COMS W4252: Intro. Computational Learning Theory Fall 2023

Computational Learning Theory

Prof. Rocco Servedio *Scribe: Ekene Ezeunala*

Contents

§1 Lecture 01—6th September, 2023

§1.1 Introduction

This course is about exploring learning models. There will be two aspects to our CLT life:

1. Developing, defining, and motivating computational learning models (the rules of the computational learning theory game)—1% of the course;
2. Exploring the model, proving things (giving algorithms, hardness results, developing useful theory) about our learning models (learning how to play the game)—99% of the course.

A *learning model* encodes how learning works, that is, the rules of the learning game. When we specify a learning model we implicitly specify whether the learning process is supervised (the data is labelled with the correct classification) or unsupervised (the data is not labelled with the correct classification, and we need to group data points into groups/"clusters"). In this course, we will exclusively focus on supervised learning models: that is, we will assume that the labelled data set $\left\{\left(x^{(i)}, f\left(x^{(i)}\right)\right)\right\}_{i=1}^{n}$ is given to the learner as input, and the learner (really, the learning algorithm) tries to infer f. Our focus in the course will be exclusively restricted to learning (binary) classification rules, that is, an unknown Boolean function $f : \{0,1\}^n \to \{0,1\}$. This is often called *concept learning* and the Boolean functions to be learned are called *concepts*. A few questions spring up immediately:

1. *How is information available to the learner?* There are multiple ways by which the learner learns:
   - Passive models—models in which the learner is given examples, but doesn't control which examples it gets—and active models—models in which the learner can make queries of various forms about the examples before actually learning. Some of the queries we may potentially be interested in are (1) membership queries ("Does an example belong to the class or not?") and (2) equivalence queries ("Is the hypothesis that I have learned equivalent to the target concept?").
   - Models with a teacher—perhaps the teacher could be entirely helpful, slightly uninformed, or even adversarial.
   - Learning from noisy information—sometimes the information fed to the learner is not 100% good or maybe only partially correct.
2. *What prior knowledge does the learner have?* Without any regularity or structure, it is very hard to learn, so we need assumptions on the "structure" of the thing we're learning. One big-picture assumption is that we should only try to learn things that are somewhat "simple," by this we mean that we will generally assume that the unknown target concept/Boolean function f has a prespecified syntactic representation (e.g. a linear threshold function, an AND of input variables, etc).
3. *What are the constraints on the learner?* We will be concerned with efficient, polynomial-time algorithms that use poly(n) amount of data.
4. *How do we measure our success?* Our performance criteria will vary across the different learning models we will use:

- In online learning models, we measure performance as the learner learns. In offline learning models, we won't.
- For some models, we will be concerned with the form of the hypothesis that the learner guesses.
- Sometimes we will also be concerned with accuracy, perhaps statistical accuracy, or maybe the number of mistakes the learner makes for each guess.
- Every time we will worry about the computational efficiency of the learning algorithm. This will be closely related to the constraints imposed on the learner.
- Sometimes we will be concerned with the data efficiency of the learning algorithm, that is, the number of examples the learner needs to see before it can learn.

Some of the models we will explore in this course are the following:

- Online Learning with a Mistake Bound (the OLMB model).
- Probably Approximately Correct Learning (the PAC model).
- Statistical Query Learning (the SQ model).
- Exact Learning with Membership Queries (the EQ model).

As the course progresses, we will examine

- specific learning models for specific problems,
- general techniques for designing learning algorithms,
- how much data we need to learn a concept,
- computational barriers to learning (P vs NP, cryptographic barriers) and how to circumvent them,
- learning from noise in different settings,
- boosting accuracy of weak learners, and
- compare and contrast different learning models.

§1.2 Basic notions and terminology

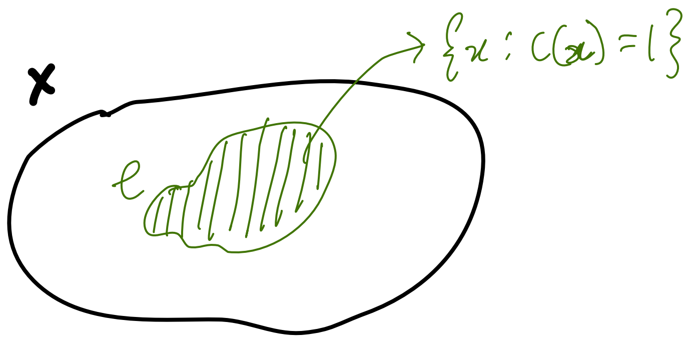
Across the course, we will refer to X as the domain of the functions we will be learning; we will also sometimes refer to X as an instance space, a set of encodings of instances or objects in the learner's world.

Example 1.1.

1. If we are learning classifiers for sports cars, then X could be the set of all possible sports cars.
2. For us, we will be considering Boolean functions, so $X = \{0,1\}^n$, where n is the dimension of our instance space. Sometimes we will also consider $X = \{-1,1\}^n$ and, more generally, $X = \mathbb{R}^n$.

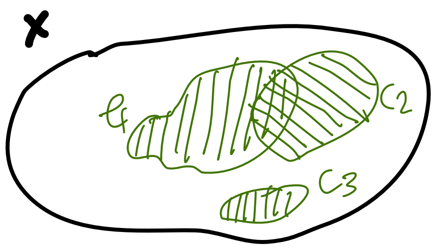
Definition 1.2 (Concept)**.** *A concept is a subset $c \subseteq X$ of the instance space, or equivalently, a Boolean function $c : X \to \{0, 1\}$.*

Here's an example of a concept:



Definition 1.3 (Concept class)**.** *A concept class $\mathcal{C}$ is a set of concepts c over the same X.*

Here's an example of a concept class:



Here, the concept class $\mathcal{C}$ is the set of the three concepts, $\{c_1, c_2, c_3\}$.

Here's the basic idea of our learning models:

1. We have a known instance space X, as well as a known concept class $\mathcal{C}$.
2. There is an unknown target concept $c \in \mathcal{C}$.
3. We (the learner) get some source of information about how c labels various x's in X.
4. From this information, we wish to find (exactly or approximately) what c is.

Example 1.4.
- Let X be the instance space of all animals. Then an example concept class could be $\mathcal{C} = \{\text{elephants, mammals, frogs, whales}\}$.
- Let $X = \{0, 1\}^n$ be the instance space of all n-bit strings. Then an example concept class could be $\mathcal{C} = \{\text{all Boolean conjunctions over } X\}$. A Boolean literal is a bit or its negation, x_i or $\overline{x_i}$, where $x_i \in \{0, 1\}$. A Boolean conjunction is an AND of literals, e.g. $c(x) = x_1 \wedge \overline{x_3} \wedge x_5 \wedge x_6 \wedge \overline{x_{11}}$.

Remarks 1.5.
1. *There are 3^n conjunctions over n literals—for each $1 \leqslant i \leqslant n$, either x_i, $\overline{x_i}$, or neither appears in the conjunction. Thus, the concept class $\mathcal{C}$ above is finite for a finite selection of n.*

2. *Let us say that a conjunction is monotone if it does not contain any negated literals. Then there are 2^n monotone conjunctions over n literals—either x_i or it is absent. Thus, the concept class $\mathcal{C}$ above is finite for a finite selection of n.*

A disjunction is an OR of literals, e.g. $c(x) = x_1 \vee \overline{x_3} \vee x_5 \vee x_6 \vee \overline{x_{11}}$. We can easily check that the concept class of Boolean disjunctions (resp. monotone Boolean disjunctions) is exactly the same in cardinality as the concept class of Boolean conjunctions (resp. monotone Boolean conjunctions).

Next time, we will see more example classes, introduce the OLMB learning model, and come up with algorithms for particular $\mathcal{C}$'s in the OLMB model.

§2 Lecture 02—11th September, 2023

Last time.

- Introduced general learning scenario: learner knows X (instance space), and $\mathcal{C}$ (concept class—sets of concepts over X, or sets of sets of instances in X); the goal is to learn an unknown target concept $c \in \mathcal{C}$.

Today.

- A few more examples of concept classes $\mathcal{C}$ (DNFs, CNFs, LTFs).
- A first learning model: Online Learning with a Mistake Bound (OLMB).
- The elimination algorithm for monotone disjunctions, and some extensions.
- Decision lists; an OLMB algorithm for learning them.

§2.1 Some concept classes of interest

Example 2.1. 1. *Let $X = \{0,1\}^n$ and $\mathcal{C} = \{$all s-term DNF formulas$\}$.* An s-term DNF (disjunctive normal form) is an s-way disjunction of a conjunction of literals. For example, consider the following 3-term DNF over $n = 8$ variables: $c(x) = (x_1 \wedge x_2 \wedge x_3) \vee (\overline{x_1} \wedge x_4 \wedge x_5) \vee (x_6 \wedge \overline{x_7} \wedge x_8)$.

Similarly, an s-clause CNF (conjunctive normal form) is an s-way conjunction of a disjunction of literals.

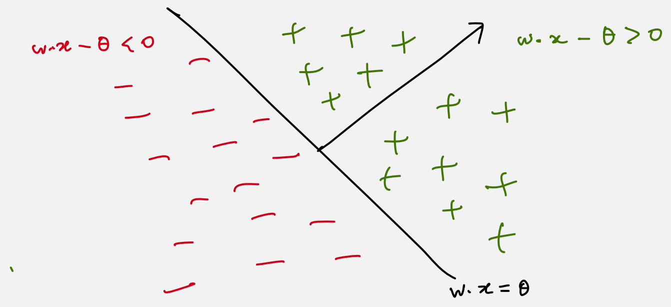
A way to bound the complexity of a CNF or DNF is to restrict the number of the literals at the lowest level of the hierarchy. One way to bound this is the use of k-DNFs. Over $X = \{0,1\}^n$, a k-DNF is an OR of k-literal ANDs—there is no bound on the number (or the fan-in) of ORs, but each AND has at most k literals. Similarly, a k-CNF is an AND of k-literal ORs.

2. *Let $X = \mathbb{R}^n$ and $\mathcal{C} = \{$all linear threshold functions (LTFs) or halfspaces$\}$.* We will first define what a linear threshold function (LTF) is. A function $c : \mathbb{R}^n \to \{-1, +1\}$ is an LTF if there are real weights $w_1, \ldots, w_n \in \mathbb{R}$ and a real threshold $\theta \in \mathbb{R}$ such that

$c(x) = \text{sign}(w \cdot x - \theta)$, where

$$\text{sign}(t) = \begin{cases} +1 & \text{if } t \geqslant 0 \\ -1 & \text{if } t < 0 \end{cases}$$

Here's an LTF classifying points $\mathbb{R}^2$ into positive and negative points:



Note that we can also write the halfspace as $c(x) = \mathbb{1}[w \cdot x \geqslant \theta]$. Note that the concept $c(x) = \text{sign}(100x_1 - 7x_2 + 8x_3 - 16)$ is a halfspace—it cuts euclidean space $\mathbb{R}^2$ in some way according to the weights $w = \begin{bmatrix} 100 & -7 & 8 \end{bmatrix}^\top$ and the threshold $\theta = 16$.

We can also consider LTFs over the Boolean hypercube $\{0,1\}^n$. In this case, we can classify vertices of the Boolean hypercube according to some rules on the bitstring defined over $\{0,1\}^n$.

Remark 2.2. *As we will come to see, our central measure of good algorithms in the OLMB model will be whether or not they make a finite number of mistakes or whether or not they make an arbitrarily large number of mistakes in the worst case.*

§2.2 Our First Learning Model: Online Mistake-Bounded Learning

In the *online mistake-bounded learning model* [Blu05], a learning sequence consists of a sequence of *trials*. Throughout, the learner maintains some hypothesis (or a "guess") $h : X \to \{0,1\}$ of the target concept $c : X \to \{0,1\}$. We will play the learning game like this:

1. For each trial:
    - The learner receives an unlabelled example $x \in X$.
    - The learner outputs a prediction $h(x) \in \{0,1\}$.
    - The learner receives the true label $c(x) \in \{0,1\}$:
        - If $h(x) = c(x)$, then the learner is right, and it does not update its hypothesis.
        - If $h(x) \neq c(x)$, then the learner is wrong, it is charged a mistake, and it updates its hypothesis via some update rule before the next trial.

The update rule along with the initial hypothesis h_0 defines the learning algorithm. We will measure our performance by counting the worst-case total number of mistakes we can possibly make.

Definition 2.3 (Mistake bound)**.** *An online learning algorithm A has mistake bound M for $\mathcal{C}$ if for any arbitrarily long sequence of examples from X, along with $c \in \mathcal{C}$, A makes at most M mistakes when learning c.*

Remarks 2.4.
1. *If $|X| < \infty$, then for any $\mathcal{C}$ over X, we can achieve a mistake bound $m \leqslant |X|$ by simply memorizing all the examples and their labels. This is not very interesting.*
2. *If $|\mathcal{C}| < \infty$, then we can achieve a mistake bound of $|\mathcal{C}| - 1$ by simply trying out every single concept until you eventually find the right one. This is also not very interesting.*

Example 2.5.
1. (Learning initial intervals.) Let the instance space be given by $X = \{0, 1, \dots, 2^n - 1\}$ and let $\mathcal{C}$ be the concept class of all initial intervals over X. That is, a concept $c \in \mathcal{C}$ takes the form $c = \{0, 1, 2, \dots, a\}$ where $a \in X$. We can achieve a mistake bound of n by using the binary search algorithm. Note that we just need to find some terminal point a for each initial interval for each c. Start with an initial hypothesis $h_0 = \{0, 1, \dots, 2^{n-1}\}$ which is the midpoint of the interval, and perform updates like you would with regular binary search.
2. (Learning initial intervals over $\mathbb{R}$.) Let the continuous instance space be $X = [0, 1]$, and $\mathcal{C}$ be the concept class of all initial intervals over $\mathbb{R}$. Note that we cannot achieve a finite mistake bound over $\mathbb{R}$ here, since the reals are dense in $\mathbb{N}$.

§2.3 The elimination algorithm for learning monotone disjunctions

Definition 2.6 (Monotone disjunction)**.** *A monotone disjunction is a concept $c : \{0,1\}^n \to \{0,1\}$ of the form $c(x) = \bigvee_{i \in S} x_i$, where $S \subseteq [n]$.*

We will now present a learning algorithm for these monotone disjunctions. It is called the *elimination algorithm*.

Algorithm (Elimination Algorithm):

1. Set the initial hypothesis $h(x) = x_1 \vee \ldots \vee x_n$.
2. Get an example $z \in \{0,1\}^n$, and predict $h(z)$.
3. Given the true label $c(z)$, update the hypothesis h as follows:
   - (False positive mistake.) If $h(z) = 1$ and $c(z) = 0$, remove x_i from h for all i such that $z_i = 1$.
   - (False negative mistake.) If $h(z) = 0$ and $c(z) = 1$, stop and fail.
   - (Correct hypothesis.) If $h(z) = c(z)$, then keep h the same.

Example 2.7. For example, if $n = 5$, we start with the initial hypothesis $h(x) = x_1 \vee x_2 \vee x_3 \vee x_4 \vee x_5$. If we get the example $z = 01001$, we predict $h(z) = 1$. Suppose however that the true label is $c(z) = 0$. Then we remove x_2 and x_5 from h, and we get $h(x) = x_1 \vee x_3 \vee x_4$.

Let's reason about why this algorithm "makes sense." Suppose that c is a monotone disjunction of the form $c(x) = \bigvee_{i \in S} x_i$. Then if h is a correct hypothesis, then $h(x) = c(x)$ for all $x \in \{0,1\}^n$. In particular, $h(z) = c(z)$ for all $z \in \{0,1\}^n$. So if $h(z) = c(z)$, then we do not need to update h at all. If $h(z) = 1$ and $c(z) = 0$, then we know that we have some $z_i = 1$ in the disjunction that is "hindering" z from being classified as 0; we want to progressively remove this z_i from h. On the other hand, if $h(z) = 0$ and $c(z) = 1$, then we know that $z_i = 0$ for all $i \in S$. So we cannot remove any x_i from h for all $i \in S$, and so we fail—but by means of two claims, we will show that this never actually happens:

Claim 2.8. *If x_i is in c, then x_i is never removed from h.*

Proof. The only way that an x_i can be removed is if an example z has $z_i = 1$ and $c(z) = 0$. But if x_i is in c, then $c(z) = 1$, and so x_i is never removed from h. □

Claim 2.9. *The elimination algorithm only makes false positive mistakes—it never fails.*

Proof. By the previous claim, h always includes all the variables in c. So on example z, if $c(z) = 1$ because of $z_i = 1$, z_i also ensures that $h(z) = 1 \neq 0$. So there are no false negative mistakes. □

We can now prove the mistake bound of the elimination algorithm.

Theorem 2.1. *The elimination algorithm has mistake bound n for the concept class $\mathcal{C}$ of monotone disjunctions over $\{0,1\}^n$.*

Proof. We know that the elimination algorithm doesn't fail. Furthermore, variables in h are a superset of the variables in c. Every mistake removes at least one variable from h. But h initially had n variables, so the algorithm can make at most $n \cdot 1 = n$ mistakes. □

Remarks 2.10. *A few remarks about the elimination algorithm:*

1. *Note that we can exhaustively describe any $c \in \mathcal{C}$ with n bits, where n is the size of the input $x \in \{0,1\}^n$.*
2. *The elimination algorithm is efficient, despite its simplicity. It runs in time $\mathcal{O}(n)$ per trial (since we need to check each bit of the sample), and it makes at most m mistakes.*
3. *The elimination algorithm is extremely susceptible to noise: there is no way to recover from eliminating variables based on corrupted examples. Fortunately, the OLMB model is not concerned with noise.*
4. *There is a natural extension of the elimination algorithm to the class of all disjunctions over $\{0,1\}^n$—it suffices to change the initial hypothesis to $h(x) = x_1 \vee \overline{x_1} \vee \ldots \vee x_n \vee \overline{x_n}$. So for example, suppose the example $z = 10100$ has the label $c(z) = 0$. Then we will remove $x_1, \overline{x_2}, x_3, \overline{x_4}, \overline{x_5}$ from h, since all of these evaluate to 1. The updated hypothesis is then $h(x) = \overline{x_1} \vee x_2 \vee \overline{x_3} \vee x_4 \vee x_5$.*

   Note that after one mistake, we immediately start learning monotone disjunctions "in disguise."

§2.4 Decision lists

Definition 2.11 (Decision lists)**.** *A 1-decision list (1-DL) is a concept $c : X \subseteq \{0,1\}^n \to \{0,1\}$ which is an ordered list of if-then-else rules:*

$$\begin{array}{rl} \mathcal{C} = & \text{“if } \ell_1 \text{ is true, then output } b_1\text{,”} \\ & \text{“else if } \ell_2 \text{ is true, then output } b_2\text{,”} \\ & \vdots \\ & \text{“else if } \ell_r \text{ is true, then output } b_r\text{,”} \\ & \text{“else output } b_{r+1}\text{.”} \end{array}$$

We will usually represent a 1-DL diagrammatically like this:

$$\begin{tikzcd}
x_7 \arrow[r, "0"] \arrow[d, "1"'] & \overline{x_3} \arrow[r, "0"] \arrow[d, "1"'] & \overline{x_5} \arrow[r, "0"] \arrow[d, "1"'] & x_2 \arrow[r, "0"] \arrow[d, "1"'] & 1 \\
0 & 0 & 1 & 0 &
\end{tikzcd}$$

There are $r = 4$ rules in this 1-DL.

Remarks 2.12. *We can say a few things about 1-DLs:*

1. *We never need to repeat literals or variables in a 1-decision list; if we do so, the repeated literals or variables are redundant—the rule we wanted them to enact has been enacted anyway.*
2. *Any conjunction or disjunction is expressible as a decision list. For example, the decision list above is equivalent to the conjunction $\overline{x_7} \wedge x_3 \wedge x_5 \wedge \overline{x_2}$.*
   Indeed, $\mathcal{C}_{\text{conjunctions}} \subseteq \mathcal{C}_{1-\text{DLs}}$.
3. *The total number of 1-DLs of length r is approximately equal to $2^{r+1} \cdot (2n)^r \approx (4n)^r$. This is because there are 2^{r+1} ways to choose the r literals, and for each literal, there are $2n$ ways to choose the value of the literal.*
4. *The total number of possible rules in a 1-DL, including the two “default rules,” is $4n + 2$. This is because there are $2n$ ways to choose the literal, and for each literal, there are two possible values (plus the two default rules).*

§3 Lecture 03—13th September, 2023

Last time.

- Discussed s-term DNFs, s-clause CNFs, k-DNFs, and k-CNFs.
- Discussed LTFs or halfspaces: linear separators of the form $c(x) = \text{sign}(w \cdot x - \theta)$.
- Introduced the Online Learning with a Mistake Bound (OLMB) model.
- Presented the elimination algorithm for learning monotone disjunctions.
- Introduced 1-decision lists.

Today. More examples of OLMB algorithms for specific concept classes $\mathcal{C}$:

- An $\mathcal{O}(nr)$ mistake bound algorithm for learning a 1-DL of length r over $\{0,1\}^n$ (mistake bound $\mathcal{O}(n^2)$ for *any* 1-DL).
- Winnow1 algorithm for learning sparse monotone disjunctions—has a $\mathcal{O}(k \cdot \log n)$ mistake bound for k-sparse monotone disjunctions over $\{0,1\}^n$.

§3.1 Learning decision lists

Recall that the total number of possible rules for a decision list (including the two default rules) is $4n + 2$. We will now present an algorithm for learning 1-decision lists of length r over $\{0,1\}^n$ with mistake bound $\mathcal{O}(nr)$. The hypothesis of this algorithm will be a 1-DL, with "structure":

- the rules in the decision list are partitioned into layers 1, 2, 3, and so on,
- each of the $4n + 2$ rules belong to one of the levels,
- within each level, the rules are lexicographically ordered.
- all rules of a lower level come before rules of a higher level.

Here's the algorithm [Riv87].

