exp\left(-\frac{m\epsilon}{4}\right). && \text{As } 1 - x \leq e^{-x} \text{ (B.1).}
\end{aligned}$$

For any $\delta > 0$, picking $m \geq \frac{4}{\epsilon}\log\left(\frac{4}{\delta}\right)$ suffices to ensure that $\mathbb{P}[\mathcal{E}] \leq \delta$. In other words, with probability at least $1 - \delta$, $\text{err}(h_{R'}; c_R, D) \leq \epsilon$.

A couple of remarks are in order. We should think of ϵ as being the accuracy parameter and δ being the confidence parameter. The bound $m \geq \frac{4}{\epsilon}\log\left(\frac{4}{\delta}\right)$ suggests that as we demand higher accuracy (smaller value of ϵ) and higher confidence (smaller value of δ) of our learning algorithm, we need to supply more data.[5] This is indeed a reasonable requirement. Furthermore, the cost of achieving higher accuracy and higher confidence is relatively modest. For example, if we want to halve the error while keeping the confidence parameter constant, say go from $\epsilon = 0.02$ to $\epsilon = 0.01$, the amount of data required (as suggested by the bound) only doubles.[6]

1.2 Key Components of the PAC Learning Framework

We will use the insights gleaned from the rectangle learning game to develop key components of a mathematical framework for automatic learning from data. First let us make a few observations:

1. The learning algorithm does not know the target concept to be learnt (obviously, otherwise there is nothing to learn!). However, the learning algorithm does know the set of possible target concepts. In the rectangle learning game, the unknown target is always an axis-aligned rectangle.
2. The learning algorithm has access to data drawn from some distribution D. We do assume that the observations are drawn independently according to D. However, no assumption is made on the distribution D itself. This reflects the fact that the environments in which learning agents operate may be very complex and it is unrealistic to assume that the observations are generated according to some distribution that is easy to describe.

[5] So far, we have only established sufficient conditions, i.e. upper bounds, on the sample complexity required for learning. In later chapters we will establish necessary conditions, i.e. lower bounds on the amount of data required for learning algorithms.

[6] We are using the word "required" a bit losely here. All we can say is our present analysis of this particular algorithm suggests that the amount of data required scales linearly as $\frac{1}{\epsilon}$. We will see lower bounds of this nature that hold for any algorithm in later chapters.

3. The output hypothesis is evaluated with respect to the same distribution D that generated the training data.
4. We would like learning algorithms to be *statistically efficient*, i.e. they should require a relatively small training sample to guarantee high accuracy and confidence, as well as *computationally efficient*, i.e. they should run in a reasonable amount of time. In general, we shall take the view that learning algorithms for which the training sample size and running time scales polynomially with the size parameters are efficient. However, in some cases we will be more precise and specify the exact running time and sample size.

Let us now formalise a few other concepts related to learning.

Instance Space

Let X denote the set of possible instances; an instance is the *input* part, $\mathbf{x}$, of a training example $(\mathbf{x}, y)$, and y is the target label. In the rectangle learning game, the instances were points in $\mathbb{R}^2$; the instance space was $\mathbb{R}^2$. When considering binary classification problems for images, the instances may be 3 dimensional arrays, containing the RGB values of each pixel. Mostly, we shall be concerned with the case when $X = \{0,1\}^n$ or $X = \mathbb{R}^n$; other instance spaces can be usually mapped to one of these, as is often done in machine learning.

Concept Class

A *concept* c over an instance space X is a boolean function $c : X \to \{0,1\}$. (We will consider learning target functions that are not boolean later in the course.) A concept class C over X is a collection of concepts c over X. In the rectangle learning game, the concept class is the set of all axis-aligned rectangles in $\mathbb{R}^2$. The learning algorithm has knowledge of C, but not of the specific concept $c \in C$ that is used to label the observations.

Data Generation

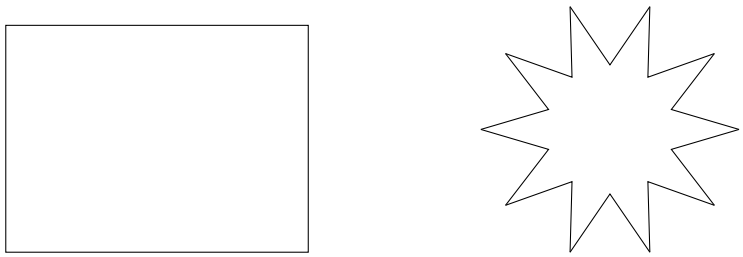
Let D be a probability distribution over X. The training data is obtained as follows. An instance $\mathbf{x} \in X$ is drawn according to the distribution D. If c is the target concept, the instance $\mathbf{x}$ is labelled accordingly as $c(\mathbf{x})$. The learning algorithm observes the example $(\mathbf{x}, c(\mathbf{x}))$. We will refer to this process as an example oracle, denoted by $\mathsf{EX}(c, D)$. We assume that a learning algorithm can query the oracle $\mathsf{EX}(c, D)$ at unit cost and each query yields an *independent* training example.

1.2.1 PAC Learning: Take I

Let $h : X \to \{0,1\}$ be some hypothesis; we typically refer to the boolean function output by a learning algorithm as a *hypothesis* to distinguish it from the *target*. For a distribution D over X and a fixed target $c \in C$, the error of h with respect to c and D is defined as:

$$\mathrm{err}(h; c, D) = \mathbb{P}_{\mathbf{x} \sim D}\left[h(\mathbf{x}) \neq c(\mathbf{x})\right]. \tag{1.2}$$

When c and D are clear from context, we will simply refer to this as $\mathrm{err}(h)$.



(a) (b)

Figure 1.2: Different shape concepts in $\mathbb{R}^2$.

***Definition 1.1* – PAC Learning: Take I.** *Let C be a concept class over X. We say that C is* PAC (take I) learnable *if there exists a learning algorithm L that satisfies the following: for every concept $c \in C$, for every distribution D over X, for every $0 < \epsilon < 1/2$ and $0 < \delta < 1/2$, if L is given access to $\mathsf{EX}(c, D)$ and inputs ϵ and δ, L outputs a hypothesis $h \in C$ that with probability at least $1 - \delta$ satisfies $\mathrm{err}(h) \leq \epsilon$. The probability is over the random examples drawn from $\mathsf{EX}(c, D)$ as well as any internal randomisation of L. The number of calls made to $\mathsf{EX}(c, D)$* (sample complexity) *must be bounded by a polynomial in $\frac{1}{\epsilon}$ and $\frac{1}{\delta}$.*

We further say that C is efficiently *PAC (take I) learnable if the running time of L is polynomial in $1/\epsilon$ and $1/\delta$.*

The term PAC stands for probably approximately correct. The *approximately correct* part captures the notion that the most that can be guaranteed is that the error of the output hypothesis can be bounded to be below a desired level; demanding higher accuracy (lower ϵ) is possible, but comes at a cost of increased running time and sample complexity. In most cases, achieving *exactly* zero error is infeasible as it is possible that two target concepts may be identical except on one instance which is very unlikely to be drawn according to the distribution D.[7] The *probably* part captures the notion that there is some chance that the algorithm may fail completely. This may happen because the observations are not representative of the underlying data distribution, a low probability event, though very much a possible event. Our confidence (lower δ) in the correctness of our algorithm is increased as we allow more sample complexity and running time.

Based on our analysis of the rectangle learning game in Section 1.1, we have essentially already proved the following theorem.