Algorithm (Decision-List Learning Algorithm):

1. The initial hypothesis has all the $4n + 2$ rules in the first level, in lexicographic order:

$$\begin{tikzcd}
x_1 \arrow[r, "0"] \arrow[d, "1"'] & x_1 \arrow[r, "0"] \arrow[d, "1"'] & \overline{x_1} \arrow[r, "0"] \arrow[d, "1"'] & \overline{x_1} \arrow[r, "0"] \arrow[d, "1"'] & \cdots \\
0 & 1 & 0 & 1 &
\end{tikzcd}$$

2. After example x:
   - If $h(x) = c(x)$, then keep h the same; no updates.
   - If $h(x) \neq c(x)$, then take the rule that was used to make the incorrect prediction on x, and move it down one level.

Example 3.1. Suppose we are learning a 1-DL via the above hypothesis and we have the following example: $x = 101$ such that $c(x) = 1$. Now the initial hypothesis h misclassifies x, since it says we should output a 0 when we see a 1. Therefore we will move this rule to level 2 after we make this mistake.

This algorithm gives us the following mistake bound:

Theorem 3.1. *The decision-list learning algorithm makes at most $(4n+2)(r+1) = \mathcal{O}(nr)$ mistakes on any 1-DL of length r over $\{0,1\}^n$.*

Proof. We will make a couple of observations and the fact of the theorem will become clear:

1. *The first rule in the target concept c (call it r_1) is never moved below the first level.* This is true because if its literal ℓ_1 holds and it outputs b_1, then b_1 is exactly the $c(x)$ value.
2. *The second rule in the target concept c (call it r_2) is never moved below the second level.* This is true because if h classifies x based on r_2 while r_2 is at level 2, then r_1 must remain at level 1 by the previous observation, and thus x violates r_1's condition, and h agrees with c since they both classify x based on r_2.
3. *Proceeding inductively, the ith rule in c is never moved below the ith level.* So no rule in c is ever moved below level $r+1$, including the default rule b_{r+1}. Hence, no rule at all ever goes below $r+2$ (because b_{r+1} is in the level at most $r+1$).
4. *Key observation.* Each mistake moves a rule down a level. There are $4n+2$ rules, each of which moves $\leqslant r+1$ levels. So there are at most $(4n+2)(r+1)$ mistakes, and this is $\mathcal{O}(nr)$ asymptotically.

The proof is done. □

Remark 3.2. *This is computationally efficient—each stage takes $\mathcal{O}(n)$ time!*

§3.2 Learning sparse disjunctions: Winnow1

Recall that the elimination algorithm over $\{0,1\}^n$ has a mistake bound of n for monotone disjunctions. We said that this is an okay result, but in fact, it's not necessarily good; consider the case where $n = 10^6$ and the target disjunction is of length $k = 5$. Things are not that great—we can make almost 10^6 mistakes! We will improve on this via the Winnow1 algorithm, an algorithm with mistake bound $\mathcal{O}(k \cdot \log n)$ for k-sparse monotone disjunctions over $\{0,1\}^n$. Although we are learning disjunctions, this algorithm—Winnow1—will use a linear threshold function $h(x) = \mathbb{1}\{w \cdot x \geqslant \theta\}$ for a given threshold θ and weight vector $w \in \mathbb{R}^n$. This makes sense because a lot of the Boolean functions we will see are actually equivalent to linear threshold functions; for instance,

1. The disjunction OR can be represented by $h(x) = \mathbb{1}\{x_1 + x_2 + \ldots + x_n \geqslant \frac{1}{2}\}$.
2. The conjunction AND can be represented by $h(x) = \mathbb{1}\{x_1 + x_2 + \ldots + x_n \geqslant n - \frac{1}{2}\}$.
3. The majority function MAJORITY can be represented by $h(x) = \mathbb{1}\{x_1 + x_2 + \ldots + x_n \geqslant \frac{n}{2}\}$.

As an aside, we can express any 1-DL that outputs -1 or $+1$ as a linear threshold function via the following procedure:

1. Construct a term for the head of the 1-DL by multiplying the output value by the variable at the head by $2n+1$.
2. Construct a similar term for the chid of the head of the list, except that final factor is $2n$.
3. Construct a similar term for the child of the child of the head of the list, except that final factor is $2n-1$. Continue in this fashion until a term is created for all the nodes in the 1-DL.
4. Add these terms together and add the final output value of the 1-DL.
5. The linear threshold function is the sign function of the resulting expression.

However, we cannot express any 1-DL as a MAJORITY Boolean function. But we digress; here's the Winnow1 algorithm [Lit88].

Algorithm (Winnow1):

1. Initial hypothesis: $h(x) = \mathbb{1}\{x_1 + \ldots + x_n \geqslant n\}$, initial weights $w_i = 1$ for all i, fixed threshold $\theta = n$.
2. (Updates.) For each example x:
   - If $h(x) = c(x)$, then keep w the same; no updates.
   - (Demotion for false positive.) If $h(x) = 1$ and $c(x) = 0$, then for each i such that $x_i = 1$, set $w_i \leftarrow 0$.
   - (Promotion for false negative.) If $h(x) = 0$ and $c(x) = 1$, then for each i such that $x_i = 1$, set $w_i \leftarrow 2w_i$.

Before we analyse this algorithm, let's see why it makes sense. If the hypothesis is correct for a given example, obviously we don't fix it. If we have a false positive, then $w \cdot x > n$; maybe it's reasonable to just delete the weights that are making this happen. If we have a false negative, then $w \cdot x < n$ but it should have been $\geqslant n$; maybe it's reasonable to just double the weights that are preventing this from happening.

Lemma 3.3. *1. No weight w_i is ever negative.*

2. In each promotion step, at least one variable $x_i \in c$ will get promoted (i.e., w_i will be doubled).

3. For every $i = 1, \ldots, n$, we will always have $w_i \leqslant 2n$.

Proof. The first two are obvious; let's focus on the third instead. Note that w_i is only promoted if $w_i < n$ (otherwise $w_i \geqslant n$ and on $x_i = 1$, we have $w \cdot x \geqslant n$ and no false negative mistake), so it is never $\geqslant 2n$. □

Lemma 3.4. *The total number of promotions (for false negatives) is at most $k \cdot \log 2n$.*

Proof. Observe that no $x_i \in c$ ever gets demoted (demotion only happens if no variable in c is 1).

So for an $x_i \in c$, w_i looks like

$$\underbrace{1,\dots,1,\dots,2,\dots,4,\dots,4,\dots,8,\dots 8,\dots 2^{\ell},\dots,2^{\ell},\dots,2^{\ell+1}-2}_{\leqslant \log 2n \text{ times when it is promoted}}$$

By this and the second part of Lemma 3.3, the total number of promotions is at most $k \cdot \log 2n$. $\square$

Lemma 3.5. *Let d be the number of demotion steps and p the number of promotion steps. Then $d \leqslant p+1$.*

Proof. Let $w = w_1 + \ldots + w_n$. Initially, $w = n$ (since $w_i = 1$ for all i). At each demotion step on example x,

$$\begin{aligned} w \cdot x = w_1x_1 + \ldots + w_nx_n &\text{ was } \geqslant n \\ = \sum_{i:x_i=1} w_i, & \end{aligned}$$

and these w_i are set to 0, so that w decreases by $\geqslant n$. At each promotion step on example x,

$$\begin{aligned} w \cdot x = w_1x_1 + \ldots + w_nx_n &\text{ was } < n \\ = \sum_{i:x_i=1} w_i, & \end{aligned}$$

and these w_i are doubled, so that w increases by $< n$. So w decreases by $\geqslant n$ at each demotion step and increases by $< n$ at each promotion step. Recalling that $w \geqslant 0$ always, we see that

$$0 \leqslant w \leqslant n + pn - dn \implies d \leqslant p+1. \qquad \square$$

The theorem is then a matter of combining lemmas.

Theorem 3.2. *The Winnow1 algorithm makes at most $2k \cdot \log 2n + 1 = \mathcal{O}(k \cdot \log n)$ mistakes on any k-sparse monotone disjunction over $\{0,1\}^n$.*

Proof. By Lemmas 3.3 and 3.4, the total number of promotions is at most $2k \cdot \log 2n$. By Lemma 3.4, the total number of demotions is at most $2k \cdot \log 2n + 1$. So the total number of mistakes is at most $2k \cdot \log 2n + 1 = \mathcal{O}(k \cdot \log n)$ many mistakes when learning length k monotone disjunctions. $\square$

Remarks 3.6. 1. *This is computationally efficient!* 🙂

2. *This is attribute efficient—$\log n$ depends on n!* 🙂

Open Problem 1. *Is there an algorithm that is both computationally efficient and attribute efficient for length-k decision lists?*

§4 Lecture 04—18th September, 2023

Last time.

- Presented an $\mathcal{O}(rn)$ mistake bound algorithm for length r 1-decision lists over $\{0,1\}^n$.
- Presented the Winnow1 algorithm for learning k-sparse monotone disjunctions over $\{0,1\}^n$ with mistake bound $\mathcal{O}(k \cdot \log n)$.

Today. Algorithms for LTFs:

- Winnow2, an algorithm for certain LTFs with positive weights over $\{0,1\}^n$.
- Perceptron, an algorithm for LTFs with a margin over $\mathbb{R}^n$.

§4.1 Winnow2

Remember that in Winnow1 we used an LTF as the hypothesis to learn monotone disjunctions, which are simpler than LTFs. But can we learn LTFs themselves (which are a richer class of functions) efficiently? Let's look at where Winnow1 comes short; recall that Winnow1:

- sometimes set weights to 0. But for any LTF, every variable has some influence, so we may not want to do this, as it is too extreme for general LTFs.
- only gives weights like 2^i, but not all LTF weights are of that form—indeed this is too coarse for some LTFs. Consider the LTF $h(x) = \mathbb{1}\{x_1 + 2x_2 + \ldots + nx_n \geqslant n\}$ where $x_i \in \{0,1\}^n$; this granuarity cannot be captured by Winnow1, and so it will never be able to learn this LTF. ☹

With these issues in mind, we will soon introduce Winnow2. First, let's describe the subclass of LTFs that Winnow2 will be able to learn by means of the following definition.

Definition 4.1 (δ-separable monotone LTFs)**.** *Fix some separability parameter $0 < \delta < 1$. Define $F(\delta)$, the class of δ-separable monotone LTFs over $\{0,1\}^n$ as follows: $c \in F(\delta)$ if and only if there exist weights $u_i \geqslant 0$ such that for all $x \in \{0,1\}^n$, we have*

$$c(x) = \begin{cases} 1 & \textit{if and only if } u \cdot x \geqslant 1, \\ 0 & \textit{if and only if } u \cdot x \leqslant 1 - \delta. \end{cases}$$

The δ of "δ-separable" is encoded in the second requirement above—there is a separation (or an ℓ_1-margin) between positive and negative examples. Intuitively, if δ is not too small, then there is some "noticeable" difference between the positive and negative examples, and so we can learn the LTF. If δ is too small, then the positive and negative examples are too close together, and so it is much harder to learn the LTF. We will see that Winnow2 can learn $F(\delta)$ for any $\delta > 0$.

Example 4.2 (r-out-of-k linear threshold function)**.** Consider the r-out-of-k linear threshold function $h(x) = \mathbb{1}\{x_{i_1} + \ldots + x_{i_k} \geqslant r\}$ over $\{0,1\}^k$. As long as at least r of the variables are set to 1, the function fires; otherwise it doesn't. Note that $c(x) = 1$ if and only if $x_{i_1} + \ldots + x_{i_k} \geqslant r$, which is equivalent to

$$\frac{1}{r} \cdot (x_{i_1} + \ldots + x_{i_k}) \geqslant 1.$$

If we have that $x_{i_j} = 1$ for all the $j = 1, \ldots, k$, then the LTF fires. If we have that $x_{i_j} = 0$ for any of the $j = 1, \ldots, k$, then at most $r - 1$ of them satisfy $x_{i_j} = 1$, and so the left hand side is at most $1 - \frac{1}{r} < 1$. Therefore, if we pick $u = (1, \ldots, 1)$, then

$$c(x) = \begin{cases} 1 & \text{if and only if } u \cdot x \geqslant 1, \\ 0 & \text{if and only if } u \cdot x \leqslant 1 - \frac{1}{r}, \end{cases}$$

and hence $c \in F(\frac{1}{r})$.

Example 4.3. Consider the LTF concept $c(x) = \mathbb{1}\{x_1 + 2x_2 + \ldots + nx_n \geqslant n\}$. Rescaling by $1/n$, this is the same as

$$c(x) = \mathbb{1}\left\{\frac{1}{n}\sum_{i=1}^{n} ix_i \geqslant 1\right\},$$

with $x_i \in \{0, 1\}$. Consider now the cases when one of the $x_i = 0$. In the best case, we are left with $\frac{1}{n}(n-1) = 1 - \frac{1}{n}$ in the left-hand side of the inequality (other cases are $\frac{1}{n}(n-2)$, $\frac{1}{n}(n-3)$, etc), which corresponds to setting x_1—the variable with the lowest weight—to 0. Therefore, if we pick $u = (1, 2, \ldots, n)$, then we see that

$$c(x) = \begin{cases} 1 & \text{if and only if } u \cdot x \geqslant 1, \\ 0 & \text{if and only if } u \cdot x \leqslant 1 - \frac{1}{n}, \end{cases}$$

and hence $c \in F(\frac{1}{n})$.

With this setup, here's the Winnow2 algorithm [Lit88].

Algorithm (Winnow2):

1. Take in an "update parameter" $\alpha > 1$ as input.
2. Initial hypothesis: $h(x) = \mathbb{1}\{w_1x_1 + \ldots + w_nx_n \geqslant n\}$, initial weights $w_i = 1$ for all i, fixed threshold $\theta = n$.
3. (Updates.) For each example x:
   - If $h(x) = c(x)$, then keep w the same; no updates.
   - (Demotion for false positive.) If $h(x) = 1$ and $c(x) = 0$, then for each i such that $x_i = 1$, set $w_i \leftarrow w_i/\alpha$.
   - (Promotion for false negative.) If $h(x) = 0$ and $c(x) = 1$, then for each i such that $x_i = 1$, set $w_i \leftarrow w_i\alpha$.

Here's a cheap-and-easy mistake bound (which we will not prove) for this algorithm:

Theorem 4.1. *For $0 \leqslant \delta \leqslant 1$, let $c : \{0,1\}^n \to \{0,1\}$ be a δ-separable monotone LTF over $\{0,1\}^n$ and let $0 \leqslant u_1, \ldots, u_n \leqslant 1$ be its weights as above. If we run Winnow2 with $\alpha = 1 + \frac{\delta}{2}$, then it makes* *at most*

$$\frac{8}{\delta^2} + \left(\frac{5}{\delta} + \frac{14 \ln n}{\delta^2}\right) \sum_{i=1}^{n} u_i = \mathcal{O}\left(\frac{\log n}{\delta^2} \sum_{i=1}^{n} u_i\right)$$

many mistakes on c.

Remarks 4.4. *Note that:*

1. *this mistake bound is logarithmic in the number of attributes!* 🙂
2. *we make many mistakes for a tiny δ—this makes sense, since in this case there is little separation (with respect to the hyperplane) for training examples; the learning problem is much harder.*
3. *If $\sum_{i=1}^{n} u_i$ is large, then we make more mistakes (e.g. if all the $u_i = 1$, then we have an extra n in the mistake bound).*

Example 4.5 (r-out-of-k LTF)**.** Remember that these functions are in $F(\frac{1}{r})$, and have n variables, k of which are nonzero. Each of the k variables have weight $1/r$ and so we have that $\sum_{i=1}^{n} u_i = \frac{k}{r}$. Therefore, the mistake bound is

$$\mathcal{O}\left(\frac{\log n}{(1/r)^2} \frac{k}{r}\right) = \mathcal{O}(kr \log n),$$

which could be very good (if k and r are small enough, then this goes much slower than n.)

Example 4.6. We can show that any 1-DL can be expressed as a δ-separable monotone LTF for some $\delta = \frac{1}{2^{\Theta(n)}}$. (To do this, we remember that the weights are like $2^r, 2^{r-1}, 2^{r-2}, \dots$ for the r variables that appear in order in the decision list.) So we can use Winnow2 to learn a length-r 1-DL with mistake bound $2^{\Theta(n)} \cdot \log n$, because we pay $1/\delta^2$ to learn the 1-DL, and the $\delta = 2^{-r}$, so that $1/\delta^2 = 1/(1/2^r)^2 = 2^{2r} \cdot \log n$. This is a different way of learning 1-DLs.

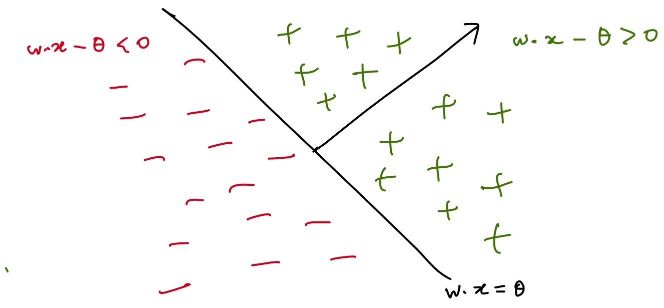
Remarks 4.7.

1. *We can extend Winnow2 to learn LTFs over $[0, 1]$—but we need to make more assumptions like the type we're about to make with the Perceptron algorithm.*
2. *We can use Winnow2 to learn non-monotone LTFs, disjunctions, etc.*
3. *Winnow2 is quite noise-tolerant in practice—things are not too weird, dividing by, say 1.1 isn't very far from multiplying by 1.1. This is good in practice.*

The Winnow algorithms developed by Nick Littlestone use multiplicative weight updates. Now, we will look at the Perceptron algorithm which was developed 30 years earlier by Frank Rosenblatt, and uses additive weight updates.

§4.2 Perceptron

Perceptron [Ros58] is an algorithm for learning linear threshold functions over $\mathbb{R}^n$ with a margin. Remember the issue with learning LTFs over $[0, 1]$—we eventually keep collapsing and getting closer and closer, but we cannot quite get to learn properly. In Perceptron, we will use a margin assumption (not just on the real line as above, but on any n-dimensional hyperplane in $\mathbb{R}^n$) to give a successful learning algorithm as long as we are promised that we don't get too close to the boundary. Let's discuss some setup. Consider the target LTF $\text{sign}(v \cdot x - \theta)$, i.e. $\mathbb{1}\{v_1 x_1 + \ldots + v_n x_n - \theta\}$:

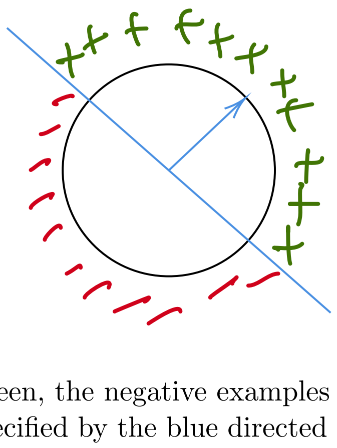


We want to learn the decision boundary above by taking points in the plane, guessing which side of the boundary it lies on, getting the actual label, and learning from our correct/incorrect guess. We make some simplifying assumptions:

1. Assume that $\theta = 0$ that is, the hyperplane separating positive and negative examples is origin-centred. This is true without loss of generality: suppose we're given data on a non-origin-centred LTF, we can tweak things to ensure it is origin-centred by adding a new coordinate $x_{n+1} = 1$ to every example, and add a new coordinate to the weight vector. So $x = (x_1, x_2, \ldots, x_n, 1)$ and $v = (v_1, v_2, \ldots, v_n, -\theta)$.
2. Assume each example $x \in \mathbb{R}^n$ (for origin-centred $v \cdot x \geqslant 0$) is a unit vector, i.e. $\|x\|_2 = \sqrt{\sum_{i=1}^n x_i^2} = 1$. This is true without loss of generality: suppose we're given data on a non-unit vector, we can tweak things to ensure it is unit by dividing each example by its norm.
3. Assume that without loss of generality, the target vector $v \in \mathbb{R}^n$ has $\|v\|_2 = 1$. This is natural since $v \cdot x \geqslant 0$, for example

$$3x_1 + 4x_2 \geqslant 0 \iff \frac{3}{5}x_1 + \frac{4}{5}x_2 \geqslant 0 \iff \left\langle \frac{3}{5}, \frac{4}{5} \right\rangle \cdot \langle x_1, x_2 \rangle \geqslant 0.$$

Here's a geometric picture of the learning problem (in the case $n = 2$):



Here the positive examples are in green, the negative examples in red, and the linear separator is in blue. An arbitrary data vector is specified by the blue directed line segment, and the dot product of

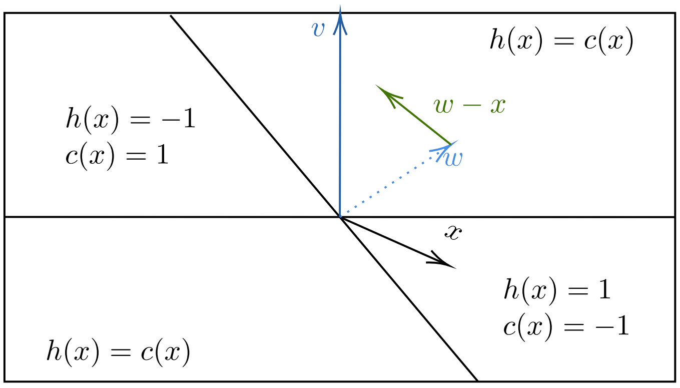
this vector with the weight vector is the signed distance from the origin to the line. The sign of this dot product determines the label of the data vector.
Here's the main Perceptron algorithm:

Algorithm (Perceptron):

1. Maintain a hypothesis weight vector $w \in \mathbb{R}^n$, initially set to 0^n.
2. On each example $x \in \mathbb{R}^n$, predict $\operatorname{sign}(w \cdot x)$.
3. Make the following updates:
    - (Correct prediction.) If the prediction is correct, make no update.
    - (False positive.) If the prediction gives $w \cdot x > 0$ but the truth $v \cdot x < 0$:
        - Set $w \leftarrow w - x$ coordinate-wise.
    - (False negative.) If the prediction gives $w \cdot x < 0$ but the truth $v \cdot x > 0$:
        - Set $w \leftarrow w + x$ coordinate-wise.

    That is, set $w \leftarrow w + \operatorname{sign}(v \cdot x)x$.

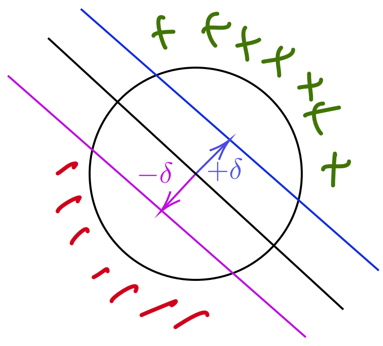
Does this algorithm make sense? Consider a false positive mistake, that is, a mistake where $w \cdot x$ is too big. When we go through Perceptron, the new weight hypothesis w is $w - x$. If we get a new w, the new hypothesis is $(w - x) \cdot x$, which is $w \cdot x - x \cdot x = w \cdot x - 1$. Now, $w \cdot x$ was too big; we have now reduced it by 1, so Perceptron adjusts it to be closer to the value we want to learn. Here's a more visually compelling picture:



The following theorem says that if we run Perceptron on origin-centred LTF-classified data according to our setup, where all the vectors are unit vectors, and none of the examples we get are too close to the separating hyperplane, then we won't make too many mistakes before we learn the target vector v.

Theorem 4.2 (Perceptron convergence theorem)**.** *Suppose we run Perceptron on a sequence of examples in $\mathbb{R}^n$ labelled by $v \cdot x \geqslant 0$ where v and all the examples x satisfy our assumptions above. Let $\delta = \min\{|v \cdot x|\}$, where the minimum is taken over all examples the algorithm ever receives. Then Perceptron makes at most $1/\delta^2$ mistakes before it learns v.*

Since v and x are unit vectors, $|v \cdot x|$ is the length of the projection of x in the direction of v. So what this theorem is really saying is that as long as none of the examples lie within $\pm\delta$ of the separating hyperplane, we will achieve a good mistake bound for Perceptron.



The proof works by considering two key quantities, $|w \cdot v|$ (the length of the hypothesis vector's projection in the "right" direction, v) and $\|w\|^2$ (the squared length of the hypothesis vector). The crux of the proof is to show that whenever we make a mistake, $|w \cdot v|$ increases by a nontrivial amount, and every time we make a mistake, $\|w\|^2$ increases by "not too much." We will prove the theorem by two lemmas.

Lemma 4.8. *After M mistakes, $w \cdot v \geqslant \delta \cdot M$.*

Proof. Initially, $w = 0^n$, so initially (when we have made no mistakes) $w \cdot v = 0$. Suppose now that $M > 0$. We claim that each mistake causes $w \cdot v$ to go up by at least δ. Let w be the old hypothesis vector before a mistake, so that $w + \text{sign}(v \cdot x)x$ is the new hypothesis vector after making a mistake on x. Thus, after making a mistake, we have

$$\begin{aligned}(w + \text{sign}(v \cdot x)x) \cdot v &= w \cdot v + \text{sign}(v \cdot x)v \cdot x \\ &= w \cdot v + |v \cdot x| \\ &\geqslant w \cdot v + \delta,\end{aligned}$$

by the definition of δ. So each mistake causes $w \cdot v$ to go up by at least δ, and since we make M mistakes, $w \cdot v \geqslant \delta \cdot M$. ☐

Lemma 4.9. *After M mistakes, we have that $\|w\|^2 \leqslant M$.*

Proof. We claim that each mistake causes $\|w\|^2$ to go up by at most 1. Let w be the old hypothesis vector; initially, $w = 0^n$, so initially $\|w\|^2 = 0$. Suppose now that $M > 0$. Then,

$$\begin{aligned}\|w + \text{sign}(v \cdot x)x\|^2 &= (w + \text{sign}(v \cdot x)x) \cdot (w + \text{sign}(v \cdot x)x) \\ &= w \cdot w + 2\text{sign}(v \cdot x)w \cdot x + \text{sign}(v \cdot x)^2 x \cdot x \\ &= \|w\|^2 + 2\text{sign}(v \cdot x)w \cdot x + 1 \\ &\leqslant \|w\|^2 + 1,\end{aligned}$$

where $x \cdot x = 1$ since x is a unit vector, and $\text{sign}(v \cdot x) = -1$ since we made a mistake on x. So each mistake causes $\|w\|^2$ to go up by at most 1, and since we make M mistakes, $\|w\|^2 \leqslant M$. ☐

Proof of Perceptron Convergence Theorem. For any vector w, recall that if v is a unit vector, then $w \cdot v = \|w\| \cos \beta$, where β is the angle between w and v, so $w \cdot v \leqslant \|w\|$. Thus, by Lemma 1, after M mistakes made by Perceptron, we have that

$$\delta M \leqslant w \cdot v \leqslant \|w\| \leqslant \sqrt{M} \implies M \leqslant 1/\delta^2,$$

where the first inequality follows from Lemma 1 and the last follows from Lemma 2. $\square$

Next time: More discussion of Perceptron, examples, and an extension to kernel methods.

§5 Lecture 05—20th September, 2023

Last time.

- Presented Winnow2 for $F(\delta)$, the class of δ-separated monotone LTFs.
- Perceptron algorithm ($1/\delta^2$ mistake bound for unit x with margin δ), Perceptron convergence theorem.

Today.

- Example of applying the Perceptron convergence theorem.
- Dual Perceptron, "kernelisation."

§5.1 Applying the Perceptron Convergence Theorem

Suppose there is a king with n advisors (n odd). The king feels that some subset $S \subseteq [n]$ of these advisors give bad advice. The king has to make a binary decision; to do this, he gets all n votes, negates the votes of the advisors in S, and goes with the majority vote.

Suppose you are a political analyst. You observe this process repeatedly—the votes of the council, and the king's decision. You'd like to come up with accurate predictions about what the king's decision will be given what the candidates have decided. How many mistakes will you make in the worst-case while trying to predict the king's decision, giving the advisors' votes?

This can be formulated as a linear threshold function online learning problem!

Recall that $\mathsf{MAJ} : \{0,1\}^N \to \{0,1\}$ can be written as

$$\mathsf{MAJ}(x_1, \ldots, x_n) = \mathbb{1}\left\{x_1 + \ldots + x_n \geqslant \frac{n}{2}\right\}.$$

The target could be something like $\mathsf{MAJ}(x_1, \overline{x_2}, \overline{x_3}, x_4, x_5, \overline{x_6}, x_7, x_8)$, where the untrustworthy advisors are $S = \{2, 3, 6\}$. Let's make things easier with a different, equivalent version. Define $\mathsf{MAJ} : \{\pm 1\}^n \to \{\pm 1\}$ as

$$\mathsf{MAJ}(x) = \operatorname{sign}(x_1 + \ldots + x_n).$$

The S-negated majority function we're trying to learn is then

$$\mathsf{MAJ}_S(x) = \operatorname{sign}(v_1 x_1 + \ldots + v_n x_n),$$

where

$$v_i = \begin{cases} +1 & \text{if } i \notin S, \\ -1 & \text{if } i \in S. \end{cases}$$

The Perceptron algorithm requires that v and x are both unit vectors, so we can renormalise both:

$$v_i = \begin{cases} +1/\sqrt{n} & \text{if } i \notin S, \\ -1/\sqrt{n} & \text{if } i \in S. \end{cases} \qquad x_i = \begin{cases} 1/\sqrt{n} & \text{if they vote } \phi, \\ -1/\sqrt{n} & \text{if they vote } \neg\phi. \end{cases}$$

By the Perceptron convergence theorem, we want to get some kind of lower bound on $|v \cdot x|$, so that we can adequately choose the δ satisfying $|v \cdot x| \geqslant \delta$ for all x we receive. Consider:

$$\begin{aligned} |v \cdot x| &= |v_1 x_1 + \ldots + v_n x_n| \\ &= \left| \pm \frac{1}{\sqrt{n}} \cdot \frac{1}{\sqrt{n}} \pm \ldots \pm \frac{1}{\sqrt{n}} \cdot \frac{1}{\sqrt{n}} \right| \\ &= \left| \pm \frac{1}{n} \right| \quad \text{since } n \text{ is odd} \\ &= \frac{1}{n}. \end{aligned}$$

Thus $\delta = 1/n$, so that we make at most $1/(1/n)^2 = n^2$ mistakes.

What if n was even? If we took $x \in \{+1, -1\}^n$, then $x_1 + \ldots + x_n = 0$, so $\mathsf{MAJ}(x) = 0$—this is bad! We can get past this by choosing $\text{sign}(x_1 + \ldots + x_n + \frac{1}{2})$ instead of $\text{sign}(x_1 + \ldots + x_n)$. Both functions are logically equivalent, and the second has the added benefit that it helps us avoid the margin at the cost of an extra dimension for the 1/2.

§5.2 Dual Perceptron and kernel functions

A nice thing about Perceptron is that it has a dual form that looks a bit different from Perceptron but is exactly equivalent. This form makes it possible to apply kernel methods, as we will soon see.

Key idea of dual Perceptron Note that to run the Perceptron algorithm, we only need to be able to compute inner products of pairs of examples. When we run Perceptron, initially the weight vector $w = 0^n$. After the first mistake (on $x^{(1)}$), now w is $x^{(1)}$ or $-x^{(1)}$. So $w \cdot x$ is then updated to be $x^{(1)} \cdot x$ or $-x^{(1)} \cdot x$ (the inner product of a pair of examples). At some point it makes another mistake on some $x^{(2)}$. Then if w were $x^{(1)}$, and we've made a false negative prediction, w becomes $-x^{(1)} + x^{(2)}$, so that $w \cdot x$ becomes $-x^{(1)} \cdot x + x^{(2)} \cdot x$. Since the weight vector is a simple linear combination of the examples, we can just compute the inner product between the weight vector and the fresh example as the corresponding linear combination of the inner products that constitute the weight vector and the fresh example. So after k updates on $x^{(i_1)}, \ldots, x^{(i_k)} \in \mathbb{R}^n$, we can compute w as

$$w = \sum_{j=1}^{k} c(x^{(i_j)}) x^{(i_j)},$$

and consequently the value of $w \cdot x$ (the key thing that Perceptron needs to compute to make its prediction) is precisely

$$w \cdot x = \left(\sum_{j=1}^{k} c(x^{(i_j)}) x^{(i_j)} \right) \cdot x = \sum_{j=1}^{k} c(x^{(i_j)}) x^{(i_j)} \cdot x.$$

This makes it clear that to run dual Perceptron, we only need to be able to compute inner products of pairs of examples.

Here's a more formal description of dual Perceptron:
Maintain the hypothesis as a list of training examples $\left[\left(x^{(1)}, y^{(1)}\right), \ldots, \left(x^{(k)}, y^{(k)}\right) \right]^\top$, where $x_i \in \mathbb{R}^n$ is the ith example on which we made a mistake, and $y_i \in \{-1, +1\}$ is the label of the ith example. So given a new example x, we have

$$w \cdot x = \sum_{i=1}^{k} y^{(i)} \left(x^{(i)} \cdot x \right),$$

and we can exactly simulate Perceptron by predicting $\text{sign}(w \cdot x)$.

Why do this? It seems inefficient; after k mistakes, the hypothesis is of size nk. However, this has a significant advantage over the original Perceptron algorithm: we can replace the *inner product* with something called a *kernel function*. This is very powerful: it means that we can run Perceptron by running the dual Perceptron algorithm but over a different, possibly "expanded" feature (instance) space—perhaps an exponentially large feature space—provided that we can evaluate that kernel function efficiently on that feature space.

Kernel functions and feature expansions

Definition 5.1 (Feature expansion)**.** *Let X be an original feature space containing all the data points (e.g. $X = \{0,1\}^n$ or $X = \mathbb{R}^n$) equipped with an inner product. Let X' be an aspirational feature space much richer than X (e.g. $X = \{0,1\}^N$ or $X = \mathbb{R}^N$ for $N \gg n$) and also equipped with an inner product. Then a feature mapping is a mapping $\Phi : X \to X'$ that maps each point in X to a point in X'.*

Example 5.2 (All-conjunctions feature expansion)**.** Let $N = 3^n$, so that $X = \{0,1\}^n$ and $X' = \{0,1\}^{3^n}$. Then $\Phi : \{0,1\}^n \to \{0,1\}^{3^n}$ maps $(x_1, \ldots, x_n)$ to list of all 3^n conjunctions over $x_1, \ldots, x_n$.
As an example, consider the case where $n = 2$ for this feature mapping. Then

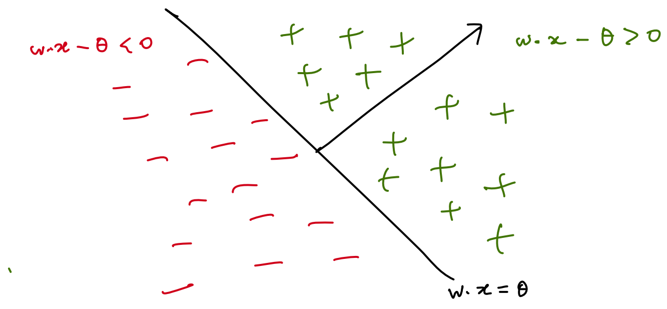
$$\Phi(x_1, x_2) = (1, x_1, \overline{x_1}, x_2, \overline{x_2}, x_1 \wedge x_2, \overline{x_1} \wedge x_2, x_1 \wedge \overline{x_2}, \overline{x_1} \wedge \overline{x_2}).$$

Then $\Phi(x)$ is essentially an expanded version of x as the example above shows.

Definition 5.3 (Kernel function)**.** *The kernel function for a feature expansion $\Phi : X \to X'$ is a function $K : X \times X \to \mathbb{R}$ such that $K(a, b) = \Phi(a) \cdot \Phi(b)$.*

Kernel functions are just inner products taken in richer spaces.

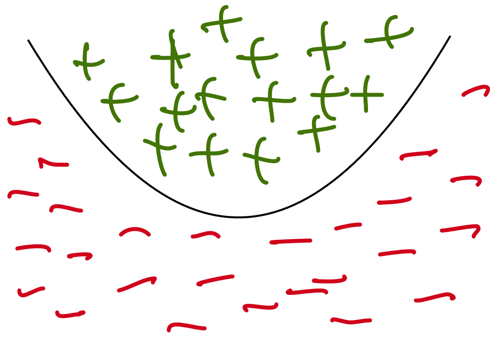
Why do feature expansions? We can use feature expansions to make our data linearly separable. Consider LTFs over $\mathbb{R}^2$, where $x = (x_i, x_j)$ are pairs within some region, $a \leqslant i, j \leqslant b$:



Consider X', the feature expansion of X obtained by allowing monomials of degree up to 2. So X' is the tuple $(1, x_1, x_2, x_1x_2, x_1^2, x_2^2)$ and

$$\Phi(x) = \Phi(x_1, x_2) = (1, x_1, x_2, x_1x_2, x_1^2, x_2^2).$$

Note that with this feature expansion, we can get "curvy" decision boundaries (or boundaries that look like a parabola or something):



So we can intuitively see that LTFs over an expanded feature space are richer and more expressive (we are not restricted to linear separability like we were in the original feature space).

Suppose now that you have data in X, but your aspiration is to run Perceptron on the Φ-feature-expanded data X'. If we didn't know about dual Perceptron and didn't have a kernel function, we'd have to do it directly:

1. Whenever you get x, write down $\Phi(x)$.
2. Maintain a weight vector w' in the X' domain in the expanded feature space.

This is reasonable, but extremely slow it takes a large time $N \gg n$ to write down $\Phi(x)$ for each x. A better way is to use the kernel function K for Φ and run dual Perceptron, replacing every $a \cdot b$ in dual Perceptron with $K(a, b)$. This exactly simulates running Perceptron over the Φ-expanded examples (this is great, given we can efficiently compute K).

The good news is that for some natural and interesting feature expansions, there *is* a computationally efficient way to compute $K(a, b)$, much faster than writing $\Phi(a)$ and $\Phi(b)$ and computing their dot product.

Example 5.4 (All-conjunctions feature expansion)**.** Recall the example above with $X = \{0,1\}^n$ and $X' = \{0,1\}^{3^n}$. We will see that given $a, b \in \{0,1\}^n$, the "conjunction kernel" $K(a,b)$ for the all-conjunctions feature expansion Φ can be computed in time poly(n), even though the dimension N of the expanded feature space is 3^n.
Let $\textsc{Same}(a,b) \subseteq [n]$ denote the set of all bit positions $i \in [n]$ such that $a_i = b_i = 1$. We claim that $K(a,b) = 3^{|\textsc{Same}(a,b)|}$. To see this, note that every subset $S \subseteq \textsc{Same}(a,b)$ corresponds to a different conjunction such that that coordinate is 1 in both $\Phi(a)$ and $\Phi(b)$. Moreover, it is easy to see that these are the only conjunctions that evaluate to 1 in both $\Phi(a)$ and in $\Phi(b)$ (because if S contains some $i \in [n] \setminus \textsc{Same}(a,b)$, then either a_i or b_i is 0, so the corresponding conjunction evaluates to 0 and not 1 in either $\Phi(a)$ or $\Phi(b)$). So, imagining that we wrote out the entire inner product as a sum of 3^n values (each o which is $0 \cdot 0$, $0 \cdot 1$, $1 \cdot 0$ or $1 \cdot 1$), we see that the number of $1 \cdot 1$ summands is precisely $3^{|\textsc{Same}(a,b)|}$, and so $\Phi(a) \cdot \Phi(b) = K(a,b) = 3^{|\textsc{Same}(a,b)|}$.
Given this, it is clearly trivial to compute $K(a,b) = 3^{|\textsc{Same}(a,b)|}$ in $\mathcal{O}(n)$ time, which is indeed faster than we could write out the 3^n dimensional vectors $\Phi(a)$ and $\Phi(b)$.

Could we run dual Perceptron over DNFs? No; it might make exponentially many mistakes when learning DNFs because the margins δ could be arbitrarily small.

Next time: Discuss more algorithms for generic concept classes $\mathcal{C}$ as well as upper and lower bounds.

§6 Lecture 06—25th September, 2023

Last time.

- Example of using the Perceptron convergence theorem to help a king decide based on his subjects' opinions.
- Overview of kernel functions, feature expansions, and dual Perceptron:
  - running Perceptron over high-dimensional feature spaces in a computationally efficient way.
  - can run Perceptron inexpensively over very high-dimensional feature spaces in poly-time using dual Perceptron along with an evaluatable kernel function, as long as we don't make too many mistakes.

Today. Start generic bounds for OLMB learning; algorithms and lower bounds that apply to any finite concept class $\mathcal{C}$.

- Halving algorithm; essentially a version of binary search, mistake bound $\leqslant \lg |\mathcal{C}|$.
- Randomised halving algorithm; a probabilistic twist on the halving algorithm, expected mistake bound $\leqslant \ln C + \mathcal{O}(1)$.
- Bad news: VC dimension.

---

§6.1 Halving algorithm

The intuition for this algorithm is that it is an online learning variant of binary search; at each round, we want to make a lot of progress and kill off at least half of the concepts in the class everytime we make a mistake. The simple way to do this is to always predict according to the majority vote of the concepts that are still alive. If you do that, then everytime they are mistakes, the majority of them were wrong, and we can cross out this majority off of the list of concepts, reducing the size of the list by at least half.

More formally, let CONSIST be the set of all $c \in \mathcal{C}$ that are consistent with all labelled examples $(x, c(x))$ seen so far. The halving algorithm [Slo89] then proceeds like this:

Algorithm (Halving algorithm, HA):

1. Initially, set CONSIST $= \mathcal{C}$.
2. Given an example (mistake) x:
    - Make a prediction as the majority vote over all $c \in$ CONSIST.
    - On every example, if we make a mistake, update CONSIST.

Example 6.1. Suppose that $|\text{CONSIST}| = 2000$ and 1150 of them predict 1 on x and 850 predict 0. Then the halving algorithm predicts 1 on x. If the true label $c(x) = 0$, then we update to the other side so that CONSIST now contains only the 850 concepts that predict 0 on x.

Theorem 6.1. *For any finite concept class $\mathcal{C}$, the mistake bound of the halving algorithm is at most* $\lg |\mathcal{C}|$.

Proof. Each mistake cuts the size of CONSIST by $\geqslant 1/2$, since we are effectively multiplying the size of CONSIST by some $0 < p \leqslant 1/2$ since we are taking the majority vote. So by updating on mistakes, the concept class's size goes from $|\mathcal{C}|$ to some number that is at least 1 (since the target is always in CONSIST). This is exactly the definition of $\geqslant \lg |\mathcal{C}|$. □

Example 6.2 (1-decision-lists of length r)**.** Let $\mathcal{C} = \{\text{1-DLs of length } r \text{ over } \{0,1\}^n\}$, so that there are $|\mathcal{C}| \leqslant (4n)^r \cdot 2 \leqslant 2^{\mathcal{O}(r)} \cdot n^{\mathcal{O}(r)}$, so that if we used the halving algorithm in this situation, the mistake bound would be $\lg |\mathcal{C}| \leqslant \mathcal{O}(r \cdot \log n)$. But the time per trial is $2^{\Theta(r)} n^{\Theta(r)}$, which is bad!

Remarks 6.3. 1. *The mistake bound above is great!* 🙂

2. *However, this is not computationally efficient! We need to spend $\approx |\mathcal{C}|$ time per trial to maintain the consist set determining the majority vote.* ☹

3. *We use a weirdish hypothesis,* $\mathsf{MAJ}(c_{i_1}, c_{i_2}, \ldots, c_{i_M})$*, where the* $c_{i_j} \in \mathcal{C}$*. This could be bad!* ☹

4. *This algorithm is very brittle if there is noise—more later.* ☹ *(We will fix (3) above using*

the randomised halving algorithm, and fix (4) using the weighted majority algorithm.)