Theorem 1.2. *The concept class of axis-aligned rectangles in $\mathbb{R}^2$ is efficiently PAC (take I) learnable.*

1.2.2 PAC Learning: Take II

Having proved our first result in PAC learning, let us discuss a couple of issues that we have glossed over so far. The first question concerns that of the

[7] In later chapters, we will consider different learning frameworks under which exact learning, i.e. achieving *zero* error, is possible.

complexity of the concepts that we are trying to learn. For example, consider the question of learning rectangles (Fig. 1.2(a)) versus more complex shapes such as shown in Fig. 1.2(b). Intuitively, we believe that it should be harder to learn concepts defined by shapes like in Fig. 1.2(b) than rectangles. Thus, within our mathematical learning framework, an algorithm that learns a more complex class should be allowed more resources (sample size, running time, memory, etc.). In order to represent an axis-aligned rectangle, we only need to store four real numbers, the lower and upper limits in both the x and y directions. The number of real numbers used to represent more complex shapes is higher.[8]

The question of representation is better elucidated by taking the case of boolean functions defined on the *boolean hypercube* $X = \{0,1\}^n$, the set of length n bit vectors. Consider a boolean function $f : X \to \{0,1\}$; there are several ways of representing boolean functions. One option is to keep the entire truth table with 2^n entries. Alternatively, we may represent f as a circuit using $\wedge$ (and), $\vee$ (or) and $\neg$ (not) gates. We may ask that f be represented in disjunctive normal form (DNF), i.e. of the form shown below

$$(x_1 \wedge \bar{x}_3 \wedge x_7 \wedge \cdots) \vee (x_2 \wedge x_4 \wedge \bar{x}_8 \wedge \cdots) \vee \cdots \vee (x_1 \wedge x_3).$$

The choice of representation can make a huge difference in terms of the amount of memory required to store a *description* of the boolean function. You are asked to show this in the case of the parity function $f = x_1 \oplus x_2 \oplus \cdots \oplus x_n$ in Exercise 1.2. There are other possible representations of boolean functions, such as decision lists, decision trees, neural networks, etc., which we will encounter later in the course.

Representation Scheme

Abstractly, a representation scheme for a concept class C is an onto function $R : \Sigma^* \to C$, where Σ is a finite alphabet.[9] Any $\sigma \in \Sigma^*$ satisfying $R(\sigma) = c$ is called a representation of c. We assume that there is a function, size : $\Sigma^* \to \mathbb{N}$, that measures the size of a representation. A concept $c \in C$ may in general have multiple representations under R. For example, there are several boolean circuits that compute exactly the same boolean function. We can define the function size on the set C by defining, $\text{size}(c) = \min_{\sigma : R(\sigma) = c} \{\text{size}(\sigma)\}$. When we refer to a concept class, we will assume by default that it is associated with a representation scheme and a size function, so that size(c) is well defined for $c \in C$. In most cases of interest, there will be a natural notion of size that makes sense for the learning problem at hand; however, some of the exercises and coloured boxes encourage you to explore the subtleties involved with representation size in greater detail.

Instance Size

Typically, instances in a learning problem also have a natural notion of size associated with them; roughly we may think of the size of an instance as the

[8] We shall assume that our computers can store and perform elementary arithmetic operations (addition, multiplication, division) on real numbers at unit cost.

[9] If representing the concept requires using real numbers, such as in the case of rectangles, we may use $R : (\Sigma \cup \mathbb{R})^* \to C$. Representing a real number will assumed to be unit cost.

When learning a target concept $c \in C_n$, in general, allowing the learning algorithm resources that increase with size(c) will be necessary. Mostly, we will consider concept classes C_n over X_n for which every size(c) can be bounded for every $c \in C_n$ by some fixed polynomial function of n. Thus, *efficient* PAC learning simply requires designing algorithms that run in time polynomial in n, $\frac{1}{\epsilon}$ and $\frac{1}{\delta}$.

Definition 1.3 is general enough to allow for the existence of "efficient" PAC learning algorithms if an overly verbose representation scheme is chosen. For example, the class of *all boolean functions* is efficiently PAC-learnable when the representation scheme uses truth tables. On the other hand, if we represent boolean functions as *decision trees*, or *boolean circuits*, or even *boolean formulae* in disjunctive normal form (DNF), it is widely believed that the class of boolean functions is *not* efficiently PAC-learnable. We will provide some evidence for this assertion based on cryptographic assumptions and on hardness of learning in the statistical query (SQ) learning model in later chapters.

amount of memory required to store it. For example, 10×10 black and white images can be represented using 100 bits, whereas 1024×1024 colour images will require over 3 million real numbers. When faced with larger instances, we should expect that learning algorithms will require time; at the very least they have to read the input data! In this course, we will only consider settings where the instance space is either $X_n = \{0,1\}^n$ or $X_n = \mathbb{R}^n$. We denote by C_n a concept class over X_n. We consider the instance space $X = \bigcup_{n \geq 1} X_n$ and the concept class $C = \bigcup_{n \geq 1} C_n$ as representing increasingly larger instances (and concepts on them).

Definition 1.3 **– PAC Learning: Take II.** *For $n \geq 1$, let C_n be a concept class over instance space X_n and let $C = \bigcup_{n \geq 1} C_n$ and $X = \bigcup_{n \geq 1} X_n$. We say that C is* PAC (take II) learnable *if there exists a learning algorithm L that satisfies the following: for every $n \in \mathbb{N}$, for every concept $c \in C_n$, for every distribution D over X_n, for every $0 < \epsilon < 1/2$ and $0 < \delta < 1/2$, if L is given access to $\mathsf{EX}(c, D)$ and inputs n,* size(c)*, ϵ and δ, L outputs $h \in C_n$ that with probability at least $1 - \delta$ satisfies* $\mathrm{err}(h) \leq \epsilon$*. The probability is over the random examples drawn from $\mathsf{EX}(c, D)$ as well as any internal randomization of L. The number of calls made to $\mathsf{EX}(c, D)$* (sample complexity) *must be bounded by a polynomial in n,* size(c)*, $\frac{1}{\epsilon}$ and $\frac{1}{\delta}$.*

We further say that C is efficiently *PAC (take II) learnable if the running time of L is polynomial in n,* size(c)*, $1/\epsilon$ and $1/\delta$.*

1.3 Learning Conjunctions

Having formulated a notion of learning, let us consider a second learning problem. Let $X_n = \{0,1\}^n$ represent the instance space of size n; note that each element $\mathbf{x} \in X_n$ denotes a possible assignment to n boolean variables $z_1, \ldots, z_n$; let $X = \bigcup_{n \geq 1} X_n$. Let $\mathsf{CONJUNCTIONS}_n$ denote the concept class of conjunctions over the n boolean variables $z_1, \ldots, z_n$. A *literal* is either a boolean variable z_i or its negation $\bar{z}_i$. A conjunction (sometimes also called a *term*) is

Algorithm 1.1: CONJUNCTIONS Learner

Input: n, m, access to $\mathsf{EX}(c, D)$
// initialize hypothesis conjunction with all literals
Set $h = z_1 \wedge \bar{z}_1 \wedge z_2 \wedge \bar{z}_2 \wedge \cdots \wedge z_n \wedge \bar{z}_n$
for $i = 1, \ldots, m$ **do**
    draw $(\mathbf{x}_i, y_i)$ from $\mathsf{EX}(c, D)$
    if $y_i == 1$ **then** // ignore negative examples
        for $j = 1, \ldots n$ **do**
            if $x_{i,j} = 0$ **then** // j^{th} bit of i^{th} instance is 0
                Drop z_i from h
            else // j^{th} bit of i^{th} instance is 1
                Drop $\bar{z}_i$ from h
Output: h

simply an *and* ($\wedge$) of literals. An example conjunction φ with $n = 10$ (say) is show in (1.3).

$$\varphi = z_1 \wedge \bar{z}_3 \wedge \bar{z}_8 \wedge z_9. \tag{1.3}$$

Formally, a conjunction over $z_1, \ldots, z_n$ can be represented by two subsets $P, N \subset [n]$. Such a pair of sets P, N represents the conjunction $\varphi_{P,N}$ defined as

$$\varphi_{P,N} = \bigwedge_{i \in P} z_i \wedge \bigwedge_{i \in N} \bar{z}_i. \tag{1.4}$$

In (1.4), the sets P and N represent the positive and negative literals that appear in the conjunction $\varphi_{P,N}$ respectively. We have not required that $P \cap N = \emptyset$; this allows us to represent a boolean function that is 0 over the entire hypercube (falsehood), e.g. as $z_1 \wedge \bar{z}_1$. Both P and N could be empty, representing an empty conjuntion that is 1 over the entire boolean hypercube (tautology). Formally

$$\mathsf{CONJUNCTIONS}_n = \{\varphi_{P,N} \mid P, N \subset [n]\},$$
$$\mathsf{CONJUNCTIONS} = \bigcup_{n \geq 1} \mathsf{CONJUNCTIONS}_n.$$

When representing a conjunction over n boolean variables, each of the sets P and N can be represented by a *bit-string* of length n; as a result any conjunction can be represented using a bit-string of length $2n$. As there are at least 2^n conjunctions (can you count the number of conjunctions exactly?) we should expect to need at least n bits to represent a conjunction. Thus, this representation scheme is fairly succinct. Thus, our goal is to design an algorithm that runs in time polynomial in n, $1/\epsilon$ and $1/\delta$.