5. *For some concept classes, the mistake bound from the halving algorithm is the best-possible mistake bound.*

> **Example 6.4.** Let X be a finite set and $\mathcal{C}$ is the exhaustive concept class of every Boolean function over X, so that $|\mathcal{C}| = 2^{|X|}$. Then the mistake bound of the halving algorithm is $\lg |\mathcal{C}| = |X|$. Now, no matter what any algorithm says on each $x \in X$, it could be wrong: no algorithm has mistake bound $\leqslant |X| - 1$. So the halving algorithm is optimal.

6. *On the other hand, there are other concept classes $\mathcal{C}$ where the halving algorithm does better than $\lg |\mathcal{C}|$ mistakes. The canonical example is the set of singletons: $X = \{1, \ldots, N\}$, and $\mathcal{C}$ has a c_i for each $i \in [N]$ defined as $c_i = \{i\}$. Then $|\mathcal{C}| = N$, but the halving algorithm makes only 1 mistake, since although the initial prediction is 0 one mistake eliminates all but the target $\mathcal{C}$:*

$$A = \begin{array}{r} \\ x=1 \\ x=2 \\ \vdots \\ x=N \end{array} \!\!\begin{array}{c} \begin{array}{cccc} c=c_1 & c=c_2 & \cdots & c=c_N \end{array} \\ \begin{pmatrix} 1 & 0 & \cdots & 0 \\ 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & c(x)=1 \end{pmatrix} \end{array}$$

§6.2 Randomised halving algorithm

Motivation. Our OLMB definition is very paranoid and "worst-case"; we insist that an algorithm can be learned with mistake bound M if for *every* sequence of examples (that is given to us by an adversary), and *every* choice of a target concept (which an adversary can choose), the algorithm makes at most M mistakes. This is a very strong requirement—it is equivalent to assuming that the sequence of examples $x^{(1)}, \ldots, x^{(n)} \in X$ and the target concept c are chosen by an omniscient foe; we can even change c as long as there's always some $c \in \mathcal{C}$ that is consistent with the examples seen so far.

Here's a relaxed, indifferent assumption. Assume that the target concept and the sequence of examples are both chosen ahead of time, and only then does the learning process start. This assumption is sometimes called the assumption of an *oblivious adversary*. If we make this assumption, it could be productive for our learning algorithm to use some extent of randomness. (Note that in the cases we've been studying until now, the *omniscient adversary* knows what the coin will come up as, but there's no point in tossing a coin or taking chances.)

We will now change the rules of the learning game under the assumption of an oblivious adversary. We then define a "slicker" version of the halving algorithm that uses randomness, called the randomised halving algorithm (for an arbitrary finite concept class $\mathcal{C}$). This algorithm is very similar to sampling one random voter in an election and asking them who they voted for; if they voted for the majority, then we are likely to be right.

Here's the algorithm, formally stated:

Algorithm (Randomised halving algorithm, RHA):

1. Maintain Consist as before and initially, set Consist $= \mathcal{C}$.
2. Each time we make a mistake on $x \in X$:
    - Update the hypothesis to be a uniformly random concept in Consist.

Remark 6.5. *Note that this is in some sense more efficient than the original halving algorithm—we don't need to tabulate every concept predicted on an example to make a concrete prediction. It's still as hard to sample and maintain the consist set, but it is certainly better than the original halving algorithm.*

We now state and prove the mistake bound of the randomised halving algorithm.

Theorem 6.2. *For any fixed concept $c \in \mathcal{C}$ and any sequence of examples $x^{(1)}, \dots, x^{(n)} \in X$, the expected number of mistakes made by the randomised halving algorithm is* $\ln |\mathcal{C}| + \mathcal{O}(1)$.

Proof. Fix $c \in \mathcal{C}$ and $x^{(1)}, \dots, x^{(n)} \in X$. Consider the point in the execution of the randomised halving algorithm where there are r concepts left in Consist, i.e. $|\text{Consist}| = r$. Write M_r to denote the expected number of mistakes the randomised halving algorithm makes from this point onwards, that is, on the rest of the example sequence and this c. Notice that

$$M_{|\mathcal{C}|} = \mathbb{E}\left[\text{number of mistakes the RHA makes in total on all the elements of Consist}\right].$$

is exactly the quantity we want to upper-bound. Suppose now that we order the concepts in Consist by when the remaining sequence of examples and the target concept eliminate them:

$$\text{Consist} = \{c_1, c_2, \dots, c_r\}.$$

Notice that c_r is the target concept; it is the right answer, so it is never eliminated. This is a well-defined ordering because we have a fixed remaining sequence of exampls and a fixed target concept (because of the oblivious adversary assumption).
Consider a point in the execution of the randomised halving algorithm when h, the uniform selection over $c_1, \dots, c_r$, is chosen. If $h = c_r$, then we will make no more mistakes—there is a $1/r$ chance of this happening. If $h = c_t$ for some $t < r$, one mistake eliminates at least $c_1, \dots, c_t$. Therefore at most $r - t$ concepts are left, so the expected number of mistakes made on this bunch is M_{r-t}. Also, there is a $1/r$ chance for each of the other possibilities of $t = 1, \dots, r-1$. What all of this is saying is that

$$M_r \leqslant \underbrace{\frac{1}{r} \cdot 0}_{\text{for } c_r} + \underbrace{\frac{1}{r} \cdot \sum_{t=1}^{r-1} (1 + M_{r-t})}_{\text{for all the concepts before } c_r}.$$

Now we have a recurrence relation which lets us control the expected number of mistakes. By the definition of the mistake bound we only need consider the worst-case scenario:

$$\begin{aligned} M_r &= \frac{r-1}{r} + \frac{1}{r} \cdot (M_1 + \ldots + M_{r-1}) \\ M_1 &= 0. \end{aligned}$$

We now solve this rcurrence relation. Multiplying out by r yields

$$\begin{aligned} rM_r &= r - 1 + (M_1 + \ldots + M_{r-1}) \\ (r-1)M_r &= r - 2 + (M_1 + \ldots + M_{r-2}) \end{aligned}$$

Subtracting leads to

$$rM_r - (r-1)M_r = (r-1) - (r-2) + M_{r-1}.$$

This means that

$$r(M_r - M_{r-1}) = 1 \iff M_r = M_{r-1} + \frac{1}{r}.$$

Therefore, we develop the sequence

$$\begin{aligned} M_1 &= 0 \\ M_2 &= \frac{1}{2} \\ M_3 &= \frac{1}{2} + \frac{1}{3} \\ M_4 &= \frac{1}{2} + \frac{1}{3} + \frac{1}{4} \\ &\vdots \\ M_r &= \frac{1}{2} + \frac{1}{3} + \ldots + \frac{1}{r} = \ln r + \gamma, \end{aligned}$$

where γ is the Euler-Mascheroni constant, so that $M_r \leqslant \ln r + \mathcal{O}(1)$. The proof is done; we can conclude that $M_{|\mathcal{C}|} \leqslant \ln|\mathcal{C}| + \mathcal{O}(1)$. □

Remarks 6.6. *1. This is better! We've saved quite a bit:* $\lg|\mathcal{C}| - \ln|\mathcal{C}| = \ln 2$. ☺

2. The hypothesis is simpler! It is now an element of $\mathcal{C}$, *not a majority vote over* $\mathcal{C}$. ☺

3. It is easier to predict $h(x)$, *compared to how it would be with the halving algorithm.* ☺

4. There is a drawback: we assumed that there is an oblivious adversary, and the analysis makes no sense without this assumption. ☹

5. There is another drawback: it's still computationally inefficient. ☹

§6.3 VC dimension: a lower bound for online learning

The Vapnik-Chervonenkis (VC) dimension [BEHW89] is a combinatorial parameter of a concept class that measures how expressive, rich, complex, etc. the concept class is. The concept class is complicated—it is a set system, sometimes even a set of sets of sets, so there's no single way to exhaustively capture its complexity, but the VC dimension is a good first step. We will present a number of definitions for the VC dimension, but first, we need to discuss what it means to shatter a set.

Definition 6.7 (Shattered set)**.** *Fix $\mathcal{C}$ a concept class over domain X. Let $S \subseteq X$. We say that S is shattered by $\mathcal{C}$ if every subset of S is achieved by some concept in $\mathcal{C}$. That is:*

- *(subset point-of-view) for every subset $T \subseteq S$, there is some $c \in \mathcal{C}$ such that $c \cap S = T$.*
- *(labelling point-of-view) for every Boolean labelling of S, i.e. every $f : S \to \{0,1\}$, some $c \in \mathcal{C}$ achieves that labelling, i.e. $c(x) = f(x)$ for all $x \in S$.*

Example 6.8. Let $X = \{1,2,3,4,5\}$ and $\mathcal{C} = \{c_1, \dots, c_6\}$, where:

$$\begin{aligned} c_1 &= \{1,2,3\} \\ c_2 &= \{2,4,5\} \\ c_3 &= \{3,4\} \\ c_4 &= \{1,2,5\} \\ c_5 &= \{1,3,5\} \\ c_6 &= \{5\}. \end{aligned}$$

The set $S = \{5\}$ is shattered—note that $T = \{5\} \subseteq S$, with $T \subseteq c_2, c_4, c_5, c_6$ and $T \not\subseteq c_1, c_3$.
The set $S = \{2,4\}$ is shattered—all possible subsetes of S are achieved via the concepts c_1 through c_6.
Observation. No set of size 3 is shattered: to shatter a 3-element set, we'd need to have $2^3 = 8$ different subsets, but there are only 6 concepts in $\mathcal{C}$.

The above example leads us to a definition for the VC dimension, more later:

Definition 6.9. *The VC dimension of the concept class $\mathcal{C}$ is the size of the largest set $S \subseteq X$ that is shattered by $\mathcal{C}$.*

§7 Lecture 07—27th September, 2023

Last time. Started generic bounds for online learning:

- Halving algorithm, mistake bound $\lg |\mathcal{C}|$.
- Randomised halving algorithm, expected mistake bound $\ln |\mathcal{C}| + \mathcal{O}(1)$.
- Started to define VCDIM($\mathcal{C}$).

Today. The edges of OLMB learning:

- VCDIM($\mathcal{C}$) as the lower bound for any OLMB algorithm.
- Predicting from expert advice (with rebranded noise-tolerant variants of the halving algorithm and the randomised halving algorithm).
  - The weighted majority algorithm.
  - The randomised weighted majority algorithm.

§7.1 VC dimension as a lower bound for OLMB learning

Recall the definitions from last time:

Definition 7.1 (Shattered set)**.** *Fix $\mathcal{C}$ a concept class over domain X. Let $S \subseteq X$. We say that S is shattered by $\mathcal{C}$ if every subset of S is achieved by some concept in $\mathcal{C}$. That is:*

- *(subset point-of-view) for every subset $T \subseteq S$, there is some $c \in \mathcal{C}$ such that $c \cap S = T$.*
- *(labelling point-of-view) for every Boolean labelling of S, i.e. every $f : S \to \{0,1\}$, some $c \in \mathcal{C}$ achieves that labelling, i.e. $c(x) = f(x)$ for all $x \in S$.*

Definition 7.2 (VC dimension)**.** *The VC dimension of the concept class $\mathcal{C}$ is the size of the largest set $S \subseteq X$ that is shattered by $\mathcal{C}$.*

Remark 7.3. *Remember that the VC dimension is a combinatorial parameter of a concept class—it makes sense to talk about the VC dimension of the class of 1-DLs or the class of s-term DNFs, etc. It doesn't make sense to talk about the VC dimension of a concept or an example or a hypothesis, etc. Importantly, it is a property of a concept class, not of a concept.*

Definition 7.4 (VC dimension)**.** *The VC dimension of the concept class $\mathcal{C}$ is the smallest number d such that no set of $d+1$ examples in the domain X of $\mathcal{C}$ is shattered by $\mathcal{C}$.*

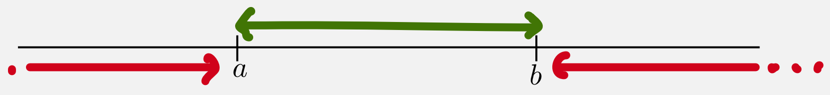
VC dimensions of concept classes can be infinite.

Remark 7.5. *If for every d, there exist some set S with $|S| = d$ that S is shattered by $\mathcal{C}$, then we say that* $\mathrm{VCDIM}(\mathcal{C}) = \infty$.

This definition should provide some insight as to how we usually find VC dimensions—we cannot find them by solving some quadratic equation or some shit. To show that $\mathrm{VCDIM}(\mathcal{C}) = \delta$, we typically:

1. exhibit a set of points $S \subseteq X$, with $|S| = \delta$, and argue that $\mathcal{C}$ shatters S (this shows that $\mathrm{VCDIM}(\mathcal{C}) \leqslant \delta$).
2. argue that *no* set $S \subseteq X$ with $|S| = \delta + 1$ can be shattered by $\mathcal{C}$ (this shows that we have that $\mathrm{VCDIM}(\mathcal{C}) < \delta + 1$).

Example 7.6. Take $X = \mathbb{R}$ and $\mathcal{C}$ to be all closed closed intervals $[a, b]$:



1. Can we shatter one point? That is, is there a placement of a point x_1 on the line such that $\mathcal{C}$ labels it positive or negative? Obviously yes—we can either place it within the interval or outside the interval. So $\mathrm{VCDIM}(\mathcal{C}) \geqslant 1$.
2. Can we shatter two points? That is, is there a placement of two points x_1, x_2 with $x_1 < x_2$ on the line such that $\mathcal{C}$ labels them positive or negative? Yes—we can either

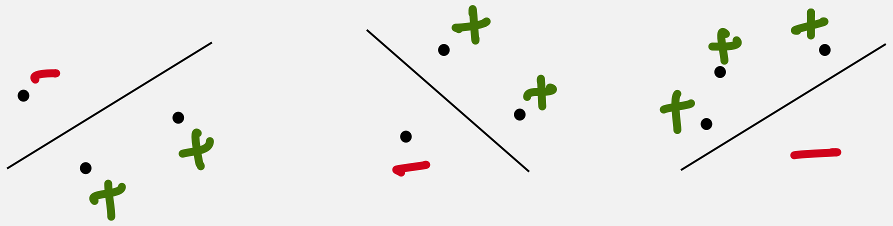
place them both within the interval, both outside the interval, or one within and one outside. So $\mathrm{VCDIM}(\mathcal{C}) \geqslant 2$.

3. Can we shatter three points? No! Consider three points x_1, x_2, x_3 on the real line with $x_1 < x_2 < x_3$. There is no closed interval that includes x_1 and x_3 (marks them positive) but excludes x_2 (marks it negative). So $\mathrm{VCDIM}(\mathcal{C}) < 3$.

By this analysis, $\mathrm{VCDIM}(\mathcal{C}) = 2$.

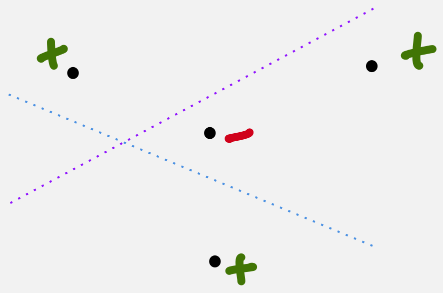
Example 7.7. Take the plane $X = \mathbb{R}^2$ and $\mathcal{C}$ to be the concept class of all halfspaces on X, i.e. the set of all the linear threshold functions taking this form. We claim that $\mathrm{VCDIM}(\mathcal{C}) \geqslant 3$.

1. If we pick three collinear points a, b, c, these are not shatterable—there is some labelling $(a, b, c) \mapsto (+, -, +)$ which is not attainable by any halfspace. But there *is* a set of three points that can be shattered—just pick any non-collinear triple of points a, b, c, and this will be shatterable:

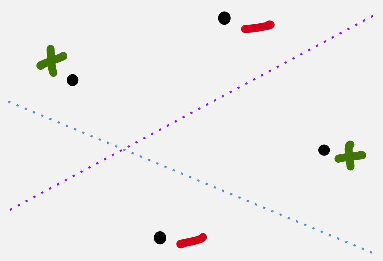


Since there *is* a shattered set of three points—not every set of three points can be shattered, but this is not a requirement of the definition—we then conclude that $\mathrm{VCDIM}(\mathcal{C}) \geqslant 3$.

2. We will now argue that $\mathrm{VCDIM}(\mathcal{C}) > 4$, by showing that no set of four points is shattered.
   - If three of the points are collinear, then by the same argument as above, we cannot shatter them.
   - If no three of the points are collinear:
     - If one point lies in the triangle formed by the other three, then we cannot shatter the four points, as the following labelling demonstrates:



     - If the four points form a convex quadrilateral, then this is not shatterable, as the following labelling shows:



Thus, $\text{VCDIM}(\mathcal{C}) < 4$, and we can conclude that $\text{VCDIM}(\mathcal{C}) = 3$.

Note that in the example above we have used the fact that if a set of n points cannot be shattered by $\mathcal{C}$, then any set with $m > n$ points cannot be shattered by $\mathcal{C}$ either.

Example 7.8. Consider the concept class $\mathcal{C}$ which is the set of all monotone conjunctions over $\{0,1\}^n$. We claim that $\text{VCDIM}(\mathcal{C}) \geqslant n$. Consider a list of all the Boolean sentences

$$\begin{gathered}0,1,1,1,\ldots,1,1,\\1,0,1,1,\ldots,1,1,\\1,1,0,1,\ldots,1,1,\\\vdots\\1,1,1,1,\ldots,0,1,\\1,1,1,1,\ldots,1,0.\end{gathered}$$

This is shatterable—we can classify all of them as positive using the empty conjunction $\{\}$, and we can classify all of them as negative using the conjunction $\bigwedge_{i=1}^n x_i$. Within those extremes, we can achieve a labelling with the concept $c = \bigwedge x_i$ wherever $x_i = 0$ in the original list of Boolean sentences. So $\text{VCDIM}(\mathcal{C}) \geqslant n$.
We also claim that $\text{VCDIM}(\mathcal{C}) \leqslant n$. Recall that $|\mathcal{C}| = 2^n$, and we saw last time that $\text{VCDIM}(\mathcal{C}) \leqslant \lg|\mathcal{C}|$ (as we need 2^k concepts to shatter k points). Thus, $\text{VCDIM}(\mathcal{C}) = n$.

Let's now return to OLMB learning, with a lower bound on every learning algorithm in this model.

Claim 7.9. *If* $\text{VCDIM}(\mathcal{C}) = d$, *then any OLMB algorithm for* $\mathcal{C}$ *must have (worst-case) mistake bound* $\geqslant d$.

Proof. Let $S = \left\{x^{(1)}, x^{(2)}, \ldots, x^{(d)}\right\}$ be a size-d shattered set. For some d, on a specified example sequence $x^{(1)}, x^{(2)}, \ldots, x^{(d)}$, no matter which bit $h(x^{(i)})$ the learner predicts, there is a $c \in \mathcal{C}$ such that $c(x^{(i)}) \neq h(x^{(i)})$ for all $i = 1, \ldots, d$—this means that the mistake bound of the algorithm is at least d. □

Remarks 7.10.

1. *This claim does not guarantee the existence of an OLMB learning algorithm with mistake bound* d. *It only says that if such an algorithm exists, then it must have mistake bound* $\geqslant d$.
2. *We now know that the elimination algorithm for monotone conjunctions (or disjunctions) is optimal!* 🙂

3. *With the oblivious adversary model however, we can obtain a lower bound of $\geqslant d/2$ expected mistakes. The adversary can chose a priori and an example sequence to be the shattered set $\{x^{(1)}, x^{(2)}, \ldots, x^{(d)}\}$, and then picks a concept c uniformly at random from the 2^d concepts in $\mathcal{C}$ that shatter these points. In this setting, the learner's task is precisely to predict a sequence of d fair coin flips, which has expected number of mistakes $\geqslant d/2$.*

   So we don't improve very much in this setting, unfortunately. ☹

§7.2 Prediction from expert advice

§7.2.1 The weighted majority algorithm

Consider the following scenario: You go to the racetrack with a group of friends. There is a sequence of races; for each race, a friend makes a bet (a prediction on what the final outcome of the race will be). You don't know horses, so you want to ask your friends questions, combine their predictions for each race, and make your own bet. Your goal is to predict really well, at least as well as the most successful of our friends.
In hindsight—if we knew at the end of the day which of our friends would make the most winning bets/correct predictions—we could have just copied all of their bets. Also, if all the friends do really badly, we cannot expect to do well. So we have no absolute guarantees on performance, and this hindsight is the best we can reasonably hope for.

As it turns out, we can provably do almost as well as the best friend (or, the most successful expert), even without knowing who that friend is or having the benefit of hindsight. Formally speaking, here's the setup for the weighted majority algorithm. We have a pool of N experts (which we can think of as N experts in a concept class) and a sequence of trials. Each expert has a 0/1 prediction at each trial. Also, we arbitrarily set a parameter $0 \leqslant \beta < 1$ prior to commencing the algorithm. Here's the weighted majority algorithm:

Algorithm (Weighted majority algorithm, WMA):

1. (Hypothesis.) Each expert has a weight w_i; initially all $w_i = 1$.
2. At each trial, the ith expert makes some prediction $z_i \in \{0, 1\}$:
   - Compute $q_0 = \sum_{i:z_i=0} w_i$ and $q_1 = \sum_{i:z_i=1} w_i$.
     (Note that $q_0 + q_1 = \sum_{i=1}^{N} w_i = W$, the total weight.)
   - If $q_0 \geqslant q_1$, predict 0; otherwise, predict 1.
3. (Update rule.) Given the result of a trial and the true outcome:
   - For each expert i such that z_i was wrong, set $w_i \leftarrow \beta \cdot w_i$.