Let c denote the target conjunction. The example oracle $\mathsf{EX}(c, D)$ returns examples of the form $(\mathbf{x}, y)$ where $y \in \{0, 1\}$. $y = 1$ if c evaluates to 1 (true) after assigning $z_i = x_i$ for $i = 1, \ldots, n$. In other words, $y = 1$ if $\mathbf{x}$ is a satisfying assignment of the conjunction c, and 0 otherwise.

Algorithm 1.1 is learns the concept class CONJUNCTIONS. We describe the high-level idea before giving the complete proof.

i) The algorithm begins by conservatively constructing a hypothesis h that is a conjunction of all the $2n$ possible literals. Clearly, this conjunction will always output 0 on any given input. The algorithm then makes use of data to remove harmful literals from h.

ii) The algorithm draws m independent examples $(\mathbf{x}_i, y_i)$ from the oracle $\mathsf{EX}(c, D)$; all the *negatively labelled* examples ($y_i = 0$) are ignored. For positively labelled examples literals that would cause these to be labelled as negative by h are dropped from h. The resulting hypothesis h is returned. Thus, the algorithm outputs the "longest" conjunction (containing the most number of literals) that is consistent with the observed data. This is because only those literals that absolutely cannot be part of the target conjunction (as dictated by the positively labelled data) are dropped.

Theorem 1.4. *Provided* $m \geq \frac{2n}{\epsilon}\log\left(\frac{2n}{\delta}\right)$, *Algorithm 1.1* efficiently *PAC (take II) learns the concept class* CONJUNCTIONS.

Proof. Let c be the target conjunction and D the distribution over $\{0,1\}^n$. For a literal ℓ (which may be z_i or $\bar{z}_i$), let $p(\ell) = \mathbb{P}_{\mathbf{x}\sim D}\left[c(\mathbf{x}) = 1 \wedge \ell(\mathbf{x}) = 0\right]$; here, we interpret ℓ itself as a conjunction with 1 literal. Thus, if $\ell = z_i$, then $\ell(\mathbf{x}) = x_i$; if $\ell = \bar{z}_i$, then $\ell(\mathbf{x}) = 1 - x_i$. Notice that if $p(\ell) > 0$, then the literal ℓ cannot be present in c; if it were, then there can be no $\mathbf{x}$ such that $c(\mathbf{x}) = 1$ and $\ell(\mathbf{x}) = 0$.

We define a literal ℓ to be *harmful* if $p(\ell) \geq \frac{\epsilon}{2n}$. We will ensure that all harmful literals are eliminated from the hypothesis h. For a harmful literal ℓ, let A_ℓ denote the event that after m independent draws from $\mathsf{EX}(c, D)$, ℓ is not eliminiated from h. Note that this can only happen if no $\mathbf{x}$ such that $c(\mathbf{x}) = 1$ but $\ell(\mathbf{x}) = 0$ is drawn. This can happen with probability at most $\left(1 - \frac{\epsilon}{2n}\right)^m$. Let B denote the set of harmful literals and let $\mathcal{E} = \bigcup_{\ell\in B} A_\ell$ be the event that at least one bad literal survives in h. We shall choose m large enough so that $\mathbb{P}[\mathcal{E}] \leq \delta$. Consider the following,

$$\begin{aligned}
\mathbb{P}[\mathcal{E}] &\leq \sum_{\ell\in B}\mathbb{P}[A_\ell] && \text{By the Union Bound (A.1).}\\
&\leq 2n\left(1-\frac{\epsilon}{2n}\right)^m && |B| \leq 2n \text{ and for each } \ell\in B, \mathbb{P}[A_\ell] \leq \left(1-\frac{\epsilon}{2n}\right)^m.\\
&\leq 2n\exp\left(-\frac{m\epsilon}{2n}\right). && \text{As } 1-x \leq e^{-x} \text{ (B.1).}
\end{aligned}$$

Thus, whenever $m \geq \frac{2n}{\epsilon}\log\left(\frac{2n}{\delta}\right)$, we know that $\mathbb{P}[\mathcal{E}] \leq \delta$. Now, suppose that $\mathcal{E}$ does not occur, i.e. all harmful literals are eliminated from h. Let B^c be the set of literals that are not harmful.

$$\begin{aligned}
\mathrm{err}(h) &= \mathbb{P}_{\mathbf{x}\sim D}\left[c(\mathbf{x}) = 1 \wedge h(\mathbf{x}) = 0\right]\\
&\leq \sum_{\ell\in B^c}\mathbb{P}_{\mathbf{x}\sim D}\left[c(\mathbf{x}) = 1 \wedge \ell(\mathbf{x}) = 0\right]\\
&\leq 2n\cdot\frac{\epsilon}{2n} \leq \epsilon.
\end{aligned}$$

This completes the proof. □

It is worth pointing out that Algorithm 1.1 only makes use of positively labelled examples. The algorithm works correctly even if no positively labelled examples are obtained from the oracle $\mathsf{EX}(c, D)$; this is because if no positive examples are obtained after drawing m independent examples (for a sufficiently large m), then returning a hypothesis h that always predicts 0 is sufficient to achieve low error.

1.3.1 Learning k-CNF

We can generalize Algorithm 1.1 to learn richer classes of boolean functions. A *clause* is a disjunction ($\vee$) of boolean literals. The *length* of a clause is the number of (not necessarily distinct) literals in it. For example, $z_1 \vee \bar{z}_7 \vee \bar{z}_{15}$ is a clause of length 3. Let $\mathsf{clauses}_{n,k}$ denote the set of all clauses of length exactly k on the n boolean variables $z_1, \ldots, z_n$. We define the class of boolean functions that can be written in conjunctive normal form using clauses of length exactly k as:

$$k\text{-}\mathsf{CNF}_n = \{\bigwedge_i c_i \mid c_i \in \mathsf{clauses}_{n,k}\},$$
$$k\text{-}\mathsf{CNF} = \bigcup_{n \geq 1} k\text{-}\mathsf{CNF}_n.$$

A representation of boolean function as in the class k-CNF is called a k-CNF formula. There are at most $(2n)^k$ possible clauses of length k on n boolean variables and so each k-CNF formula over n variables can have at most $(2n)^k$ clauses. (Allowing clauses to have the same literal multiple times and letting the order of literals matter, we shall assume in the rest of this section that there are exactly $(2n)^k$ clauses of length k.)

It is completely straightforward to modify Algorithm 1.1 to start with a hypothesis h that is a k-CNF formula with all $(2n)^k$ clauses and eliminate the clauses that cause positive examples to be labelled negative. This algorithm is *efficient* if we assume the representation scheme to have length $(2n)^k$, and in any case the running time and sample complexity is polynomial in n for any fixed constant k.

Rather than redo the proof of Theorem 1.4, we shall sketch a different approach that also introduces the notion of a reduction between learning problems. Suppose the target function is a k-CNF formula over the boolean variables $z_1, \ldots, z_n$; we create new boolean variables $(z'_{\ell_1,\ldots,\ell_k})$ where each ℓ_i is either some z_j or $\bar{z}_j$. When placed in parentheses, $(z'_{\ell_1,\ldots,\ell_k})$ denotes the set of all possible $(2n)^k$ boolean variables; whereas by itself $z'_{\ell_1,\ldots,\ell_k}$ denotes the specific variable corresponding to the tuple of literals $(\ell_1, \ldots, \ell_k)$. The boolean variable $z'_{\ell_1,\ldots,\ell_k}$ is meant to respresent the clause $\ell_1 \vee \cdots \vee \ell_k$. Given an assignment to the boolean variables $z_1, \ldots, z_n$ denoted by some bit-vector $\mathbf{x} \in \{0,1\}^n$, an assignment to $(z'_{\ell_1,\ldots,\ell_k})$ can be uniquely determined, by assigning the variable $z'_{\ell_1,\ldots,\ell_k}$ the value 1 if and only if $\mathbf{x}$ is a satisfying assignment of the clause $\ell_1 \vee \cdots \vee \ell_k$. This yields a bit vector in $\{0,1\}^{(2n)^k}$ that represents the assignment to all $(z'_{\ell_1,\ldots,\ell_k})$. Let us denote this map from $\{0,1\}^n$ to $\{0,1\}^{(2n)^k}$ by f and observe that it is injective.

Now consider the following "natural" bijective map, denoted by g, between k-CNF formulae over $z_1, \ldots, z_n$ and *monotone conjunctions* over $(z'_{\ell_1,\ldots,\ell_k})$:

given a k-CNF formula φ, the literal $z'_{\ell_1,\ldots,\ell_k}$ appears in the monotone conjunction $f(\varphi)$ if and only if φ contains the clause $\ell_1 \vee \cdots \ell_k$.[10] (A conjunction is *monotone* if it does not contain any *negated* literals; Algorithm 1.1 modified to start with $h = z_1 \wedge \cdots \wedge z_n$ clearly learns the class of *monotone conjunctions*.)

Let D be a distribution over $\{0,1\}^n$ and let $f(D)$ denote the distribution over $\{0,1\}^{(2n)^k}$ obtained by first drawing $\mathbf{x}$ according to D and then applying f to $\mathbf{x}$. Let $c, h \in k\text{-CNF}_n$, then it can be easily verified that

$$\mathrm{err}(h; c, D) = \mathrm{err}(g(h); g(c), f(D)).$$

The only thing that remains is to observe that the maps f, g and g^{-1} are (trivially) polynomial time computable and that the example oracle $\mathsf{EX}(g(c), f(D))$ can be simulated given access to $\mathsf{EX}(c, D)$. Thus, we have proved the following result.

Theorem 1.5. *The concept class k-CNF is efficiently PAC (take II) learnable.*

1.4 Hardness of Learning 3-term DNF

Having seen a few examples of concept classes that are PAC (take II) learnable, we shall temper our optimism by proving that a class of boolean functions (not significanly more complex than CONJUNCTIONS) is not PAC (take II) learnable, assuming an unproven, but widely believed, conjecture from computational complexity theory. The class is that of boolean functions that can expressed as DNF formulae with exactly 3 terms. A term is simply a conjunction over n boolean variables $z_1, \ldots, z_n$. Formally, the class is defined as

$$\text{3-TERM-DNF}_n = \{T_1 \vee T_2 \vee T_3 \mid T_i \in \text{CONJUNCTIONS}_n\},$$
$$\text{3-TERM-DNF} = \bigcup_{n \geq 1} \text{3-TERM-DNF}_n.$$

Note that any DNF formula with 3 terms can be expressed as a bit-string of length at most $6n$—there are three terms, each of which is a boolean conjunction expressible by a boolean string of length $2n$; as a result, the representation size for each $c \in \text{3-TERM-DNF}_n$ can be bounded by $6n$. Thus, an efficient algorithm for learning 3-TERM-DNF needs to run in time polynomial in n, $1/\epsilon$ and $1/\delta$. The next result shows that such an algorithm in fact is unlikely to exist. Formally, we'll prove the following theorem.

Theorem 1.6. 3-TERM-DNF *is not efficiently PAC (take II) learnable unless* RP = NP.

Let us first discuss the condition "unless RP = NP". We will briefly define the class RP here, but those unfamiliar with (randomized) complexity classes may wish to refer to standard texts on complexity theory (cf. Chapter Notes in Section 1.8). The class RP consists of languages for which membership can be determined by a randomised polynomial time algorithm that errs on only

[10] We treat this map as purely syntactic. In particular, for truth assignments the order of the variables does not matter; however, for the purpose of the map g, the 2-CNF formulae $(x_1 \vee x_2) \wedge (x_3 \vee x_4)$ and $(x_2 \vee x_1) \wedge (x_4 \vee x_3)$ would be mapped to the (distinct) conjunctions, $z'_{z_1,z_2} \wedge z'_{z_3,z_4}$ and $z'_{z_2,z_1} \wedge z'_{z_4,z_3}$ respectively.

one side. More formally, a language $L \in \mathsf{RP}$, if there exists a randomised polynomial time algorithm A that satisfies the following

- For string $\sigma \notin L$, $A(\sigma) = 0$
- For string $\sigma \in L$, $A(\sigma) = 1$ with probability at least $1/2$.

The rest of this section is devoted to prove Theorem 1.6. We shall reduce an NP-complete language to the problem of PAC (take II) learning 3-TERM-DNF. Suppose L is a language that is NP-complete. Given an instance (string) σ we wish to decide whether $\sigma \in L$. We will construct a training sample, a set of positive instances S_+ and negative instances S_-, where S_+ and S_- are disjoint. We will show that there exists a 3-term DNF formula φ such that all instances in S_+ are satisfying assignments of φ and that none of the instances in S_- satisfy φ, if and only if $\sigma \in L$.

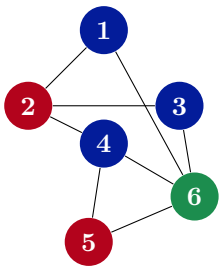
Let us see how an algorithm that PAC (take II) learns 3-TERM-DNF can be used to test whether or not $\sigma \in L$. Let $S = S_+ \cup S_-$, where S_+ and S_- are the sets as constructed above, and let D be a distribution that is uniform over S, i.e. a distribution that assigns probability mass $\frac{1}{|S|}$ to every instance that appears in S, and 0 mass to all other instances. Let $\epsilon = \frac{1}{2|S|}$ and $\delta = 1/2$. Now, let us suppose that $\sigma \in L$, then indeed there does exist 3-term DNF formula, φ, that is consistent with the sample S. So we can simulate a valid example oracle $\mathsf{EX}(\varphi, D)$, by simply returning a random example $(\mathbf{x}, y)$ where $\mathbf{x}$ is chosen uniformly at random from S, and $y = 1$ if $\mathbf{x} \in S_+$, and $y = 0$ otherwise. By the PAC (take II) learning guarantee, with probability at least $1/2$, the algorithm returns $h \in$ 3-TERM-DNF, such that $\mathrm{err}(h) \leq \frac{1}{2|S|}$. However, as there are only $|S|$ instances in S and the distribution is uniform, it must be that h correctly predicts the labels of all instances in S, which implies $\sigma \in L$. Notice that given h, it can easily be checked in polynomial time that h indeed correctly predicts the labels for all instances in S.

On the other hand, if $\sigma \notin L$, there is no 3-term DNF formula that correctly assigns labels to the instances in S. Hence, the learning algorithm cannot output such an $h \in$ 3-TERM-DNF. Again, given the output hypothesis h, checking whether h correctly labels all the instances in S or not, can be easily done in polynomial time. Thus, assuming a PAC (take II) learning algorithm for 3-TERM-DNF exists, we also have a randomised algorithm to solve the decision problem for the NP-complete language L. This in turn implies that $\mathsf{RP} = \mathsf{NP}$, something that is widely believed to be untrue.

All that is left to do is to identify a suitable NP-complete language and show how to construct a sample S with the desired property. In this case, we will use the fact that *graph* 3-*colouring* is NP-complete.

Graph 3-Colouring reduces to PAC (Take II) Learning 3-TERM-DNF

The language 3-COLOURABLE consists of representations of graphs that can be 3-*coloured*. We say a graph is 3-*colourable* if there is an assignment from the vertices to the set of three colours, $\{r, g, b\}$, such that no two adjacent vertices are assigned the same colour. As already discussed, given a graph G, we only need to produce disjoint sets S_+ and S_- of instances that are positively and negatively labelled respectively, such that the graph G is 3-colourable if and



(a) Graph

$v(1)$	$(0,1,1,1,1,1)$
$v(2)$	$(1,0,1,1,1,1)$
$v(3)$	$(1,1,0,1,1,1)$
$v(4)$	$(1,1,1,0,1,1)$
$v(5)$	$(1,1,1,1,0,1)$
$v(6)$	$(1,1,1,1,1,0)$

(b) Positive Examples

$e(\{1,2\})$	$(0,0,1,1,1,1)$
$e(\{1,6\})$	$(0,1,1,1,1,0)$
$e(\{2,3\})$	$(1,0,0,1,1,1)$
$e(\{2,4\})$	$(1,0,1,0,1,1)$
$e(\{3,6\})$	$(1,1,0,1,1,0)$
$e(\{4,5\})$	$(1,1,1,0,0,1)$
$e(\{4,6\})$	$(1,1,1,0,1,0)$
$e(\{5,6\})$	$(1,1,1,1,0,0)$

(c) Negative Examples

Figure 1.3: (a) A graph G along with a valid three colouring. (b) Positive examples of the sample generated using G. (c) Negative examples of the sample generated using G.

only if there exists a 3-term DNF formula that correctly predicts the labels of all instances in $S_+ \cup S_-$.