Remarks 7.11. *1. If $\beta = 0$, then this is exactly the halving algorithm! Here's an explicit correspondence:*

$$\textit{expert } i \longleftrightarrow i\textit{th concept in } \mathcal{C}_i \textit{ (where } |\mathcal{C}| = n)$$

$$\textit{prediction of expert } i \textit{ on trial } j \longleftrightarrow c_i(x^j), \textit{ the label of the } j\textit{th example in an ex. sequence}$$

2. *We can also see that this syncs up nicely with a situation in whichb no concept in the class is perfect—we can imagine that there is an ideal expert who makes the fewest mistakes and the other experts are just noisy, "corrupted" versions of this ideal expert. So once again, this is like a kind of generalisation of the halving algorithm.*

3. *In particular, the halving algorithm performs poorly with mistakes; this algorithm doesn't!* 🙂

We can now prove a bound on the performance of the weighted majority algorithm.

Theorem 7.1. *For any sequence of trials, suppose that the best expert makes m mistakes. Then the weighted majority algorithm makes at most*

$$\frac{\log N + m \cdot \log \frac{1}{\beta}}{\log \frac{2}{1+\beta}}$$

many mistakes.

Before we go into the proof, here are some bounds on the mistakes that this theorem gives:

- If $\beta = 1/2$, then the weighted majority algorithm makes $\leqslant 2.41(m + \ln N)$ mistakes.
- If $\beta = 3/4$, then the weighted majority algorithm makes $\leqslant 2.2m + 5.2 \ln N$ mistakes.
- If $\beta \to 1$ in the sense of $\beta = 1 - \varepsilon$ where $\varepsilon \to 0$, then the weighted majority algorithm makes $\approx 2m + \frac{2}{\varepsilon} \ln N$ mistakes.

Proof of WMA mistake bound. Let $W = q_0 + q_1 = \sum_{i \in \mathbb{N}} w_i$ be the total weight of the experts. Initially $W = N$. At each mistake, at least half the total weight W predicts wrong and is multiplied by β, so after a mistake, the total weight goes from W to at most $\frac{W}{2} + \frac{W}{2}\beta = W(1+\beta)/2$. Thus, after M mistakes, the total weight is at most $N\left((1+\beta)/2\right)^m$.
On the other hand, the best expert makes m mistakes, so her weight t is $\geqslant \beta$ always, so that $W \geqslant t \geqslant \beta^m$. Therefore we have by combining the inequalities that

$$\begin{aligned} \beta^m \leqslant M \leqslant N\left(\frac{1+\beta}{2}\right)^m &\iff m \log \beta \leqslant \log N + M \cdot \log \frac{1+\beta}{2} \\ &\iff m \log \frac{1}{\beta} \geqslant -\log N + M \cdot \log \frac{2}{1+\beta} \\ &\iff M \geqslant \frac{\log N + m \cdot \log \frac{1}{\beta}}{\log \frac{2}{1+\beta}}, \end{aligned}$$

and the proof is done. □

Next time: Randomised weighted majority algorithm, PAC learning basics.

§8 Lecture 08—2nd October, 2023

Last time. Concluding OLMB learning:

- VCDIM($\mathcal{C}$): examples, the VC dimension as the lower bound for any OLMB algorithm.
- Prediction with expert advice: the weighted majority algorithm.

Today. Moving on from OLMB:

- Prediction with expert advice: the randomised weighted majority algorithm.
- Start the next unit: Probably Approximately Correct (PAC) learning.
  - motivation
  - basic definition
  - learning $\mathcal{C} = \{\text{intervals of } \mathbb{R}\}$.

---

§8.1 The randomised weighted majority algorithm

Recall that in the limit, as $\beta \to 1$, the number of mistakes made by the weighted majority algorithm is $\approx 2m + \frac{2}{\varepsilon}\ln N$. Why is this divisible by 2?

Consider a run of the weighted majority algorithm on a trial where vote $q_0 = 0.51N$ and $q_1 = 0.49N$. Then the algorithm predicts 0, *for sure*. This is a perpetual predictability guarantee that the omnipotent adversary can potentially exploit. This is potentially something we can ignore—if instead we had $q_0 = 0.9N$ and $q_1 = 0.1N$, then the algorithm only shoots itself in the foot by predicting 1 instead—but here, where the votes are so close together, what if weighted majority didn't go with the majority vote instead? This is almost as good...

The discussion above leads to a natural variant of weighted majority that takes advantage of our predictability: we flip a biased coin, biased towards the likelier outcome. This is the key idea behind the randomised weighted majority algorithm.

Algorithm (Randomised weighted majority algorithm, RWMA):

1. As before, assume that we have an expert pool of N experts, as well as the demotion parameter β.
2. (Hypothesis.) Each expert has a weight w_i; initially all $w_i = 1$.
3. At each trial:
   - Predict z_i
   - Output z_i with probability $\frac{w_i}{W}$, where $W = \sum_{i=1}^{N} w_i$.
4. (Update rule.) Given the result of a trial and the true outcome:
   - For each expert i such that z_i was wrong, set $w_i \leftarrow \beta \cdot w_i$.

Remark 8.1. *The z_i need not be binary anymore—we're more flexible here!* 🙂

Theorem 8.1. *Assume the oblivious adversary model (that is, the sequence of trial outcomes is fixed in advance before randomised weighted majority starts running). If the best expert in the pool makes*

m mistakes, then the expected number of mistakes M made by the randomised weighted majority algorithm is

$$M \leqslant \frac{m \cdot \ln \frac{1}{\beta} + \ln N}{1-\beta}.$$

Again before we prove this, here are a few examples:

- If $\beta = 1/2$, then the randomised weighted majority algorithm makes $\leqslant 1.39m + 2\ln N$ mistakes.
- If $\beta = 3/4$, then the randomised weighted majority algorithm makes $\leqslant 1.15m + 4\ln N$ mistakes.
- If $\beta \to 1$ in the sense that $\beta = 1 - \varepsilon$ and $\varepsilon \to 0$, then the randomised weighted majority algorithm makes $\approx m + \frac{1}{\varepsilon}\ln N$ mistakes.

Notice how randomisation always shaves a factor of 2 off in the third example—although it is under a different model, that of the oblivious adversary.

Proof. Consider any sequence of T trials, and denote F_i as the fraction of the total weight at trial i that is on the wrong predictions. Then we have that

$$F_i = \Pr[\text{RWM makes a mistake on trial } i].$$

Let M be the expected number of mistakes made by randomised weighted majority. Then $M = \sum_{i=1}^{T} F_i$ is exactly the quantity we want to bound. Before the ith trial, the total weight

$$W = \underbrace{(1-F_i)W}_{\text{correct weight}} + \underbrace{F_i W}_{\text{incorrect weight}}.$$

After the ith trial, the new W is $(1-F_i)W + \beta F_i W = (1-(1-\beta)F_i)\,W$. Initially the weight $W = N$, so the final total weight is

$$W = N \prod_{i=1}^{T} (1-(1-\beta)F_i)\,W.$$

Just as before, the best expert made m mistakes, so also, $W \geqslant \beta^m$. Therefore,

$$N \cdot \prod_{i=1}^{T} (1-(1-\beta)F_i)\,W \geqslant \beta^m \iff -\ln N - \sum_{i=1}^{T} \ln(1-(1-\beta)F_i) \geqslant m \ln \frac{1}{\beta}.$$

Recallng that $1 - x \leqslant e^{-x} \iff x \leqslant -\ln(1-x)$ for all x, we have that

$$-\ln N - \sum_{i=1}^{T} \ln(1-(1-\beta)F_i) \geqslant m \ln \frac{1}{\beta} \implies M = \sum_{i=1}^{T} F_i \leqslant \frac{m \cdot \ln \frac{1}{\beta} + \ln N}{1-\beta}$$

and the proof is done. □

§8.2 A New Learning Model: Probably Approximately Correct (PAC) Learning

Motivation Let's think about the online learning with mistake bound model. What are its drawbacks?

1. Online mistake-bounded learning insists on suspicion and paranoia. We're forced to deal with a worst-case assumption on the example seqeunce. As we progressed, we used even darker language, sometimes insisting that the data itself is "adversarial."
2. The mistake bound criterion—our performance metric in OLMB learning—looks at the whole process right from the beginning. This is perhaps too harsh—we're judging the learning algorithm when it hasn't even had time to learn.
3. The mistake bound could also be unsatisfying. If we haven't saturated the mistake bound, there are no guarantees of a good prediction on the next example.

Probably approximately correct (PAC) learning [Val84] will address all of these issues. In PAC learning:

- we assume that all the examples $x \in X$ that the learner gets are independently and identically distributed from some fixed, unknown, arbitrary distribution $\mathcal{D}$ over X—we will be learning from random examples, not from adversarially chosen examples.
- we have a *batch* model—we don't use any online learning setting where we punish the algorithm for making mistakes, etc—we train the algorithm on a data set of $(x, c(x))$ pairs, where $x \sim_{\text{i.i.d.}} \mathcal{D}$, and the algorithm outputs just one h based on the whole data set.
- our new performance guarantee is that the algorithm should "do well" on future $x \sim \mathcal{D}$.

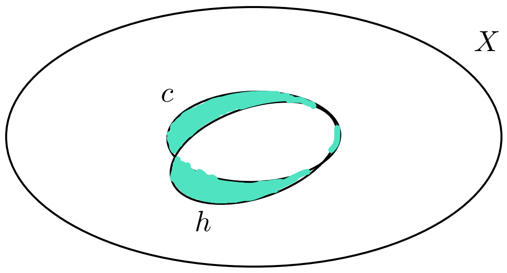
Let's now get more precise about PAC learning, before we proffer a definition. The PAC framework for learning a concept class $\mathcal{C}$ using a hypothesis class $\mathcal{H}$ is as follows:

- like OLMB learning, where there is an unknown target concept $c \in \mathcal{C}$, and $\mathcal{C}$ is known to the algorithm. New to the learning game is some unknown distribution $\mathcal{D}$ over X. In each learning stage,
  - the learner is giving a training data set of m (this will be important later) labelled examples $(x^{(1)}, c(x^{(1)})), \dots, (x^{(m)}, c(x^{(m)}))$, where $x^{(i)} \sim_{\text{i.i.d.}} \mathcal{D}$. Equivalently, the learner is given access to an example oracle $\text{EX}(c, \mathcal{D})$ and can use it m times to obtain $(x, c(x))$ where $x \sim_{\text{i.i.d.}} \mathcal{D}$. (Note that this is a passive model—we have no control over which examples the learner gets.)
  - the learner does some computation (runs the learning algorithm) on m data points, and on the basis of that computation, outputs one hypothesis $h \in \mathcal{H}$ where $\mathcal{H}$ is the hypothesis class and $h : X \to \{0, 1\}$.
- we measure our performance by finding how likely a fresh random point drawn from this distribution is to be labelled correctly or incorrectly by this hypothesis. The definition below helps us formalise this notion.

Definition 8.2. *Let $h, c : X \to \{0, 1\}$, let $\mathcal{D}$ be a distribution over X. Then the error of h on c under $\mathcal{D}$ is*

$$\text{err}_{\mathcal{D}}[h, c] := \Pr_{x \sim \mathcal{D}}[h(x) \neq c(x)].$$

Consider the following picture:



Then $\mathrm{err}_{\mathcal{D}}[h,c] = \Pr_{x\sim\mathcal{D}}[x \in (c \setminus h) \cup (h \setminus c)]$ is the probability that x lies in the shaded region (i.e. the probability that x lies in the symmetric difference of the sets c and h).

Remarks 8.3. *1. It might not be possible to achieve $\mathrm{err}_{\mathcal{D}}[h,c] = 0$, as there could be very small probability mass on some arbitrary example. So we shouldn't for instance expect 0 error in the PAC learning model.*

2. In fact, we cannot expect a 100% guarantee even of moderate error: there is always some chance of achieving non-representative samples.

Both of the remarks above can be captured by an error parameter ε and a confidence parameter δ respectively. Furthermore, we can expect to achieve lowe error with high confidence—this is exactly what it means to succeed in the PAC learning model.

Definition 8.4 (PAC-learning)**.** *An algorithm A PAC-learns a concept class $\mathcal{C}$ using a hypothesis class $\mathcal{H}$ if it is the case that:*

- *for every concept $c \in \mathcal{C}$,*
- *for every distribution $\mathcal{D}$ over X,*
- *for every error (or accuracy) parameter $\varepsilon > 0$, and confidence parameter $\delta > 0$,*

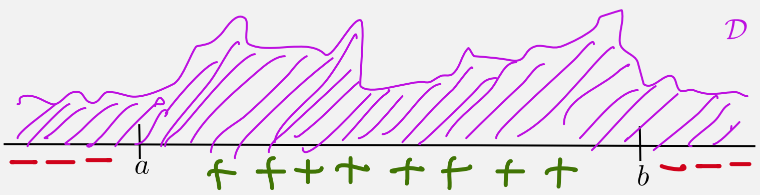
if A is given ε and δ, as well as access to $\mathrm{EX}(c,\mathcal{D})$, then with probability at least $1-\delta$, A outputs a hypothesis $h \in \mathcal{H}$ such that

$$\mathrm{err}_{\mathcal{D}}[h,c] \leqslant \varepsilon.$$

Let's make a few comments about this definition.

1. The probability $\Pr\left[\mathrm{err}_{\mathcal{D}}[h,c] \leqslant \varepsilon\right] \geqslant 1-\delta$ is taken over the m calls to the oracle $\mathrm{EX}(c,\mathcal{D})$ and any internal randomness of the algorithm A.
2. The number of samples m used by the algorithm is also called the *sample complexity* of the algorithm—it's really how much data the algorithm uses—and can depend on ε and δ.
3. *Running time.* An efficient PAC-learner runs in time $\mathrm{poly}(\frac{1}{\varepsilon}, \frac{1}{\delta})$ time. If $X = \{0,1\}^n$ or $X = [0,1]^n$, then we should achieve $\mathrm{poly}(n, \frac{1}{\varepsilon}, \frac{1}{\delta})$ time for the PAC-learner.
4. If $\mathcal{H} = \mathcal{C}$, we say that the learner is a proper learner.

Example 8.5 (Learning intervals of [0, 1])**.** Take $X = [0,1]$ and c to be any arbitrary interval $[a,b]$ with $0 \leqslant a, b \leqslant 1$, e.g. $c = [0.03, 1/\pi]$. We want to learn the target concepts $c = [a,b]$.



Here, the curve $\mathcal{D}$ represents the boundaries of some distribution such that the area under $\mathcal{D}$ is 1. Recall that we cannot learn this concept class in the mistake bound model—the adversary could just keep increasing the precision of the boundaries of the interval. But there is a very efficient PAC-learning algorithm for this class:

1. Draw m examples.
2. Let a' be the smallest value in the sample, and let b' be the largest value in the sample.
3. Return the interval $h = [a', b']$.

Next time, we will analyse this algorithm.

§9 Lecture 09—4th October, 2023

Last time. Learning with expert advice, intro. PAC learning:

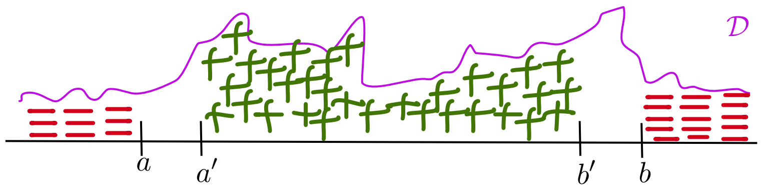
- Randomised weighted majority algorithm, expected mistakes $\leqslant (1-\beta)^{-1}\left(m \ln \frac{1}{\beta} + \ln N\right)$.
- Probably approximately correct (PAC) learning: motivation, basic definition, example.

Today. More PAC learning:

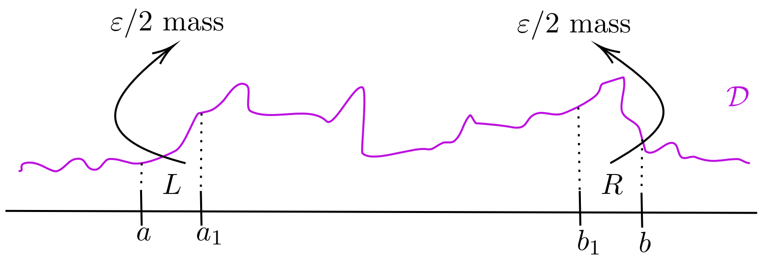
- PAC-learnability of intervals, continued.
- Converting OLMB algorithms to PAC algorithms. (Can get all OLMB results for free in PAC setting—good!)
- Revisiting PAC learning definition: size of concepts, efficiency of evaluating h.

§9.1 PAC learning intervals, continued

Recall the problem of "learning intervals" from last time: $X = [0, 1]$, $\mathcal{C}$ is all the $[a, b]$, with $0 \leqslant a, b \leqslant 1$:



Recall that we set a' (resp. b') to be the smallest (resp. largest) positive example in the m samples given, and that the algorithm outputs $h = [a', b']$. Note that the only source of randomness in this scenario for our algorithm are false negatives, those case where the hypothesis $h = 0$ but the concept $c = 1$ (that is, the misclassified example lies in the gap between the positively-classified and negatively-classified examples on either side). We want to show that with high probability, h doesn't miss more than ε amount of $\mathcal{D}$'s probability mass. To show this, let's focus on the distribution of the examples instead of the actual examples:



Let the target concept $c = [a, b]$. Define a_1 to be the value such that $\Pr_{x\sim\mathcal{D}}[x \in [a, a_1]] = \varepsilon/2$ and b_1 to be the value such that $\Pr_{x\sim\mathcal{D}}[x \in [b_1, b]] = \varepsilon/2$. Now $L = [a, a_1]$, and $R = [b_1, b]$. We observe that if our sample of labelled examples contains a point in L or a point in R, then

$$\mathrm{err}_{\mathcal{D}}[h, c] = \Pr_{x\sim\mathcal{D}}[x \in [a, a_1]] + \Pr_{x\sim\mathcal{D}}[x \in [b_1, b]] \leqslant \varepsilon/2 + \varepsilon/2 = \varepsilon.$$

This is good, as it provides a sufficient condition for a good hypothesis. So we want to simply upper bound the probability Pr[the example does not hit L] and Pr[the example does not hit R]. Since a single $x \sim \mathcal{D}$ misses L with probability $1 - \frac{\varepsilon}{2}$, we have by independence that

$$\Pr[m \text{ i.i.d. examples all miss } L] = \left(1 - \frac{\varepsilon}{2}\right)^m,$$

and likewise,

$$\Pr[m \text{ i.i.d. examples all miss } R] = \left(1 - \frac{\varepsilon}{2}\right)^m.$$

By the union bound—that is, the fact that $\Pr[A \cup B] \leqslant \Pr[A] + \Pr[B]$ for any events A and B—it suffices to have

$$2\left(1 - \frac{\varepsilon}{2}\right)^m \leqslant \delta \implies \left(1 - \frac{\varepsilon}{2}\right)^m \leqslant \frac{\delta}{2}.$$

So m must satisfy $(1 - \frac{\varepsilon}{2})^m \leqslant \frac{\delta}{2}$, and so recalling that $1 - x \leqslant e^{-x}$, we have that the choice $m = \frac{2}{\varepsilon}\ln\frac{2}{\delta}$ is sufficient, because,

$$\left(1 - \frac{\varepsilon}{2}\right)^{\frac{2}{\varepsilon}\ln\frac{2}{\delta}} \leqslant \left(e^{-1}\right)^{\ln\frac{2}{\delta}} = \frac{\delta}{2},$$

where we have applied the fact that $(1 - x)^{\frac{1}{x}} \leqslant e^{-1}$. The proof is done.

Remarks 9.1. *1. The confidence parameter δ guards against unlikely samples. A hypothesis is "good" with probability $1 - \delta$.*

2. Even very good hypotheses $h \in \mathcal{H}$ are allowed to have an error of $\leqslant \varepsilon$ fraction of $\mathcal{D}$.

§9.2 OLMB learnability $\implies$ PAC learnability

Let A be an OLMB algorithm for a concept class $\mathcal{C}$ with mistake bound M. Can we get a PAC learner for A? The answer is yes.

Definition 9.2 (Conservative algorithm)**.** *A mistake-bounded learning algorithm is called conservative if it changes its hypothesis only when it makes a mistake.*

Lemma 9.3. *If L is a mistake-bounded learning algorithm for learning a concept class $\mathcal{C}$, then there is a conservative mistake-bounded learning algorithm L' for learning $\mathcal{C}$ with the same mistake bound.*

Proof. Let L' mimic L', except that after each correct prediction, it does not change h. Suppose that L' makes m' mistakes on an input sequence $x^{(1)}, \dots, x^{(n)}$, and let the subsequence on which the m' mistakes are made be $x^{(i_1)}, \dots, x^{(i_{m'})}$. Then L' run on this subsequence will still make m' mistakes, since it is conservative.

However, L will make the same mistakes as L' when run on this subsequence, as the two algorithms behave the same after they make a mistake. Thus, for each input sequence, if L' makes m' mistakes on this input sequence, then there esists a sequence such that L also makes m' mistakes, so that L' and L have the same mistake bound. □

Theorem 9.1. *Let A be an OLMB algorithm for a concept class $\mathcal{C}$ with mistake bound M. Then there is a PAC learning algorithm for $\mathcal{C}$ with sample complexity*

$$m \leqslant M + \frac{M+1}{\varepsilon} \ln \frac{M+1}{\delta}.$$

Proof. Without loss of generality, we know from the lemma above that A is conservative—it only changes its hypothesis when it makes a mistake. We claim that the following algorithm A' is a PAC learner for $\mathcal{C}$:

1. Run A on the sequence of fresh i.i.d. draws from $\mathrm{EX}(c, \mathcal{D})$.
2. Keep track of the number of current examples h got right. If h ever goes for
$$\frac{1}{\varepsilon} \ln \frac{M+1}{\varepsilon}$$
examples without making a mistake, stop and output h.

We claim that the sample complexity of A' is as claimed. Consider a sequence of trials on the examples fed into the algorithm, and write ✓ for a correct prediction and ✗ for an incorrect prediction. Then some sequence of trials might look like

$$✓✓\cdots✓\ ✗✓✓\cdots✓\ ✗✓\cdots✓✗$$

Since A is an OLMB algorithm with mistake bound M, we have $\leqslant M$ number of ✗'s in the sequence. Also, by the algorithm, each contiguous block of ✓'s has length $\leqslant \frac{1}{\varepsilon} \ln \frac{M+1}{\varepsilon}$—there are $M+1$ of these blocks, since they are organised around the ✗'s. Thus the total length of the sequence of ✗ and ✓ is

$$m \leqslant M + \frac{M+1}{\varepsilon} \ln \frac{M+1}{\delta},$$

as desired. We will now show that A' is a PAC learner for $\mathcal{C}$. Let us say that a hypothesis h is bad if $\mathrm{err}_{\mathcal{D}}[h, c] > \varepsilon$. Suppose that the current hypothesis h is bad. Then by independence,

$$\Pr[h \text{ gets } k \text{ examples right in a row}] < (1-\varepsilon)^k,$$

and so

$$\begin{aligned}\Pr\left[h \text{ gets } \frac{1}{\varepsilon}\ln\frac{M+1}{\varepsilon} \text{ examples right in a row}\right] &< (1-\varepsilon)^{\frac{1}{\varepsilon}\ln\frac{M+1}{\delta}} \\ &\leqslant e^{-\ln\frac{M+1}{\delta}} \\ &= \frac{\delta}{M+1}.\end{aligned}$$

Since A is conservative, it only ever uses $\leqslant M+1$ different hypotheses h. Hence, at most $M+1$ bad hypotheses are ever used by A. Thus, by the union bound, the probability that A' outputs a bad hypothesis is

$$\Pr[\text{any bad hypothesis is output by } A] \leqslant \frac{\delta}{M+1}\cdot(M+1) = \delta,$$

so that A' is a PAC learner for $\mathcal{C}$. □

Remarks 9.4. *1. If A is computationally efficient, then so is A'—this is a very low-overhead conversion. So we can PAC-learn conjunctions, disjunctions, r-out-of-k LTFs, etc. So if we plug in A as the elimination algorithm with mistake bound n, we'd need a sample complexity of*

$$m(\varepsilon, \delta) = n + \frac{n+1}{\varepsilon}\ln\frac{n+1}{\delta} = \mathcal{O}\left(\frac{n}{\varepsilon}\ln\frac{n}{\delta}\right).$$

But this is not optimal—we'll see why later.

2. PAC learning is no harder than OLMB learning: if $\mathcal{C}$ is OLMB learnable, then $\mathcal{C}$ is PAC learnable.

§9.3 Revisiting the PAC learning algorithm

There are two more issues with the preliminary definition for PAC learning that we gave:

1. *Size/complexity of target concept.* For some concept classes $\mathcal{C}$, there's a natural notion of $\mathrm{size}(c)$, the size of $c \in \mathcal{C}$. To fix this, we could have a refined notion of PAC learning: if we're learning a concept class $\mathcal{C}$ and the target concept $c \in \mathcal{C}$ has size $\mathrm{size}(c)$, then we cna measure time and sample complexity of the learner in terms of $\mathrm{size}(c)$: for example if $X = \{0,1\}^n$, an efficient PAC learner runs in time $\mathrm{poly}(n, \frac{1}{\varepsilon}, \frac{1}{\delta}, \mathrm{size}(c))$.

   Example 9.5 (DNF formulas)**.** If $X = \{0,1\}^m$, then any concept $f : \{0,1\}^n \to \{0,1\}$ is an s-term DNF for sufficiently large s. We could consider the notion of a DNFSIZE(f), which we will define as the minimum number of terms in any DNF for f. For instance, $x_1x_2 \vee x_3x_4$ is a 2-term DNF computing some f, but even if $x_1x_2x_3 \vee x_1x_2\overline{x_3} \vee x_3x_4$ computes the same function, then DNFSIZE$(f) = 2$.

Example 9.6 (Decision trees)**.** The concept class of decision trees is a generalisation of the decision list class. It's a tree that works like this: at each node, read a variable; if it is 0, move left, and if it is 1, move right. Output the label at the terminal node. We can define a function DTSIZE(f) to be the minumum number of leaves in any decision tree for a concept f.

2. *Hypotheses.* When we are learning a Boolean function, we really think of it as an abstract mapping. A learning algorithm outputs a representation of the hypothesis function $h : X \to \{0,1\}$, i.e. a program. If a learning algorithm outputs a program which you cannot solve efficiently, then it's not a very useful learning algorithm. So we want to restrict the class of hypotheses that we consider.

   Definition 9.7. *A hypothesis class $\mathcal{H}$ (a class of programs) over $X = \{0,1\}^n$ is the efficiently evaluatable (or polynomially evaluatable) if every $h \in \mathcal{H}$ can be run on any $x \in X$ in* $\operatorname{poly}(n)$ *time.*

We now redefine our notion of efficient PAC learning:

Definition 9.8. *An efficient PAC learning algorithm A is one such that the runtime of A is polynomial, and for every h that A might output, the runtime of h is polynomial, i.e. A uses only polynomially evaluatable $\mathcal{H}$.*

Next time: Chernoff bounds, hypothesis testing, generic way to do PAC learning.

§10 Lecture 10—9th October, 2023

Last time. More PAC learning:

- PAC-learnability of intervals, continued.
- Converting OLMB algorithms to PAC algorithms. (Can get all OLMB results for free in PAC setting—good!)
- Revisiting PAC learning definition: size of concepts, efficiency of evaluating h.

Today. Concentration inequalities hypothesis testing, Occam's razor.

- Chernoff bounds / tail bounds for sums of independent random variables (with applications to hypothesis testing).
- Learning by finding consistent hypotheses (from a fixed $\mathcal{H}$).
- Occam's razor—a "cardinality" based version of learning.

§10.1 Chernoff bounds

Motivation. Suppose you have a hypothesis $h : X \to \{0,1\}$, as well as an oracle $\mathrm{EX}(c, \mathcal{D})$. We could be interested in determining how good h is, i.e. determining $\mathrm{err}_{\mathcal{D}}[h,c] = \Pr_{x\sim\mathcal{D}}[h(x) \neq c(X)]$.

But we may not be able to figure this out exactly, because $\mathcal{D}$ could be arbitrarily complicated. So a natural thing would be to try to estimate $\text{err}_{\mathcal{D}}[h,c]$ by drawing m examples from $\text{EX}(c,\mathcal{D})$, evaluate $h(x)$ on each, and compute

$$\hat{\varepsilon} = \frac{1}{m}\left(\text{number of the } m \text{ examples such that } h(x) \neq c(x)\right).$$

A natural question is, *How good is this estimate?* Clearly if m is small (resp. m is large), then the estimate $\hat{\varepsilon}$ is likely to be coarse (resp. fine). We need tools from probability to answer this question more informatively; it is exactly equivalent to the following scenario: We get a biased coin which comes up heads with probability p and tails with probability $1-p$, but we don't know p. To estimate p, we compute

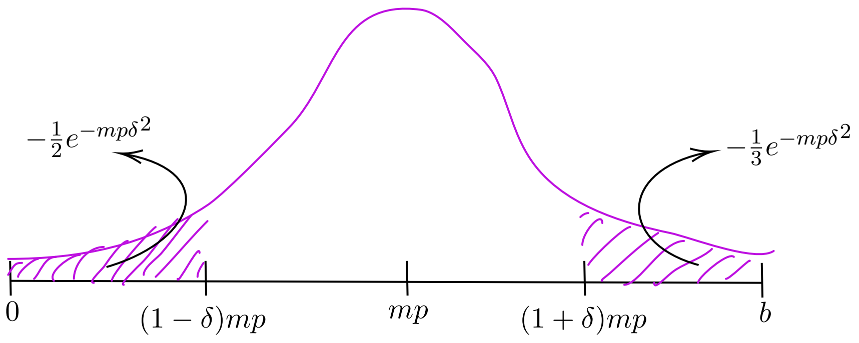
$$\hat{p} = \frac{1}{m}\left(\text{number of heads (or tails) in } m \text{ flips}\right).$$

How good is this estimate?

The Chernoff bound helps us answer this equivalent question. Here's a simplified version of the relative error version of the Chernoff bound: Let $X_1,\ldots,X_m$ be i.i.d. 0/1 random variables satisfying $\mathbb{E}[X_i] = \Pr[X_i = 1] = p$. Let $X = X_1 + \ldots + X_m$, so that $\mathbb{E}[X] = mp$. Then for any $\delta \in (0,1)$,

$$\underbrace{\Pr[X \leqslant (1-\delta)mp]}_{\substack{\text{probability that } X \text{ comes}\\ \text{out surprisingly small}}} \leqslant \exp\left(-\frac{1}{2}\delta^2 mp\right),$$

$$\underbrace{\Pr[X \geqslant (1+\delta)mp]}_{\substack{\text{probability that } X \text{ comes}\\ \text{out surprisingly large}}} \leqslant \exp\left(-\frac{1}{3}\delta^2 mp\right).$$



A similar bound—the Hoeffding bound—is the "additive error" version of the Chernoff bound. Let $X_1,\ldots,X_m$ as above, and write $\hat{p} = \frac{1}{m}\sum_{i=1}^{m} X_i$ as the empirical average of m examples. This error bound says that for a given confidence parameter ε,

$$\Pr[p - \hat{p} \geqslant \varepsilon] \leqslant e^{-2m\varepsilon^2},$$
$$\Pr[\hat{p} - p \geqslant \varepsilon] \leqslant e^{-2m\varepsilon^2}.$$

Note that the Chernoff bounds only apply in the case of *independent* random variables!

Example 10.1 (How many games is Chelsea Football Club likely to win?)**.** Suppose that Chelsea FC wins each game independently with probability $\frac{1}{2}$ in a 162-game period across three Premier League seasons. What is the maximum probability that we win at most 47 games across the three seasons?
We first find one bound: we have that for all $i = 1, \ldots, 162$,

$$X_i = \begin{cases} 1 & \text{a win with probability } \frac{1}{2} \\ 0 & \text{a loss with probability } \frac{1}{2} \end{cases}$$

Comparing $\Pr[X \leqslant 47]$ to $\Pr[X \leqslant (1-\delta)mp]$ where $m = 162$ and $p = \frac{1}{2}$, we have that $81(1-\delta) = 47$, so that $\delta = \frac{34}{81}$. Then by the Chernoff bound, we have that

$$\Pr[X \leqslant 47] \leqslant \exp\left(\frac{-1}{2} \cdot \left(\frac{34}{81}\right)^2 \cdot 162 \cdot \frac{1}{2}\right) = 0.0008,$$

and this low probability is good news for fellow Chelsea fans!

Example 10.2 (Making surveys)**.** You are surveying people to find out what fraction of (independently sampled) people like chocolate ice cream. You want to estimate this fraction to within ± 0.03 with probability at least 0.95. How many people do we need to poll?
We will use the additive bound here. Since $\varepsilon = 0.03$, and

$$e^{-2m\varepsilon^2} = 0.025 = \frac{1}{40} \implies m = \frac{\ln 40}{0.0018}.$$

We need to sample so many people to achieve reasonable accuracy.

Remarks 10.3.

1. *To reiterate, the Chernoff bounds only apply to independent random variables. If the random variables are not independent, then we can't use the Chernoff bounds.*
2. *The Chernoff bounds—in most regimes where we care to apply them—tend to be fairly close to tight, and there are theoretical results on precisely how tight they are.*

Markov's inequality. Here's a weaker, more general tail bound—Markov's inequality. If X is a nonnegative random variable, then for any $k \geqslant 1$, we have that

$$\Pr[X \geqslant k \cdot \mathbb{E}[X]] \leqslant \frac{1}{k}.$$

We will not prove this, but we know it must be true since otherwise the average value of X would be larger than $\mathbb{E}[X]$ itself!

Example 10.4. Let X be the number of children in a random household. Suppose that $\mathbb{E}[X] = 1.85$. Then $\Pr[X \geqslant 10] < \frac{1}{5}$.

§10.2 Learning by finding consistent hypotheses

If $\mathcal{H}$ is fixed, and you can find a hypothesis $h \in \mathcal{H}$ that is consistent with the "enough data", then that's a PAC learner—this is the bumper sticker summary here. The following theorem says that if we fix the concept and hypothesis classes, and draw "enough" examples, then it's really unlikely that any bad hypothesis gets them all right—this is natural!

Theorem 10.1. *Fix a concept class $\mathcal{C}$ and a hypothesis class $\mathcal{H}$, some unknown $c \in \mathcal{C}$, and some distribution $\mathcal{D}$ over X. Let $\left(x^{(1)}, c\left(x^{(1)}\right)\right), \ldots, \left(x^{(m)}, c\left(x^{(m)}\right)\right)$ be m i.i.d. draws from $\mathrm{EX}(c, \mathcal{D})$, where*

$$m = \frac{1}{\varepsilon}\left(\ln\frac{1}{\delta} + \ln|\mathcal{H}|\right).$$

Let us say that h is bad if $\mathrm{err}_{\mathcal{D}}[h, c] > \varepsilon$. Then

$$\Pr\left[\text{any bad } h \in \mathcal{H} \text{ is consistent with draws } \left\{\left(x^{(i)}, c\left(x^{(i)}\right)\right)\right\}\right] \leqslant \delta.$$

Proof. Fix any bad h. Then by independence,

$$\Pr[h \text{ is consistent with } m \text{ examples}] \geqslant (1-\varepsilon)^m.$$

Also, h has at most $|\mathcal{H}|$ many bad hypotheses, so

$$\begin{aligned}\Pr\left[\text{any bad } h \in \mathcal{H} \text{ is consistent with draws } \left\{\left(x^{(i)}, c\left(x^{(i)}\right)\right)\right\}\right] &< |\mathcal{H}| \cdot (1-\varepsilon)^m \\ &= (1-\varepsilon)^{\frac{1}{\varepsilon}\left(\ln\frac{1}{\delta}+\ln|\mathcal{H}|\right)} \cdot |\mathcal{H}| \\ &= e^{-\ln\frac{1}{\delta}-\ln|\mathcal{H}|} \cdot |\mathcal{H}| \\ &= e^{-\ln\frac{|\mathcal{H}|}{\delta}} \cdot |\mathcal{H}| \\ &= \frac{\delta}{|\mathcal{H}|} \cdot |\mathcal{H}| \\ &= \delta,\end{aligned}$$

and so the proof is done. □

We will use this theorem to make the following definition:

Definition 10.5 (Consistent hypothesis finder, CHF)**.** *Fix concept class $\mathcal{C}$ and hypothesis class $\mathcal{H}$. A consistent hypothesis finder (CHF) for the concept class $\mathcal{C}$ with respect to the hypothesis $\mathcal{H}$ is an algorithm B with the following property: if you give B a sample of m labelled examples (labelled according to $c \in \mathcal{C}$) $\left\{\left(x^{(i)}, c\left(x^{(i)}\right)\right)\right\}_{i=1}^{m}$, then B outputs an $h \in \mathcal{H}$ consistent with them, that is, $\forall i \in \{1, \ldots, m\}$, $h(x^{(i)}) = c(x^{(i)})$.*

Note that given a CHF B, we can immediately find a PAC learner for $\mathcal{C}$ using $\mathcal{H}$:

Algorithm (PAC learner via CHF):

1. Draw $m = \frac{1}{\varepsilon}\left(\ln|\mathcal{H}| + \ln\frac{1}{\delta}\right)$ samples.
2. Feed it to the CHF B.
3. Output the hypothesis h returned by B.

Example 10.6 (Monotone disjunctions)**.** Let $\mathcal{C}$ be all monotone disjunctions over $\{0,1\}^n$. Recall that we already have a few algorithms for learning $\mathcal{C}$:

- the elimination algorithm: OLMB mistake bound $M \leqslant n$.
- OLMB to PAC conversion: sample complexity $m \approx \frac{n}{\varepsilon} \ln \frac{n+1}{\varepsilon}$.

In fact, the elimination algorithm is a CHF for $\mathcal{C}$ with respect to $\mathcal{H} = \mathcal{C}$! So $m = \frac{1}{\varepsilon}\left(\ln|\mathcal{H}| + \ln\frac{1}{\delta}\right)$ is enough. Since $|\mathcal{H}| = |\mathcal{C}| = 2^n$, the sample complexity is just

$$m = \frac{1}{\varepsilon}\left(\ln\frac{2^n}{\delta}\right) = \mathcal{O}\left(\frac{1}{\varepsilon}\left(n + \ln\frac{1}{\delta}\right)\right),$$

for the PAC learner.
(Note that we can also extend this to all 3^n nonmonotone disjunctions with the same asymptotic sample complexity.)

Remarks 10.7.

1. *If you have a CHF for $\mathcal{C}$ using $\mathcal{H}$, then you get a PAC learner for free. Otherwise, things are bad. We will see soon that sometimes the job of a CHF is computationally hard.*
2. *It is* crucial *that we have an a priori fixed $\mathcal{H}$ to apply this theorem. For example, consider the following bogus CHF:*
   - *given sample $S = \left\{\left(x^{(i)}, c\left(x^{(i)}\right)\right)\right\}_{i=1}^m$,*
   - *let h be a lookup table:*

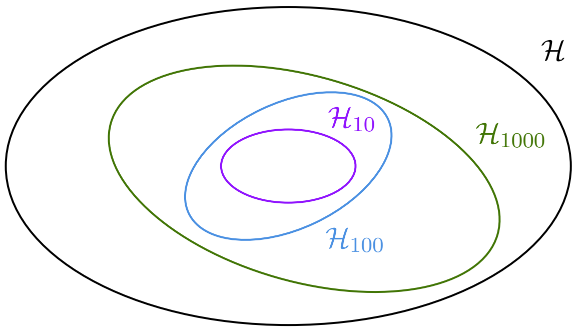
$$\begin{aligned}\mathcal{C} = \quad & \text{“if } x = x^{(1)}\text{, then output } c\left(x^{(1)}\right)\text{,”} \\ & \text{“else if } x = x^{(2)}\text{, then output } c\left(x^{(2)}\right)\text{,”} \\ & \vdots \\ & \text{“else if } x = x^{(m)}\text{, then output } c\left(x^{(m)}\right)\text{.”}\end{aligned}$$

   This is a bogus CHF—we don't have a fixed $\mathcal{H}$ here.

§10.3 Occam's razor

The fundamental principle of Occam's razor [BEHW87] is that we should always favour short concise explanations over long, complicated ones. We will see that this principle is useful in learning theory. Say you have a data set and drew your m labelled examples from the oracle $\mathrm{EX}(c, \mathcal{D})$. The hypothesis h "needs to explain" the m labelling bits. In particular, $\mathcal{H}$ with size $|\mathcal{H}|$ requires $\lg|\mathcal{H}|$ bits to encode any $h \in \mathcal{H}$. So if $m \gg \lg|\mathcal{H}|$, then any $h \in \mathcal{H}$ is a succinct (and accurate) explanation of the data. We can interpret the previous theorem in this way.

One key point of note is that we can "boil down" the stuff from the consistent hypothesis finder to a small-m regime. Recall that the previous theorem says that for all m (the number of examples), the CHF gives some $h \in \mathcal{H}$. Maybe if m is small, we can find a consistent h in a subclass $\mathcal{H}_m$ of $\mathcal{H}$?



If so, we can take advantage of it. Another version of the theorem above is the following:

Theorem 10.2 (Occam's razor)**.** *Fix the sample complexity m, a concept class $\mathcal{C}$, and a hypothesis class $\mathcal{H}$. Suppose that there is a subset $\mathcal{H}_m \subseteq \mathcal{H}$ such that*

$$m \geqslant \frac{1}{\varepsilon}\left(\ln|\mathcal{H}_m| + \ln\frac{1}{\delta}\right)$$

such that given any set of m labelled examples according to some concept $c \in \mathcal{C}$, there is a hypothesis $h \in \mathcal{H}_m$ consistent with the m examples. So if L is any algorithm that outputs such an $h \in \mathcal{H}_m$ when given a sample of size m, then L run on m examples from $\mathrm{EX}(c, \mathcal{D})$ *is an (ε, δ)-PAC learner.*

Proof. Same as before, but with $|\mathcal{H}_m|$ instead of $|\mathcal{H}|$. □

§11 Lecture 11—11th October, 2023

Last time. Concentration inequalities, hypothesis testing, Occam's razor.

- Chernoff bounds / tail bounds for sums of independent random variables (with applications to hypothesis testing).
- Learning by finding consistent hypotheses (from a fixed $\mathcal{H}$).
- Occam's razor—a "cardinality" based version of learning.

Today.

- Application of Occam's razor: learning sparse disjunctions with few examples, with a hypothesis that is a pretty sparse disjunction, via a "greedy set cover" heuristic.
- Proper vs improper learning:
  - can efficiently improperly learn 3-term DNFs. 🙂
  - unless NP ⊆ RP, cannot efficiently properly learn 3-term DNFs. 🙁

Let's remind ourselves of Occam's razor:

Theorem 11.1 (Occam's razor)**.** *Fix the sample complexity m, a concept class $\mathcal{C}$, and a hypothesis class $\mathcal{H}$. Suppose that there is a subset $\mathcal{H}_m \subseteq \mathcal{H}$ such that*

$$m \geqslant \frac{1}{\varepsilon}\left(\ln|\mathcal{H}_m| + \ln\frac{1}{\delta}\right)$$

such that given any set of m labelled examples according to some concept $c \in \mathcal{C}$, there is a hypothesis $h \in \mathcal{H}_m$ consistent with the m examples. So if L is any algorithm that outputs such an $h \in \mathcal{H}_m$ when given a sample of size m, then L run on m examples from $\mathrm{EX}(c, \mathcal{D})$ is an (ε, δ)-PAC learner.