For notational convenience, in this section, we will denote the instances as $v(i)$ and $e(i,j)$ rather than the more usual $\mathbf{x}$. Suppose G has n vertices. For vertex $i \in G$, we let $v(i) \in \{0,1\}^n$ that has a 1 in every position except i. For an edge (i,j) in G, we let $e(\{i,j\}) \in \{0,1\}^n$ that has a 1 in all positions except i and j. Let $S_+ = \{v(i) \mid i \text{ a vertex of } G\}$ and $S_- = \{e(\{i,j\}) \mid \{i,j\} \text{ an edge of } G\}$; clearly S_+ and S_- are disjoint. Figure 1.3 shows an example of a graph that is 3-colourable along with the sets S_+ and S_-.

First, suppose that G is 3-colourable. Let V_r, V_g, V_b be the set of vertices of G that are labelled red (r), blue (b) and green (g) respectively. Let $z_1, \ldots, z_n$ denote the n boolean variables (one corresponding to each vertex of G). Let $T_r = \bigwedge_{i \notin V_r} z_i$. T_g and T_b are defined similarly. Consider the 3-term DNF formula $\varphi = T_r \vee T_g \vee T_b$; we will show that all instances in S_+ satisfy φ and that none of the instances in S_- do. First consider $v(i) \in S_+$. Without loss of generality, suppose i is coloured red, i.e. $i \in V_r$. Then, we claim that $v(i)$ is a satisfying assignment of T_r and hence also of φ. Clearly, the literal z_i is not contained in T_r and there are no negative literals in T_r. Since all the bits of $v(i)$ other than the i^{th} position are 1, $v(i)$ is a satisfying assignment of T_r. Now, consider $e(\{i,j\})$. We claim that $e(\{i,j\})$ is not a satisfying assignment of any of T_r, T_g or T_b and hence it also does not satisfy φ. For a colour $c \in \{r,g,b\}$, either i is not coloured c or j isn't. Suppose i is the one that is not coloured c, then T_c contains the literal z_i, but the i^{th} bit of $e(\{i,j\})$ is 0 and so $e(\{i,j\})$ is not a satisfying assignment of T_c. This argument applies to all colours and hence $e(\{i,j\})$ is not a satisfying assignment of φ. This completes the "if" part of the proof.

Next, suppose that $\varphi = T_r \vee T_g \vee T_b$ is a 3-term DNF such that all instances in S_+ are satisfying assignments of φ and none in S_- are. We use φ to assign colours to the vertices of G that represent a valid 3-colouring. For a vertex i, since $v(i)$ is a satisfying assignment of φ, it is also a satisfying assignment of at least one of T_r, T_g or T_b. We assign it a colour based on the term for which it is a satisfying assignment (ties may be broken arbitrarily). Since for every vertex i, there exists $v(i) \in S_+$, this ensures that every vertex is assigned a colour.

Next, we need to ensure that no two adjacent vertices are assigned the same colour. Suppose there is an edge $\{i, j\}$ such that i and j are assigned the same colour. Without loss of generality, suppose that this colour is red (r). Since we know that $e(\{i, j\})$ is not a satisfying assignment of φ, $e(\{i, j\})$ also does not satisfy T_r. Also, as i and j were both coloured red, $v(i)$ and $v(j)$ do satisfy T_r. This implies that the literals z_i and z_j are not present in T_r. The fact that $v(i)$ satisfies T_r ensures that the literal $\bar{z}_k$ for any $k \neq i$ cannot appear in T_r. However, if T_r does not contain any negated literal, other than possibly z_i, and if it does not contain the literals z_i and z_j, then $e(\{i, j\})$ satisfies T_r and hence φ, a contradiction. Hence, there cannot be any two adjacent vertices that have been assigned the same colour. This completes the proof of the "only if" part and with it also the proof of Theorem 1.6.

1.5 Learning 3-CNF vs 3-TERM-DNF

In Section 1.3.1, we proved that the concept class k-CNF, and hence 3-CNF, is *efficiently* PAC (take II) learnable. On the other hand, Theorem 1.6 shows that under the widely believed assumption that RP $\neq$ NP, the class 3-TERM-DNF is not *efficiently* PAC (take II) learnable. Let us recall the distributive law of boolean operations

$$(a \wedge b) \vee (c \wedge d) \equiv (a \vee c) \wedge (a \vee d) \wedge (b \vee c) \wedge (b \vee d). \tag{1.5}$$

By applying the rule (1.5), we can express any $\varphi \in$ 3-TERM-DNF as some $\psi \in$ 3-CNF.

$$\varphi = T_1 \vee T_2 \vee T_3 \equiv \bigwedge_{\substack{\ell_1 \in T_1 \\ \ell_2 \in T_2 \\ \ell_3 \in T_3}} (\ell_1 \vee \ell_2 \vee \ell_3) = \psi$$

For any distribution D over $X_n = \{0, 1\}^n$, the example oracles $\mathsf{EX}(\varphi, D)$ and $\mathsf{EX}(\psi, D)$ are indistinguishable. Thus, if we use a PAC (take II) learning algorithm for 3-CNF that outputs some $h \in$ 3-CNF, with probability at least $1 - \delta$, we will have

$$\mathrm{err}(h; \varphi, D) = \mathrm{err}(h; \psi, D) \leq \epsilon.$$

What this suggests is that if our goal is simply to *predict* as well as the target concept $\varphi \in$ 3-TERM-DNF, then there is no impediment (in terms of statistical or computational resources) to doing so. The difficulty arises because our definition of PAC (take II) learning requires us to express the output hypothesis as a 3-term DNF formula. Arguably from the point of view of learning, being able to *predict* labels correctly is more important than the exact hypothesis we use to do so. Our final definition of PAC learning in Section 1.6 will allow learning algorithms to output hypothesis that do not belong to the concept class being learnt. We will still need to put some restrictions on what is allowable as an output hypothesis; you are asked to explore the implications of loosening these requirements further in Exercise 2.3. It may also be the case that the computational savings (being able to run in polynomial time) come at

a statistical cost, something we will explore in greater detail after having seen some general methods of designing learning algorithms.[11]

1.6 PAC Learning

In our final definition of PAC learning, we shall remove the requirement that the output hypothesis actually belongs to the concept class being learnt. We then have to specify in what form an algorithm may output a hypothesis. As was the case with concept classes, we can define a hypothesis class H_n over the instances X_n (implicitly we assume that there is also a representation scheme for H_n and an associated size function), and consider the hypothesis class $H = \bigcup_{n \geq 1} H_n$. We will wish to place some restrictions on the hypothesis class. (To explore why consider Exercise 2.3.) The requirement we add is that the hypothesis class H be *polynomially evaluatable*.

Definition 1.7 – **Polynomially Evaluatable Hypothesis Class.** *A hypothesis class H is* polynomially evaluatable *if there exists an algorithm that on input any instance $\mathbf{x} \in X_n$ and any representation $h \in H_n$, outputs the value $h(\mathbf{x})$ in time polynomial in n and* size(h).

In words, the requirement that H be polynomially evaluatable demands that given the description of the "program" encoding the prediction rule, h, and an instance, $\mathbf{x}$, we should be evaluate $h(\mathbf{x})$ in a reasonable amount of time. Here reasonable means polynomial in the input, i.e. size(h) and $\mathbf{x}$. We now give the final defintion of PAC learning and then end by making a few observations.

Definition 1.8 – **PAC Learning.** *For $n \geq 1$, let C_n be a concept class over instance space X_n and let $C = \bigcup_{n \geq 1} C_n$ and $X = \bigcup_{n \geq 1} X_n$. We say that C is* PAC learnable using hypothesis class H *if there exists an algorithm L that satisfies the following: for every $n \in \mathbb{N}$, for every concept $c \in C_n$, for every distribution D over X_n, for every $0 < \epsilon < 1/2$ and $0 < \delta < 1/2$, if L is given access to $\mathsf{EX}(c, D)$ and inputs n,* size(c)*, ϵ and δ, L outputs $h \in H_n$ that with probability at least $1 - \delta$ satisfies* err$(h) \leq \epsilon$*. The probability is over the random examples drawn from $\mathsf{EX}(c, D)$ as well as any internal randomization of L. The number of calls made to $\mathsf{EX}(c, D)$* (sample complexity) *must be bounded by a polynomial in n,* size(c)*, $\frac{1}{\epsilon}$ and $\frac{1}{\delta}$.*

We further say that C is efficiently *PAC learnable using H, if the running time of L is polynomial in n,* size(c)*, $1/\epsilon$ and $1/\delta$, and if H is polynomially evaluatable.*

Some comments regarding the definition of PAC Learning

i) For efficient PAC learning, although no explicit restriction is put on what size(h) can be, the requirement on the running time of the algorithm ensures that size(h) itself must be bounded by a polynomial in n, size(c), $\frac{1}{\delta}$ and $\frac{1}{\epsilon}$.

[11] The word "may" has been used in the above sentence because the claim is based only on different upper bounds on the sample complexity of *efficient* learning algorithms. No "non-trivial" lower bound on the sample complexity for a polynomial time algorithm for learning 3-TERM-DNF. This will be discussed in greater detail in Chapter 2.

ii) When H is not explicitly specified, by *efficient* PAC learning C, we mean that there exists some polynomially evaluatable hypothesis class H, such that C is *efficiently* PAC learnable using H.

iii) In terms for our final definition of PAC learning, PAC (take II) learning C refers to PAC learning C using C. When *efficiency* is a consideration, the learning algorithm has to be efficient and C itself needs to be polynomially evaluatable. In the literature (and in the rest of this course), (efficient) PAC (take II) learning is referred to as (efficient) *proper* PAC learning. Sometimes to distinguish PAC learning from *proper* PAC learning, the word *improper* is added in front of PAC learning.

iv) In the definition of PAC learning (all of them), we do require that the number of calls to $\mathsf{EX}(c, D)$ is bounded by a polynomial in n, size(c), $\frac{1}{\epsilon}$ and $\frac{1}{\delta}$. This corresponds to the *sample complexity* or the amount of data used by the learning algorithm. Even when we allow *inefficient* algorithms, we do require the amount of data used to be modest; this is mainly to capture the idea that *automated learning* is about learning the target function using a modest amount of data. When arbitrary computational power is permitted, there is not much to be gained from using more data; this follows from Exercise 2.3.

1.7 Exercises

1.1. This question is about the rectangle learning problem.

a) Modify the analysis of the rectangle learning algorithm to work in the case that D is an arbitrary probability distribution over $\mathbb{R}^2$.

b) The concept class of *hyper-rectangles* over $\mathbb{R}^n$ is defined as follows

$$\mathsf{RECTANGLES}_n = \{\mathbb{1}_{[a_1,b_1]\times\cdots\times[a_n,b_n]} \mid a_i, b_i \in \mathbb{R}, a_i < b_i\}.$$

For a set $S \subset \mathbb{R}^n$, the notation $\mathbb{1}_S$ represents its indicator, i.e. the boolean function that is 1 if $x \in S$ and 0 otherwise. Generalise the algorithm for learning rectangles in $\mathbb{R}^2$ and show that it *efficiently* PAC learns the class of hyper-rectangles. Give bounds on the number of examples required to guarantee that with probability at least $1-\delta$, the error of the output hypothesis at most ϵ. The sample complexity and running time of your algorithm should be polynomial in n, $\frac{1}{\epsilon}$ and $\frac{1}{\delta}$.

1.2. Let $f : \{0,1\}^n \to \{0,1\}$ be the parity function on the n bits, i.e. $f = z_1 \oplus z_2 \oplus \cdots \oplus z_n$. In words, when given n bits as input f evaluates to 1 if and only if an odd number of the input bits are 1.

a) A boolean circuit with n inputs is a directed acyclic graph with exactly n source nodes and 1 sink node. The source nodes contain the inputs $z_1, \ldots, z_n$ (at the time of evaluation each z_i is assigned a value in $\{0,1\}$). Each internal node is labelled with either $\wedge$, $\vee$, or $\neg$; internal nodes labelled by $\wedge$ or $\vee$ have in-degree exactly 2 and internal nodes labelled by $\neg$ have in-degree exactly 1. The nodes labelled by $\wedge$, $\vee$ and $\neg$, compute the logical *and*, *or*, and

not, of their inputs (the values at the one or two nodes that feed into them) respectively. The sink node represents the output of the circuit which will be either 0 or 1. The *size* of a boolean circuit is defined to be the number of edges in the directed acyclic graph that represents the circuit; the depth of a circuit is the length of the *longest* path from a source node to the sink node. Show that f can be represented as a boolean circuit of size $O(n)$ and depth $O(\log n)$.

b) Show that representing f in disjunctive normal form (DNF) requires at least 2^{n-1} terms.

1.3. Say that an algorithm L *perhaps learns* a concept class C using hypothesis class H, if for every n, for every concept $c \in C_n$, for every distribution D over X_n and for every $0 < \epsilon < 1/2$, L given access to $\mathsf{EX}(c, D)$ and inputs ϵ and size(c), runs in time polynomial in n, size(c) and $1/\epsilon$, and outputs a polynomially evaluatable hypothesis $h \in H_n$, that with probability at least 3/4 satisfies $\text{err}(h) \leq \epsilon$. In other words, we've set $\delta = 1/4$ in the definition of *efficient* PAC learning. Show that if C is "perhaps learnable" using H, then C is also *efficiently* PAC learnable using H.

1.4. Consider the question of learning boolean threshold functions. Let $X_n = \{0,1\}^n$ and for $w \in \{0,1\}^n$ and $k \in \mathbb{N}$, $f_{w,k} : X_n \to \{0,1\}$ is a boolean threshold function defined as follows:

$$f_{w,k}(\mathbf{x}) = \begin{cases} 1 & \text{if } \sum_{i=1}^{n} w_i \cdot x_i \geq k \\ 0 & \text{otherwise} \end{cases}$$

Define the concept class of threshold functions as

$$\mathsf{THRESHOLDS}_n = \{f_{w,k} \mid w \in \{0,1\}^n, 0 \leq k \leq n\},$$
$$\mathsf{THRESHOLDS} = \bigcup_{n \geq 1} \mathsf{THRESHOLDS}_n.$$

Prove that unless $\mathsf{RP} = \mathsf{NP}$, there is no *efficient proper* PAC learning algorithm for THRESHOLDS.

1.5. Let $X_n = \{0,1\}^n$ be the instance space. A parity function, χ_S, over X_n is defined by some subset $S \subseteq \{1, \ldots, n\}$, and takes the value 1 if and odd number of the input literals in the set $\{x_i \mid i \in S\}$ are 1 and 0 otherwise. For example, if $S = \{1, 3, 4\}$, then the function $\chi_S = x_1 \oplus x_3 \oplus x_4$ computes the parity on the subset $\{x_1, x_3, x_4\}$. Note that any such parity function can be represented by a bit string of length n, by indicating which indices are part of S. Let $\mathsf{PARITIES}_n$ denote the concept class consisting of all 2^n parity functions; observe that the the concept class $\mathsf{PARITIES}_n$ has representation size at most n. Show that the class PARITIES, defined as $\mathsf{PARITIES} = \bigcup_{n \geq 1} \mathsf{PARITIES}_n$, is *efficiently proper* PAC learnable. You should clearly describe a learning algorithm, analyse its running time and prove its correctness.

1.8 Chapter Notes

Material in this lecture is almost entirely adopted from Kearns and Vazirani [6, Chap. 1]. The original PAC learning framework was introduced in a seminal paper by Valiant [8].

While we will not make heavy use of deep results from Computational Complexity theory, acquaintance with basic concepts such as NP-completeness, will be necessary. Better understanding of computational complexity will also be beneficial to understand hardness of learning based on the $\mathsf{RP} \neq \mathsf{NP}$ conjecture and other conjectures from cryptography. The classic text by Papadimitriou [7], and the more recent book by Arora and Barak [1], are excellent resources for students wishing to read up further on computational complexity theory.