§11.1 Application: learning sparse disjunctions efficiently via greedy set cover

Set the concept class $\mathcal{C}$ to denote the set of all monotone disjunctions of length $\leqslant k$ over $x_1, \dots, x_n$ and the domain $X = \{0,1\}^n$. We have shown that we can learn this class with

- Winnow in the OLMB model with mistake bound $M = \mathcal{O}(k \ln n)$.
- OLMB to PAC conversion with sample complexity $m = \frac{k \log n}{\varepsilon} \ln \frac{k \log n}{\delta}$.

But this is not too nice, because the hypothesis we're getting out in a linear threshold function hypothesis (instead of say, a length-k disjunction—this is the concept class we're learning, so this would be proper learning).

Now we will use Occam's razor to PAC-learn with sample complexity $\frac{k \log n}{\varepsilon} \ln \frac{k \log n}{\delta}$, but now using "pretty short" monotone disjunctions as hypotheses. The idea is that we'll show how, given m examples, we can efficiently find a pretty short—i.e. length-s where $s = k \log m$ monotone disjunction that is consistent with out m examples. Now here our $\mathcal{H}_m$ is all length-s monotone disjunctions, and $|\mathcal{H}_m| \leqslant n^s$, so that as long as $n^s \leqslant \delta e^{\varepsilon m}$, we can apply the Occam result. So we could be looking to have $n^{k \cdot \ln m} \leqslant \delta e^{\varepsilon m}$ and a small m would suffice.

§11.1.1 The set cover problem

<u>*Problem* SET COVER:</u>
Input: A family of r sets $S_1, \dots, S_r$ with each $S_i \subseteq [m]$.
Output: A minimal subset of the S_i that covers all of $[m]$.

Example 11.1. Suppose $m = 7$ and the explicit sets are

$$\begin{aligned} S_1 &= \{1,2,3,4,5\} \\ S_2 &= \{1,3,5,7\} \\ S_3 &= \{2,3,4,6\} \\ S_4 &= \{3,4,7\}. \end{aligned}$$

Then the optimal solution is to take $S_2 \cup S_3 = [7]$.

The set cover problem is NP-complete; there's no polynomial time algorithm that always finds the optimal solution. However, the greedy algorithm is efficient and provably finds an almost-optimal solution. Here's the greedy algorithm/heuristic:

Algorithm (Greedy set cover):

1. Pick S_i such that $|S_i|$ is as large as possible. Delete all the elements in S_i from every other S_j.
2. Repeat using the new S_j. Stop when all $[m]$ are covered.

Example 11.2. If we run the greedy heuristic on the previous example, then after the first half of the second step we're left with:

$$\begin{aligned}S_1 &= \{1,2,3,4,5\}\\ S_2 &= \{7\}\\ S_3 &= \{6\}\\ S_4 &= \{7\}.\end{aligned}$$

So we can choose S_2, S_3, or S_4 to cover the remaining elements. Say we choose S_2. Then the third set is S_3.

Theorem 11.2. *The greedy heuristic always finds a solution that contains $\leqslant \mathsf{OPT} \cdot \ln m$ sets, where OPT is the size of the optimal solution.*

Proof. A key fact for the following proof is that for any $U \subseteq [m]$ of uncovered elements at each round, there is always some S_j with $1 \leqslant j \leqslant r$ covering at least $|U|/\mathsf{OPT}$ points in U. This is clearly true in hindsight: some OPT many S_i cover $[m]$ up to cover U in total, so that one of them covers at least $|U|/\mathsf{OPT}$ elements.
Now, let $U_i \subseteq [m]$ be the set of elements not yet covered after i stages. The fact above says that the next step covers at least $|U_i|/\mathsf{OPT}$ elements of U_i (whichever S_j of the original sets cover this many of U_i elements still covers them; uncovered elements don't get removed as we go). So we have that

$$|U_{i+1}| \leqslant |U_i| - \frac{|U_i|}{\mathsf{OPT}} = |U_i|\left(1 - \frac{1}{\mathsf{OPT}}\right) \implies |U_i| \leqslant m \cdot \left(1 - \frac{1}{\mathsf{OPT}}\right)^i.$$

So if $i = \mathsf{OPT} \cdot \ln m$,

$$|U_i| \leqslant m \cdot \left(1 - \frac{1}{\mathsf{OPT}}\right)^{\mathsf{OPT}\cdot \ln m} < m \cdot e^{-\ln m} < 1,$$

since $(1-x)^{1/x} < e^{-1}$ so that $|U_i| = 0$, and the proof is done. □

Let's return back to the learning game. Why is this useful for learning sparse monotone disjunctions? It will let us find, given a set S of m examples labelled according to some monotone disjunction of size k, a two-stage algorithm that returns a hypothesis which has length $\leqslant k \cdot \ln m$ that is consistent. Here's the algorithm:

1. *Run the elimination algorithm on sample S.* (Start with $x_1 \vee \ldots \vee x_n$; for each negative example in S_1, eliminate all the x_i that are 1 in the example.)
   This gives a monotone disjunction h_1 consistent with the sample S; say is is $x_1 \vee \ldots \vee x_r$. (Note that any subdisjunction of $x_1 \vee x_r$ is consistent with all negative examples.)
2. We want to find a small subdisjunction of $x_1 \vee \ldots \vee x_r$ that is consistent with all $\leqslant m$ positive examples—that is, we want to find a small subset of $x_1 \vee \ldots x_r$ covering all the m positive

examples.
This is exactly the set cover problem! The x_i correspond to the S_i, and the elements $i \in [m]$ correspond to the positive examples. So the second step is simply to *run greedy set cover on the positive examples.*

We know that greedy set cover is guaranteed to find a collection of $\leqslant \mathsf{OPT} \cdot \ln m$ many variables x_i which cover everything. We know that $\mathsf{OPT} \leqslant k$ by definition (as we are learning a monotone disjunction of size k by the definition of the concept class).

Therefore the algorithm has the property that, given a set S of m example labels by a monotone disjunction of size k, it finds a monotone disjuhnction of size $\leqslant k \cdot \ln m$. So $\mathcal{H}_m$, the hypothesis class of all monotone disjunctions of size $\leqslant k \cdot \ln m$, is a subset of $\mathcal{H}$ with size

$$|\mathcal{H}_m| = \binom{n}{0} + \binom{n}{1} + \ldots + \binom{n}{k \cdot \ln m} \leqslant n^{k \cdot \ln m}.$$

So as long as $|\mathcal{H}_m| \leqslant \delta e^{\varepsilon m}$, and since $|\mathcal{H}_m| \leqslant n^{k \cdot \ln m}$, we have a PAC learner using m examples. How big does m need to be to satisfy the requirement $n^{k \cdot \ln m} \leqslant \delta e^{\varepsilon m}$? It turns out that taking

$$m \geqslant 2 \cdot \left(\frac{1}{\varepsilon} \cdot k \cdot \ln m \cdot \ln \frac{1}{\delta}\right) \ln \left(\frac{1}{\varepsilon} \cdot k \cdot \ln n \cdot \ln \frac{1}{\delta}\right)$$

is enough. This is because by taking logarithms of $n^{k \cdot \ln m} \leqslant \delta e^{\varepsilon m}$, we get $k \cdot \ln m \cdot \ln n \leqslant \varepsilon m + \ln \delta$, and so we need

$$k \cdot \ln m \cdot \ln n + \ln \frac{1}{\delta} \leqslant \varepsilon m.$$

Since $A + B \leqslant AB$ for $A, B \geqslant 2$, it suffices to take

$$\frac{k \cdot \ln n \cdot \ln \frac{1}{\delta}}{\varepsilon} \leqslant \frac{m}{\ln m}.$$

Suppose that $A = \frac{k \cdot \ln n \cdot \ln \frac{1}{\delta}}{\varepsilon}$ so that $A \leqslant \frac{m}{\ln m}$. What do we take m to be? Can we take $m = A \ln A$? No—this is not good enough; if it were, then we would have

$$A \leqslant \frac{A \ln A}{\ln (A \ln A)} = \frac{A \ln A}{\ln A + \ln \ln A} < \frac{A \ln A}{\ln A} = A,$$

which would imply that $A < A$, which is false. However, if we pick $m = 2A \ln A$, then we have that

$$A \leqslant \frac{2A \ln A}{\ln (2A \ln A)} = \frac{2A \ln A}{\ln A + \ln \ln A + \ln 2} < A,$$

which is true since

$$\frac{\ln A^2}{\ln (2A \ln A)} > 1.$$

§11.2 Proper vs improper learning

The main moral lesson here is that sometimes proper learning makes learning hard.

Consider the concept class $\mathcal{C}$ of all 3-term DNFs over n variables, $T_1 \vee T_2 \vee T_3$, where each T_i is some conjunction. We can PAC-learn these efficiently with 3-CNF hypotheses. First recall that the class of all monotone conjunctions over N variables is PAC-learnable with

$$m = \mathcal{O}\left(\frac{1}{\varepsilon}\left(N + \ln\frac{1}{\delta}\right)\right)$$

examples in $\mathrm{poly}(N, \frac{1}{\varepsilon}, \ln\frac{1}{\delta})$ time. (This is by the elimination algorithm and the discussion about consistent hypothesis finders.)

Claim 11.3. *Any 3-term DNF $T_1 \vee T_2 \vee T_3$ can be written as a 3-CNF $C_1 \wedge C_2 \wedge C_3$ with clause C_i with $\leqslant 3$ literals.*

Proof. This is a consequence of De Morgan's laws—see Homework 1 Problem 1. $\square$

Next time: Finish this argument about hard proper learning and easy improper learning.

§12 Lecture 12—16th October, 2023

Last time.

- Application of Occam's razor theorem: learning sparse disjunctions with few examples, with a hypothesis that is a pretty sparse disjunction, via a "greedy set cover" heuristic.
- Started proper vs improper learning. Moral lesson: sometimes it pays more to have improper learning algorithms as opposed to proper learning algorithms.

Today.

- Finish proper vs improper learning with some bad news: we cannot efficiently properly learn 3-term DNF formulas unless $\mathsf{NP} \subseteq \mathsf{RP}$. ☹
- Start unit on the sample complexity of PAC learning (via the VC dimension).

Midterm: Monday October 23rd, in-person, closed book, closed notes, no calculators, no electronic devices.

---

§12.1 Improper learning of 3-term DNFs is computationally easy

Recall the following proposition which we started the last time:

Proposition 12.1 ([PV88]). *The concept class of 3-term DNFs is PAC-learnable using the hypothesis class of 3-CNFs.*

This is a strange result! What is intuitively going on here is that when we are restricted to 3-CNFs in the hypothesis space we are essentially stuck in a straitjacket that allows us from finding a good approximation, but the concept is more easily learnable via a richer/more expressive representation. Let's give an outline for the proof of this proposition. Recall the following facts from last time:

Fact 12.2. 1. *Any 3-term DNF $T_1 \vee T_2 \vee T_3$ can be written as a 3-CNF $C_1 \wedge C_2 \wedge C_3$ with clause C_i with $\leqslant 3$ literals.*

2. *Any monotone conjunction over N variables is PAC-learnable with $m = \mathcal{O}\left(\frac{1}{\varepsilon}\left(N + \ln\frac{1}{\delta}\right)\right)$ examples in $\mathrm{poly}(N, \frac{1}{\varepsilon}, \ln\frac{1}{\delta})$ time.*

We can combine both these facts to get our efficient learning algorithm for learning 3-term DNFs using 3-CNFs as hypotheses. The basic idea for our argument is essentially to reduces the entire learning problem to that of learning N-variable monotone conjunctions, where $N = \Theta(n^3)$. Why is that? It's because the target c (a 3-term DNF) is equivalent by the first part of the fact above to some 3-CNF over $\{0,1\}^n$. But a 3-CNF is just an AND of some number of different length-3 clauses, i.e. an AND of $\ell_1 \vee \ell_2 \vee \ell_3$ clauses, where each ℓ_i is a literal over $x_1, \ldots, x_n$. The special fact we will use here is that there are "not too many" possibilities for what one of the clauses looks like. There are at most $2n$ possibilities for each of the first, second, and third literals, and so there are at most $(2n)^3 = 8n^3$ possible length-3 clauses. So we can simply view each of these clauses as being some Boolean variable $y_1, \ldots, y_{8n^3}$, which can be written down in poly-time.

Now define $y_1, \ldots, y_N$ (where $N = 8n^3$) new variables, one for each clause. Now explicitly convert each $(x, c(x)) \sim \mathrm{EX}(c, \mathcal{D})$ to a sample $(y, c'(y))$ over the new expanded variables $y_1, \ldots, y_N$ and for a target monotone conjunction c' over the y_j. Note that the draws $(y, c'(y))$ will be distributed according to some distribution $\mathcal{D}'$ over $\{0,1\}^N$, but this is fine—PAC learning works regardless of the distribution we chose; there just has to be some distribution $\mathcal{D}'$ over $\{0,1\}^N$ such that $(y, c'(y)) \sim \mathcal{D}'$. So we may just run the conjunction learning algorithm over the y_j (or equivalently over length-3 clauses on the original x_i literals). We would then get some high-accuracy hypothesis $h(y)$, which, with probability $1 - \delta$ is ε-accurate with respect to $\mathcal{D}'$. As we know the form of $h(y)$ and the fact that the conjunction learning algorithm is proper, the hypothesis will be a monotone conjunction over the y_j (or equivalently, a 3-CNF over the original x_i literals). So we can just output $h(y)$ as our hypothesis for the original 3-term DNF $c(x)$. By so doing, we would have efficiently learned 3-term DNFs with high accuracy using 3-CNFs as hypotheses. (Note that as all of this is merely a translation, the accuracy of the 3-CNF over the original x_i literals is the same as the accuracy of the $h(y_j)$ over $\mathcal{D}'$.)

§12.2 Proper learning of 3-term DNFs is computationally hard

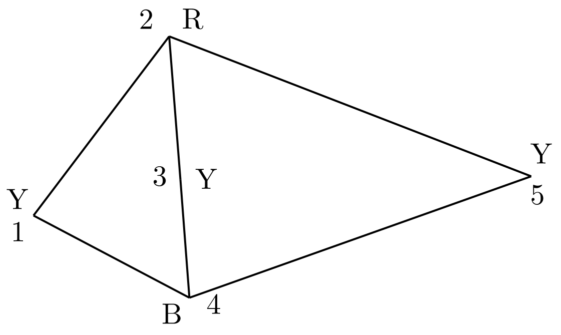
We now show that it is NP-hard to learn 3-term DNFs properly. Our main proposition is the following:

Proposition 12.3 ([PV88]). *Suppose that there exists an efficient $\mathrm{poly}(n, \frac{1}{\varepsilon}, \frac{1}{\delta})$-time proper PAC learning algorithm for 3-term DNFs. Then there exists an efficient $\mathrm{poly}(n)$-time on an n-node graph to solve the* GRAPH 3-COLOURABILITY *problem, which then implies* $\mathsf{NP} \subseteq \mathsf{RP}$ *(where* RP *is the complexity class of poly-time randomised algorithms with one-sided error).*

Problem GRAPH 3-COLOURABILITY:
Input: A graph $G = (V, E)$ with $|V| = n$.
Output: A *legit* colouring[a] of G using the three colours red (R), yellow (Y), and blue (B).

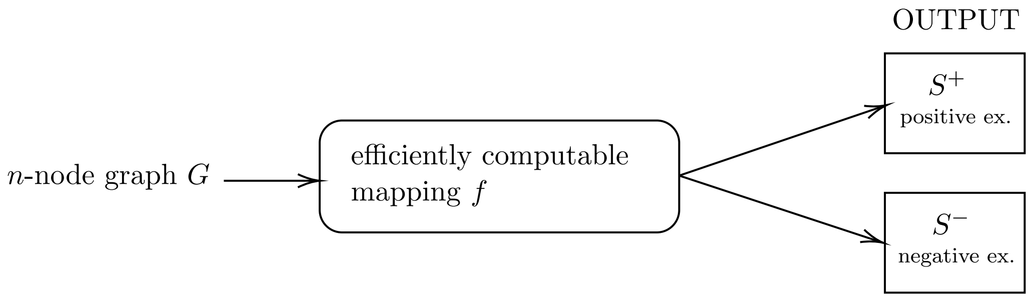
[a]Really we are concerned with whether this colouring *exists* rather than the explicit colouring, but we'll be a bit imprecise for now.

By a *legit* colouring, we mean a colouring of the vertices of G such that no two adjacent vertices have the same colour. Here's an example of an output of some algorithm for this problem (note that for easy readability, we have labelled the vertices with the letters R, Y, B instead of colours, and have distinguished the vertices using the numbers from 1 to 5 inclusive):



This problem is NP-complete; there is almost certainly no poly(n)-time (possibly randomised algorithm) to solve it.

Let's get back to learning theory. The key proof ingredient for us is some mapping f that works like this:



This mapping f has two key properties:

1. f is efficiently computable; it runs in poly(n)-time, so S has size poly(n).
2. f faithfully transforms 3-colourable graphs (which it takes as inputs) into outputs (data sets) that are consistently labelled by some 3-term DNF, and does this in an if-and-only-if way; i.e. for every graph G, G is colourable iff there exists some 3-term DNF labelling all examples in S^+ as positive and all examples in S^- as negative.

Assume for now that this mapping indeed exists (we'll construct it later). Then by the assumption in the theorem, we have an algorithm A which is an efficient proper PAC learner for 3-term DNFs. Here's our algorithm to decide the GRAPH 3-COLOURABILITY problem:

Algorithm A to decide GRAPH 3-COLOURABILITY:

1. Take a graph $G = (V, E)$ as input.

2. Apply f to G to get a data set $S = S^+ \cup S^-$ of labelled (positive or negative) examples. We know by the guarantees of f that G is 3-colourable iff the collection of labelled examples is consistent with some 3-term DNF c.

3. Let $\mathcal{D}'$ be the distribution over labelled examples which is uniform over S. Let $\varepsilon = 1/(2|S|)$ and $\delta = 0.01$. Nothing new here; just standard PAC learning stuff.
   Run A with (ε, δ) and examples drawn from $\mathcal{D}'$ to get a hypothesis h that is a 3-term DNF (a A is a proper PAC learner). Note that we have to toss coins to simulate draws from $\mathcal{D}'$.

4. Check whether h agrees with every labelled example in S. If so, output "G is 3-colourable"; otherwise output "G is not 3-colourable".

Is this algorithm efficient? Yes it is. The first step is efficient by assumption. The second step is efficient, because we said that f is efficient by construction. The third step is efficient, as $\varepsilon = 1/\text{poly}(n)$, and since $|S| \leqslant \text{poly}(n)$, δ is constant, and A is efficient by assumption. The fourth step is efficient, as we can just loop through all of the n labelled examples in $\text{poly}(n)$ time to check whether h agrees with all of them. So the entire algorithm is efficient.

Is this algorithm correct? Yes it is, but this is significantly more involved. Suppose G is 3-colourable. Then S is consistent with some 3-term DNF, which means that we *legitimately* simulated some PAC learning algorithm; the distribution $\mathcal{D}$ was a valid simulation of an example oracle $\text{EX}(c, \mathcal{D})$ for A. So with probability $\geqslant 99\%$, the output hypothesis h of A is a 3-term DNF with error $\leqslant \varepsilon = 1/(2|S|)$. Therefore this h is perfect on S—even one error would mean that the error is $\geqslant 1/|S| > \varepsilon$—and so h agrees with every labelled example in S and the algorithm outputs "G is 3-colourable".
On the other hand, suppose G is not 3-colourable. Then S is not consistent with any 3-term DNF (by the guarantee of f), so that with high probability the 3-term DNF hypothesis h is surely not consistent with S, and in the third step we can surely say that h is not consistent with S. So the algorithm outputs "G is not 3-colourable".

It remains then to give f and argue that it has the claimed properties. Here's the mapping f:

$f =$ "On input $G = (V, E)$, where $V = [n]$ and $E = (i, j)$ m-many pairs of vertices,

1. for $i, j \in [n]$, the output $f(G)$ is (S^+, S^-), where

2. S^+ is the collection of n n-bit examples in $\{0,1\}^n$, where each example has all 1's except for the ith bit, which is 0. (So S^+ is the collection of all examples where the ith bit is 0.)

3. S^- is the collection of m n-bit examples in $\{0,1\}^n$, where each example has variables that are all 1's except for the ith and jth bits, which are 0. (So S^- is the collection of all examples where the ith and jth bits are 0.)"

Example 12.4. Suppose the mapping f recieves the graph from earlier as input. Then the output is

$$S^+ = \{01111, 10111, 11011, 11101, 11110\}$$
$$S^- = \{00111, 01101, 10011, 10110, 11001, 11100\}.$$

We will now show that if a given input graph G to the mapping f is 3-colourable, then there is a 3-term DNF c that is consistent with S^+ and S^-, and vice versa.

Lemma 12.5. *1. If G is 3-colourable, then there exists a 3-term DNF c that is consistent with S^+ and S^-.*

2. *If there exists a 3-term DNF c that is consistent with S^+ and S^-, then G is 3-colourable.*

Proof. 1. Let R, B, Y be some legit 3-colouring of G. Let T_R be the conjunction that contains x_i for every non-red vertex i. Similarly define T_B and T_Y. Then in the example from above, we would have $T_R = x_1 \wedge x_3 \wedge x_4 \wedge x_5$, $T_Y = x_2 \wedge x_4$, and $T_B = x_1 \wedge x_2 \wedge x_3 \wedge x_5$. (Note that an equivalent view is that T_R is only missing the literals corresponding to the red vertices, and similarly for T_B and T_Y.)
We claim that each example in S^+ is labelled positive by $c = T_R \vee T_B \vee T_Y$. Consider the ith positive example $11\cdots101\cdots1$ in S^+. Say it is red in the colouring (i.e. where $i = 0$). Then T_R is missing the literal x_i, so that example is labelled positive by T_R. Similarly, if it is blue or yellow, then it is labelled positive by T_B or T_Y respectively.
On the other hand, we claim that each example in S^- is labelled negative by c. Consider an arbitrary example $\{i,j\}$ in S^-, $11\cdots101\cdots101\cdots1$, where the ith and jth bits are 0. Does this satisfy T_R? No; if it did, this would mean that T_R is missing x_i and x_j, which would mean that i and j are both red, which is impossible for a legit colouring as i and j are adjacent. Similarly, it does not satisfy T_B or T_Y as i and j are adjacent and so cannot both be blue or both be yellow. So the example is labelled negative by c.