Chapter 2

Consistent Learning and Occam's Razor

In the previous chapter, we studied a few different learning algorithms. Both the design and the analysis of those algorithms was somewhat *ad hoc*, based on first principles. In this chapter, we'll begin to develop tools that will serve as general methods to design learning algorithms and analyse their performance.

2.1 Occam's Razor

In the first part of this chapter, we'll study an *explanatory framework* for learning. In the PAC learning framework, what is important is a guarantee that, with high probability, the output hypothesis performs well on unseen data, i.e. data drawn from the target distribution D. Here we consider the following question: Given $(\mathbf{x}_1, y_1), (\mathbf{x}_2, y_2), \ldots, (\mathbf{x}_m, y_m)$, where $\mathbf{x}_i \in X_n$ and $y_i \in \{0, 1\}$, can we find some hypothesis, $h : X_n \to \{0, 1\}$ that is consistent with the observed data, i.e. for all i, $h(x_i) = y_i$.[1]

If there is no restriction on the output hypothesis, then this can be simply achieved by *memorizing* the data. In particular, one could output a program of the form, *"if* $\mathbf{x} = \mathbf{x}_1$*, output* y_1*, else if* $\mathbf{x} = \mathbf{x}_2$*, output* y_2*, ..., else if* $\mathbf{x} = \mathbf{x}_m$*, output* y_m*, else output* 0*"*. This output hypothesis is correct on all of the observed data and predicts 0 on all other instances. Clearly, we would not consider this as a form of learning. The basic problem here is that the "explanation" of the data is as long as the data itself. Even if one tries to rule out programmes of this kind, it is easy to see that simple concept classes are rich enough to *essentially* memorise the data (cf. Exercise 2.1).

The condition that we want to impose is that the explanation of the data be *succinct*, at the very least, shorter than the length of the data itself. In computational learning theory, this is referred to as the *Occam Principle* or *Occam's Razor*, named after the medieval philosopher and theologian, William of Ockham, who expounded the principle that "explanations should be not made unnecessarily complex".[2]

[1] In order to avoid absurdities, we will assume that for all $1 \leq i, j \leq m$, it is not the case that $\mathbf{x}_i = \mathbf{x}_j$, but $y_i \neq y_j$.

[2] This is by no means a wholly accurate depiction of the writings of William of Ockham. Those interested in the history are encouraged to look up the original work.

Philosophical Implications*

The notion of *succinct explanations* can be formalised in several ways and has deep connections to various areas of mathematics and philosophy. There are connections to Kolmogorov complexity which leads to the *minimum description length* (MDL) principle. The MDL principle itself can be given a Bayesian interpretation of assigning a larger prior probability to shorter hypotheses. The existence of a short description also implies existence of compression schemes. We will not discuss these issues in detail in this course; the interested student is referred to the following sources as a starting point [2, 5, 3].

Typically, finding the *shortest hypothesis* consistent with the data may be intractable or even uncomputable. In order to get useful results out of this principle, we do not need to find the shortest description or achieve optimal compression. It turns out that it is enough for the description of the output hypothesis to be slightly shorter than the amount of data observed. We'll formalise this notion to derive PAC-learning algorithms from explanatory hypotheses.

2.2 Consistent Learning

We'll first define the notion of a consistent learning algorithm, or consistent learner, for a concept class C.[3]

Definition 2.1 **– Consistent Learner.** *We say that a learning algorithm L is a consistent learner for a concept class C using hypothesis class H, if for all $n \geq 1$, for all $c \in C_n$ and for all $m \geq 1$, given as input the sequence of examples, $(\mathbf{x}_1, c(\mathbf{x}_1)), (\mathbf{x}_2, c(\mathbf{x}_2)), \ldots, (\mathbf{x}_m, c(\mathbf{x}_m))$, where each $\mathbf{x}_i \in X_n$, L outputs $h \in H_n$ such that for $i = 1, \ldots, m$, $h(\mathbf{x}_i) = c(\mathbf{x}_i)$. We say that L is an* efficient consistent learner *if the running time of L is polynomial in n,* size(c) *and m. Furthermore, we shall say that a concept class C is (efficiently) consistently learnable, if there exists a learning algorithm L and a polynomially-evaluatable hypothesis class H, such that L is an (efficient) consistent learner for C using H.*

A consistent learning algorithm is simply required to output a (polynomially evaluatable) hypothesis that is consistent with all the training data provided to it. So far, we have not imposed any requirement on the hypothesis class H. This notion of *consistency* is closely related to the empirical risk minimisation (ERM) principle in the statistical machine learning literature, where the risk is defined using the *zero-one* loss.

The main result we will prove is that if H is "small enough", something that is made precise in the theorem below, then a consistent learner can be used to derive a PAC-learning algorithm. This theorem shows that short explanatory hypotheses do in fact also possess predictive power.

Theorem 2.2 **– Occam's Razor, Cardinality Version.** *Let C be a concept class and H a hypothesis class. Let L be a consistent learner for C using H.*

[3] Starting from this chapter, we will avoid the cumbersome notation of treating a concept class C as $C = \cup_{n \geq 1} C_n$ (likewise $X = \cup_{n \geq 1} X_n$ and $H = \cup_{n \geq 1} H_n$) and shall assume that this is implicitly the case. Where confusion may arise we shall continue to be fully explicit about concept classes that contain concepts defined over instance spaces of increasing sizes.

Then for all $n \geq 1$, for all $c \in C_n$, for all D over X_n, for all $0 < \epsilon < 1/2$ and all $0 < \delta < 1/2$, if L is given a sample of size m drawn from $\mathsf{EX}(c, D)$, such that,

$$m \geq \frac{1}{\epsilon}\left(\log|H_n| + \log\frac{1}{\delta}\right), \tag{2.1}$$

then L is guaranteed to output a hypothesis $h \in H_n$ that with probability at least $1 - \delta$, satisfies $\mathrm{err}(h) \leq \epsilon$.

If furthermore, L is an efficient consistent learner, $\log|H_n|$ is polynomial in n and $\mathrm{size}(c)$, and H is polynomially evaluatable, then C is efficiently PAC-learnable using H.

Proof. Fix a target concept $c \in C_n$ and the target distribution D over X_n. Call a hypothesis, $h \in H_n$ "bad" if $\mathrm{err}(h) \geq \epsilon$. Let A_h be the event that m independent examples drawn from $\mathsf{EX}(c, D)$ are all consistent with h, i.e. $h(\mathbf{x}_i) = c(\mathbf{x}_i)$, for $i = 1, \ldots, m$. Then, if h is bad, $\mathbb{P}[A_h] \leq (1-\epsilon)^m \leq e^{-\epsilon m}$.

Consider the event,

$$\mathcal{E} = \bigcup_{h \in H_n : h \text{ bad}} A_h$$

Then, by a simple application of the union bound (A.1), we have,

$$\mathbb{P}[\mathcal{E}] \leq \sum_{h \in H_n : h \text{ bad}} \mathbb{P}(A_h) \leq |H_n| \cdot e^{-\epsilon m}$$

Thus, whenever m is larger than the bound given in the statement of the theorem, except with probability δ, no "bad" hypothesis is consistent with m random examples drawn from $\mathsf{EX}(c, D)$. However, any hypothesis that is not "bad", satisfies $\mathrm{err}(h) \leq \epsilon$ as required. □

Remark 2.3. *The version of the theorem described above only allows H_n to depend on C_n and n. It is possible to have a much more general version, where instead we consider the hypothesis class $H_{n,m}$ where a consistent learner when given m examples outputs some $h \in H_{n,m}$. As long as $\log|H_{n,m}|$ can be bounded by polynomial in n, $\mathrm{size}(c)$ and m^β for some $\beta < 1$, a PAC-learning algorithm can still be derived from a consistent learner. Exercise 2.2 asks to you prove this more general result. The proofs for this version appears in the book by Kearns and Vazirani [6, Chap. 2].*

2.3 Improved Sample Complexity

Learning CONJUNCTIONS

Let us revisit some of the learning algorithms we've seen so far. We derived an algorithm for learning conjunctions. At the heart of the algorithm was, in fact, a consistent learner, obtained only using positive examples. Thus, for the conjunction learning algorithm $C_n = H_n$. Note that the number of conjunctions on n literals is 3^n (each variable may appear as a positive literal, negative literal, or not at all).

Our analysis of the conjunction learning algorithm showed that if the number of examples drawn from $\mathsf{EX}(c, D)$ was at least $\frac{2n}{\epsilon}\left(\log(2n) + \log\frac{1}{\delta}\right)$, the output hypothesis with high probability has error at most ϵ. Theorem 2.2 shows that in fact even a sample of size $\frac{1}{\epsilon}\left(n\log 3 + \log\frac{1}{\delta}\right)$ would suffice.

Learning 3-TERM-DNF

Let us now consider the question of learning 3-TERM-DNF. We have shown that finding a 3-term DNF formula φ that is consistent with a given sample is NP-complete. On the other hand, we saw that it is indeed possible to find a 3-CNF formula that is consistent with a given sample. Let us compare the sample complexity bounds given by Theorem 2.2 in both of these cases. In order to do that we need good bounds on $|\text{3-TERM-DNF}|$ and $|\text{3-CNF}|$. Any 3-TERM-DNF formula can be encoded using at most $6n$ bits, each term (or a conjunction) can be represented by a a bit string of length $2n$ to indicate whether a variable appears as a positive literal, negative literal, or not at all. Thus, $|\text{3-TERM-DNF}| \leq 2^{6n}$.

Similarly, there are $(2n)^3$ possible clauses with three literals. Thus, each 3-CNF formula can be represented by a bit string of length $(2n)^3$, indicating for each of the possible clauses whether they are present in the formula or not. Thus, $|\text{3-CNF}| \leq 2^{8n^3}$. It is also not hard to show that $|\text{3-CNF}| \geq 2^{\kappa n^3}$ for some universal constant $\kappa > 0$. Thus, it is the case that $\log|\text{3-CNF}| = \Omega(n^3)$. Thus, in order to use a consistent learner that outputs a 3-CNF formula, we need a sample that has size $\Omega\left(\frac{n^3}{\epsilon}\right)$;[4] on the other hand if we had unbounded computational resources and could solve the NP-complete problem of finding a 3-term DNF consistent with a sample, then a sample of size $O\left(\frac{n}{\epsilon}\right)$ is sufficient to guarantee a hypothesis with error at most ϵ (assuming δ is constant). This suggests that there may be tradeoff between running time and sample complexity. However, it does not rule out that there may be another computationally efficient algorithm for learning 3-TERM-DNF that has a better bound in terms of sample complexity. This question is currently open.

2.4 Exercises

2.1 Given $(\mathbf{x}_1, y_1), (\mathbf{x}_2, y_2), \ldots, (\mathbf{x}_m, y_m)$, such that $\mathbf{x}_i \in X_n$ and $y_i \in \{0, 1\}$, and for all $1 \leq i, j \leq m$, it is not the case that $\mathbf{x}_i = \mathbf{x}_j$, but $y_i \neq y_j$, show that there is a DNF formula of length $O(m)$ that is consistent with the observed data.

2.2 Formulate Remark 2.3 as a precise mathematical statement and prove it. Observe that when $H_{n,m}$ instead of H_n is used in Equation (2.1), m appears on both sides of the equation. You should justify that there exists m that is still polynomial in n, size(c), $\frac{1}{\delta}$ and $\frac{1}{\epsilon}$ that satisfies the modified form of Equation (2.1).

2.3 Recall that in the definition of PAC-learning, we require that the hypothesis output by the learning algorithm be evaluatable in polynomial time. Suppose we relax this restriction, and let H be the class of all Turing machines (not necessarily polynomial time)—so the output of the learning algorithm can be any program. Let C_n be the class of all boolean circuits of size at most $p(n)$ for some fixed polynomial p and having n

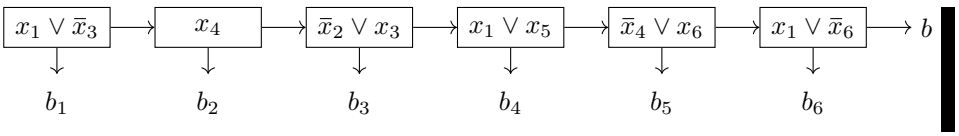
[4] At the very least, this is the lower bound we get if we apply Theorem 2.2. We will see shortly that in fact this is a lower bound on sample complexity for learning 3-CNF, no matter what algorithm is used.

boolean inputs. Show that $C = \bigcup_{n \geq 1} C_n$ is PAC-learnable using H (under this modified definition). Argue that this solution shows that the relaxed definition trivialises the model of learning.

2.4 A k-decision list over n boolean variables $x_1, \ldots, x_n$, is defined by an ordered list

$$L = (c_1, b_1), (c_2, b_2), \ldots, (c_l, b_l),$$

and a bit b, where each c_i is a clause (disjunction) of at most k literals (positive or negative) and each $b_i \in \{0, 1\}$. For $a \in \{0, 1\}^n$ the value $L(a)$ is defined to be b_j, where j is the smallest index satisfying $c_j(a) = 1$ and $L(a) = b$ if no such index exists. Pictorially, a decision list can be depicted as shown below. As we move from left to right, the first time a clause is satisfied, the corresponding b_j is output, if none of the clauses is satisfied the default bit b is output.



Give a consistent learner for the class of decision lists. As a first step, argue that it is enough to just consider the case where all the clauses have length 1, i.e. in fact they are just literals.

Appendix A

Useful Inequalities from Probability Theory

It is assumed that the reader has sufficient familiarity with the basics of the theory of probability.

A.1 The Union Bound

This is an elementary inequality, though surprisingly powerful in several applications in learning theory and the analysis of algorithms. If $A_1, A_2, \ldots$ is a (at most countable) collection of events, then

$$\mathbb{P}\left[\bigcup_i A_i\right] \leq \sum_i \mathbb{P}(A_i). \tag{A.1}$$

This inequality is known as the *union bound* (or Boole's inequality) as it shows that the probability of the union of a collection of events can be upper-bounded by the sum of the probabilities of the individual events in the union.

A.2 Hoeffding's Inequality

Let $X_1, \ldots, X_m$ be m independent random variables taking values in the interval $[0, 1]$. Let $\overline{X} = \frac{1}{m}\sum_{i=1}^{m} X_i$ and let $\mu = \mathbb{E}[\overline{X}]$. Then for every $t \geq 0$,

$$\mathbb{P}\left[\left|\overline{X} - \mu\right| \geq t\right] \leq \exp\left(-2mt^2\right). \tag{A.2}$$

This inequality is known as the Hoeffding's inequality [4].

Appendix B

Useful Inequalities

B.1 Convexity of exp

For any $x \in \mathbb{R}$, the following inequality holds,

$$1 + x \leq e^x. \tag{B.1}$$

The proof is immediate using the convexity of the exponential function.

Appendix C

Notation

C.1 Basic Mathematical Notation

$\mathbb{N}$	The set of natural numbers (not including 0)
$\mathbb{Z}$	The set of integers
$\mathbb{Q}$	The set of rational numbers
$\mathbb{R}$	The set of real numbers

C.2 The PAC Learning Framework

$\mathbf{x}$	A datum or the *input* part of an example
x_i	The i^{th} co-ordinate (attribute) of example $\mathbf{x}$

Bibliography

[1] Sanjeev Arora and Boaz Barak. *Computational Complexity: A Modern Approach*. Cambridge University Press, 2009.

[2] Ronald de Wolf. Philosophical applications of computational learning theory : Chomskyan innateness and occam's razor. Master's thesis, Erasmus Universiteit Rotterdam, 1997.

[3] Peter D Grünwald. *The minimum description length principle*. MIT press, 2007.

[4] Wassily Hoeffding. Probability inequalities for sums of bounded random variables. *Journal of the American Statistical Association*, 58(301):13–30, 1963. ISSN 01621459. URL `http://www.jstor.org/stable/2282952`.

[5] Edwin T Jaynes. *Probability theory: The logic of science*. Cambridge university press, 2003.

[6] Michael J. Kearns and Umesh K. Vazirani. *An Introduction to Computational Learning Theory*. The MIT Press, 1994.

[7] Christos H Papadimitriou. *Computational complexity*. John Wiley and Sons Ltd., 2003.

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

Index