2. Let the 3-term DNF consistent with S^+ and S^- be $c = T_R \vee T_B \vee T_Y$. Since it is consistent with S^+, every positive example (corresponding to the specified vertices of the graph) is accepted by some term. Now consider the colouring which colours vertex i of the graph either red, yellow, or blue, depending on whether T_R, T_Y, or T_B accepts the ith positive example $11\cdots10\cdots1$ (if there are ties, then we break them arbitrarily).
Is this a legit colouring? Happily the answer is indeed yes; we claim that if $\{i,j\}$ is an edge, then the above colouring does not colour i and j the same colour. Suppose the colour we are considering is red (the same argument follows for blue and yellow). We want to show that i and j are not both satisfied by T_R. Consider the edge $\{i,j\} = \{1,2\}$. Then 01111 and 10111 are both positive examples from vertices 1 and 2 respectively, and 00111 is a negative example from edge $\{1,2\}$. Now, x_1 must not be in T_R, or else 01111 would be a negative example. Also, $\overline{x}_1$ must not be in T_R, or else 10111 wouldn't be a positive example. Similarly, both x_2 and $\overline{x}_2$ cannot be in T_R. But then if T_R accepts the two positive examples, it would have to accept the negative example 00111, which it doesn't accept; contradiction. Therefore i and j are not both red (i.e. satisfied by T_R), and so the colouring is legit.

The proof is complete. □

Next time: Sample complexity of PAC learning.

§13 Lecture 13—18th October, 2023

Last time.

- Finished proof that we can efficiently improperly learn 3-term DNFs using 3-CNFs as hypotheses.
- Showed that we cannot efficiently properly learn 3-term DNFs unless $\mathsf{NP} \subseteq \mathsf{RP}$.

Today. Start unit on the sample complexity of PAC learning (via the VC dimension) for concept class $\mathcal{C}$.

- Lower bound on the sample complexity of PAC learning: $\Omega(\mathrm{VCDIM}(\mathcal{C})/\varepsilon)$.
- Upper bound on the sample complexity of PAC learning: roughly $\mathcal{O}(\mathrm{VCDIM}(\mathcal{C})/\varepsilon)$.

Next time: Midterm—Monday October 23rd, in-person, closed book, closed notes, no calculators, no electronic devices.

---

§13.1 Sample complexity of PAC learning

For the rest of this unit we will not care much about the running time of our learning algorithms, but rather about the number of examples they need to see to learn a concept class $\mathcal{C}$ to a given accuracy ε and confidence δ.

Motivation. Given some nice concept class $\mathcal{C}$, we may want to answer the most basic questions about this class. For example, is $\mathcal{C}$ PAC learnable (from a finite number of examples)? If so, how many examples do we need to see to learn $\mathcal{C}$ to a given accuracy ε and confidence δ? If not, what is the best we can do?

Suppose $\mathcal{C}$ is finite, i.e. $|\mathcal{C}| < \infty$. By Occam's theorem (see Theorem 10.2), we have half an answer; if $\mathcal{C}$ is finite, then we can PAC learn with $\mathcal{O}(\frac{1}{\varepsilon} \ln \frac{|\mathcal{C}|}{\delta})$ examples via a consistent hypothesis finder for $\mathcal{C}$ using $\mathcal{H} = \mathcal{C}$. In other words, we can just brute-force search over all the hypotheses; if you drew $\mathcal{O}(\frac{1}{\varepsilon} \ln \frac{|\mathcal{C}|}{\delta})$ examples from the oracle, then with high probability you will find a consistent hypothesis. But this is an upper bound. What about a lower bound? How many examples do we need to see before we are in the realm of possibility for learning $\mathcal{C}$? In fact, what if $|\mathcal{C}| = \infty$? Are we completely hopeless?

Fortunately the answer to that last question is no. We will see that if $\mathcal{C}$ is infinite, then we can still learn $\mathcal{C}$ with a finite number of examples, provided that $\mathcal{C}$ is "not too complicated". In particular, we will see that the entire story is determined by the VC dimension of $\mathcal{C}$. For this unit, we will fix δ to be some constant, say $\delta = 1/10$ (this doesn't affect our analysis much, as we will soon come to learn). We first present the main results of this unit of the class. Fix any $\mathcal{C}$, and let $d := \mathrm{VCDIM}(\mathcal{C})$.

1. ☹ Any PAC learning algorithm for $\mathcal{C}$ must use $\Omega(d/\varepsilon)$ examples.
   We will prove this by showing that there is some "hard" distribution such that unless we get d/ε examples, we cannot successfully PAC learn. Note that this doesn't mean that we must use d/ε examples for *every* distribution; it just means that we can stipulate some distribution for which learning is hard until we get d/ε examples. This means, by the way, that if $d = \infty$, then we cannot PAC learn $\mathcal{C}$ with finite examples.

2. ☺ (Fundamental theorem of statistical learning theory.) If d is finite, then $\mathcal{C}$ can be PAC-learned with a consistent hypothesis finder for $\mathcal{C}$ using $\mathcal{C}$ with $\approx \mathcal{O}(d/\varepsilon)$ examples.
   This is like the upper bound we saw for finite $\mathcal{C}$ earlier in the class (see Theorem 10.2), but better in that it tells us explicitly that the VC dimension of $\mathcal{C}$ is the right parameter to examine, not the size of $\mathcal{C}$. (As we recall, we needed $|\mathcal{C}|$ as a factor to defeat a union bound over all of the possible bad hypothesis or concepts in the class; we will come to learn that the VC dimension captures all the power we actually used.)

We start today with the bad news:

§13.1.1 Lower bound on the sample complexity of PAC learning

Theorem 13.1. *Fix any concept class $\mathcal{C}$, and let $d := \mathrm{VCDIM}(\mathcal{C})$. Then any PAC learner A for $\mathcal{C}$ that learns to error ε and confidence $\delta = 1/10$ must use $\Omega(d/\varepsilon)$ examples.*

Proof. First we give a distribution $\mathcal{D}$ such that $\Omega(\mathcal{D})$ examples are needed to achieve ($\varepsilon = \frac{1}{8}, \delta = \frac{1}{8}$)-PAC learning. Let $S = \{x^{(1)}, \ldots, x^{(d)}\}$ be shattered by $\mathcal{C}$. There is a hard, uninformative distribution, namely the distribution that spreads all its probability mass uniformly over all the examples in $\mathcal{D}$, so that there is $1/d$ probability for each example in S and 0 probability for all other examples. Suppose now that our learning algorithm A only gets $d/2$ calls to the oracle $\mathrm{EX}(c, \mathcal{D})$. After these calls, A has only seen labels of $\leqslant d/2$ examples, and therefore it has not seen the labels of $> d/2$ examples.
Let the target concept $c \in \mathcal{C}$ be chosen by picking it uniformly at random from the 2^d concepts that shatter S—it makes sense to do this because of the intuition we have from the VC dimension. The label of each example of the 2^d concepts is chosen by flipping a coin with probability $1/2$ of heads. We then have that the expected error over the uniform choice of the target concept is

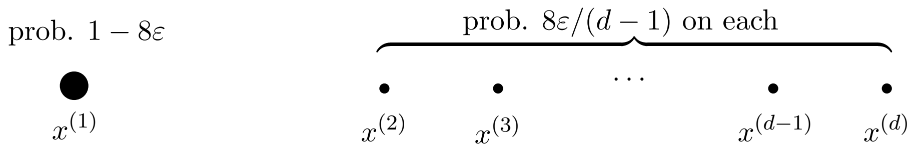
$$\mathbb{E}[\text{error of } A\text{'s hypothesis}] \geqslant \frac{1}{2} \cdot \frac{d}{2} \cdot \frac{1}{d} = \frac{1}{4},$$

since $1/2$ is the error on each unseen point (via the coin flip), there are at least $d/2$ unseen points, and each unseen point has probability $1/d$ of being chosen. So the learner is in a bad situation; its expected error is at least $1/4$. We will now do a little bit of probability lawyering to show that we cannot have a winning (ε, δ)-PAC learner.
Let the accuracy α be $\alpha = 1 - \varepsilon_A$, where ε_A is the error of the hypothesis h output by A. Then we have that $\mathbb{E}[\alpha] \leqslant 3/4$. By Markov's inequality, and noting that $\frac{7}{8} = \frac{3}{4} \cdot \frac{7}{6}$, we have that $\Pr[\alpha \geqslant 7/8] \leqslant \frac{6}{7}$. Therefore we have that

$$\Pr\left[\alpha < \frac{7}{8}\right] \geqslant \frac{1}{7} \implies \Pr\left[\varepsilon_A > \frac{1}{8}\right] \geqslant \frac{1}{7} > \frac{1}{8} = \delta,$$

which would mean that for our selection of ε and δ, $\Pr[\varepsilon_A > \varepsilon] > \delta$, which would mean that A is not a (ε, δ)-PAC learner.

Let us now prove the actual theorem. We have to give a hard distribution $\mathcal{D}'$ such that $\Omega(\mathcal{D}/\varepsilon)$ examples are required to get error ε and confidence $1 - \delta = 0.9$. Again let the shattered set be $S = \{x^{(1)}, \ldots, x^{(d)}\}$. Let $\mathcal{D}'$ be the probability distribution that assigns significant probability mass on $x^{(1)}$ and spreads the rest among $x^{(2)}, \ldots, x^{(d)}$:



The idea here is that it is very easy to achieve "high" $(1 - 8\varepsilon)$ accuracy on $x^{(1)}$, but getting 7/8 of the remaining is slow; on average we need $1/(8\varepsilon)$ draws to get one example from the crucial set $S \setminus \{x^{(1)}\}$, causing a slowdown of $\Omega(1/\varepsilon)$ on top of the earlier average.
Now suppose the learning algorithm A only gets $(d-1)/2$ examples from the crucial set $\{x^{(2)}, \ldots, x^{(d)}\}$. Let the target concept $c \in \mathcal{C}$ be randomly chosen such that each of the 2^{d-1} labellings of the crucial

set are equally likely. As above, with probability $\geqslant 1/8$, the algorithm A has error rate $\geqslant 1/8$ on the crucial set. These have probability 8ε under D', so if A gets $\leqslant (d-1)/2$ examples from the crucial set, with probability $\geqslant 1/8$, A has error $\geqslant \frac{1}{8} \cdot 8\varepsilon = \varepsilon$.

Now all we need to do is argue that unless we get something like d/ε many examples, we're very likely to get $(d-1)/2$ examples from the crucial set. Each call to $\mathrm{EX}(c, \mathcal{D})$ hits the crucial set with probability 8ε. Suppose now that A makes $\frac{d-1}{32\varepsilon}$ calls to the $\mathrm{EX}(c, \mathcal{D})$. Then

$$\mathbb{E}\left[\text{number of times we hit the crucial set } S \setminus \{x^{(1)}\}\right] = \frac{d-1}{32\varepsilon} \cdot 8\varepsilon = \frac{d-1}{4}.$$

By the multiplicative Chernoff bound (with $\delta = 1$, $pm = (d-1)/4$), we get that

$$\Pr\left[X \geqslant (1+\delta)pm\right] \leqslant e^{-(d-1)/12}.$$

Without loss of generality, assume that $d \geqslant 100$, so that $e^{-(d-1)/12} \leqslant 1/100$. Then we have that with probability $\geqslant \frac{99}{100} \cdot \frac{1}{8} \geqslant \frac{1}{9}$, the error is $\geqslant \varepsilon$. When the dust settles, we find that if the learning algorithm only got $\frac{d-1}{32\varepsilon}$-many examples, then we only have at least a 1/9-th chance that the error rate is at least ε, which is to say that we did not successfully (ε, δ)-PAC learn with $\delta = 0.1$. □

§13.1.2 Upper bound on the sample complexity of PAC learning

Let's lay down the structure of this section first. The main result is that for any distribution $\mathcal{D}$, any hypothesis $h \in \mathcal{C}$ which is consistent with

$$\approx \frac{d}{\varepsilon} \log \frac{1}{\varepsilon} + \frac{1}{\varepsilon} \log \frac{1}{\delta}$$

many random examples from the oracle $\mathrm{EX}(c, \mathcal{D})$ is, with probability at least $1 - \delta$, ε-accurate, i.e. $\mathrm{err}_{\mathcal{D}}[h, c] \leqslant \varepsilon$. The proof will take several classes to complete; here's a high-level outline for the proof:

1. We'll start by pondering $\mathrm{VCDIM}(\mathcal{C})$. To get more information, we'll consider a function called the *growth function* of $\mathcal{C}$ which gives us the number of ways that the concept class $\mathcal{C}$ works with a dataset that is labelled according to the size of the concepts in $\mathcal{C}$.
2. Via a combinatorial argument, we will prove an Amazing Theorem (usually called the Sauer-Shelah-Perles lemma) about how the growth function behaves for every concept class $\mathcal{C}$.
3. Finally, we will give a learning argument that is a lot like that used in the CHF theorem; instead of a union bound over all the bad hypotheses, we will use a union bound over all the ways the concepts in our class can label a fixed dataset, aided by the Amazing Theorem.

§13.2 The growth function

Recall that $\mathrm{VCDIM}(\mathcal{C})$ is the size of the largest shattered set of the concept class $\mathcal{C}$. Here's another perspective on the VC dimension of a concept class $\mathcal{C}$. Suppose we have $S \subseteq X$, and define $\Pi_{\mathcal{C}}(S)$ to be the set of all labellings of S induced by the concepts in $\mathcal{C}$, that is, $\Pi_{\mathcal{C}}(S) = \{c \cap S : c \in \mathcal{C}\}$.

Example 13.1. Let $S = \{1, 2, 3\}$ and let $\mathcal{C}$ be the concept class of intervals of $\mathbb{R}$. We already know that $\text{VCDIM}(\mathcal{C}) = 2$. Then we see that

$$\Pi_{\mathcal{C}}(S) = \{\varnothing, \{1\}, \{2\}, \{3\}, \{1, 2\}, \{2, 3\}, \{1, 2, 3\}\}.$$

Note that $\mathcal{C}$ shatters S if and only if $|\Pi_{\mathcal{C}}(S)| = 2^{|S|}$.

The above example motivates the following definition:

Definition 13.2 (Growth function)**.** *The growth function of $\mathcal{C}$, denoted $\Pi_{\mathcal{C}}(m)$, is defined to be the size of the largest $\Pi_{\mathcal{C}}(S)$ for every subset S satisfying $|S| = m$. That is,*

$$\Pi_{\mathcal{C}}(m) := \max_{S \subseteq X, |S| = m} |\Pi_{\mathcal{C}}(S)|,$$

where X is the domain of $\mathcal{C}$.

Next time: Midterm, in class, no notes.

§14 Lecture 14—Midterm Exam, 23rd October, 2023

Today was a midterm—we sat, we online-learned concept classes, we shattered weird sets, and we departed, hollow and empty, ready to be recharged by more PAC learning.

§15 Lecture 15—25th October, 2023

Last time. Started unit on the sample complexity of PAC learning (via the VC dimension) for concept class $\mathcal{C}$.

- Lower bound on the sample complexity of PAC learning: $\Omega(\text{VCDIM}(\mathcal{C})/\varepsilon)$.
- Upper bound on the sample complexity of PAC learning: roughly $\mathcal{O}(\text{VCDIM}(\mathcal{C})/\varepsilon)$.

Today.

- Ponder even more about $\text{VCDIM}(\mathcal{C})$.
- Introduce an Amazing Theorem about how the growth function behaves for any concept class $\mathcal{C}$.

§15.1 The growth function and an Amazing Theorem

Recall that the shatter function $\Pi_{\mathcal{C}}(S)$ of a set of examples S is defined as $\Pi_{\mathcal{C}}(S) = \{c \cap S : c \in \mathcal{C}\}$, which is essentially a shattering of S into all of the different subsets of S that are induced by concepts in the concept class $\mathcal{C}$, a collection of sets. The growth function

$$\Pi_{\mathcal{C}}(m) := \max_{S \subseteq X, |S| = m} |\Pi_{\mathcal{C}}(S)|$$

on the other hand is the largest size of the output of the shatter function on any input of size S. In other words, it is the maximum number of ways that concepts in the class $\mathcal{C}$ can label any given subset of m examples. Therefore, in this language, the VC dimension of a concept class $\mathcal{C}$ is the largest value m such that $\Pi_{\mathcal{C}}(m) = 2^m$.

Example 15.1 (Single intervals of $\mathbb{R}$)**.** Let $\mathcal{C}$ be the concept class of single intervals of $\mathbb{R}$. Then we know that $\Pi_{\mathcal{C}}(1) = 2$ (since a single point is either labelled $+$ or $-$), and $\Pi_{\mathcal{C}}(2) = 4$ (since two points can be labelled $++$, $+-$, $-+$, or $--$), but $\Pi_{\mathcal{C}}(3) = 7 < 8$ (since the labelling $+-+$ is not achievable by a closed interval).
Now consider the case of general m. There is one way to classify all of the points as negative (just have it not include any of the m points). Likewise, if we need to have one point as positive, we only need to have the interval include that single point. Finally, there are $\binom{m}{2}$ ways of labelling two or more points of S positive, since there are 2 ways of choosing the leftmost and rightmost endpoints of the m points in the interval and labelling them positive (so that any other point is labelled negative). Therefore,

$$\Pi_{\mathcal{C}}(m) = 1 + m + \binom{m}{2} = \binom{m}{0} + \binom{m}{1} + \binom{m}{2} = \sum_{i=0}^{2} \binom{m}{i},$$

which is not a coincidence, as we soon see.

This is the tip of a very large iceberg, at the start of which is the Amazing Theorem. The proof is a bit involved—one can be forgiven for thinking that human beings weren't meant to do things like this but instead to swing through the trees and crack nuts—but we will proceed nonetheless:

Theorem 15.1 (Amazing Theorem)**.** *Let $\mathcal{C}$ be a concept class with* $\mathrm{VCDIM}(\mathcal{C}) = d$*, Then:*

- *if $m \leqslant d$, then $\Pi_{\mathcal{C}}(m) = 2^m$ (as expected),*
- *if $m > d$, then $\Pi_{\mathcal{C}}(m) \leqslant \left(\frac{em}{d}\right)^d$ (truly amazing!)*

Proof. Let us ignore $\mathcal{C}$ for just a moment. Define $\Phi_d(m)$ for $d, m \geqslant 0$ as $\Phi_0(m) = \Phi_d(0) = 1$ and for $d, m \geqslant 1$, $\Phi_d(m) = \Phi_{d-1}(m) + \Phi_{d-1}(m-1)$. We will now construct a table whose entries are $\Phi_d(m)$ for $d, m \geqslant 0$.
Given this table, we claim that

$$\Phi_d(m) = \binom{m}{0} + \ldots + \binom{m}{d} = \sum_{i=0}^{d} \binom{m}{i},$$

d						$\cdots$	$\Phi_d(m)$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
4	1	2	4	8	16	$\cdots$	$\iddots$
3	1	2	4	8	15	$\cdots$	$\iddots$
2	1	2	4	7	11	$\cdots$	$\iddots$
1	1	2	3	4	5	$\cdots$	$\iddots$
0	1	1	1	1	1	1	$\iddots$

and we will prove this claim by induction on both m and d. For our base cases we have that

$$\Phi_d(0) = \sum_{i=0}^{d} \binom{0}{i} = \binom{0}{0} = 1,$$

which agrees with the definition. Also,

$$\Phi_0(m) = \sum_{i=0}^{0} \binom{m}{i} = \binom{m}{0} = 1,$$

which also agrees with the definition. Now for the inductive step:

$$\begin{aligned}
\Phi_d(m) &= \Phi_d(m-1) + \Phi_{d-1}(m-1) \\
&= \sum_{i=0}^{d} \binom{m-1}{i} + \sum_{i=0}^{d-1} \binom{m-1}{i-1} \\
&= \sum_{i=0}^{d} \left[\binom{m-1}{i} + \binom{m-1}{i-1} \right] \qquad \text{since } \binom{m-1}{-1} = 0 \\
&= \sum_{i=0}^{d} \binom{m}{i} \qquad \text{by Pascal's identity } \binom{m-1}{i} + \binom{m-1}{i-1} = \binom{m}{i}.
\end{aligned}$$

Now we will show that

$$\Phi_d(m) = \sum_{i=0}^{d} \binom{m}{i} \begin{cases} = 2^m & \text{if } m \leqslant d \\ \leqslant \left(\frac{em}{d}\right)^d & \text{if } m > d. \end{cases}$$

Note now that if $m \leqslant d$, then

$$\sum_{i=0}^{d} \binom{m}{i} = \binom{m}{0} + \binom{m}{1} + \ldots + \binom{m}{m} + \binom{m}{m+1} + \ldots + \binom{m}{d}.$$

But since $m \leqslant d$,

$$\binom{m}{m+1} = \ldots = \binom{m}{d} = 0 \implies \binom{m}{m+1} + \ldots + \binom{m}{d} = 0,$$

and since

$$\binom{m}{0} + \binom{m}{1} + \ldots + \binom{m}{m} = 2^m,$$

we are done with the equality and can move on to the inequality. If $m < d$ (i.e. $d/m < 1$), then consider the penalised sum $\left(\frac{d}{m}\right) \cdot \sum_{i=0}^{d} \binom{m}{i}$. Observe that

$$\begin{aligned}
\left(\frac{d}{m}\right)^d \cdot \sum_{i=0}^{d} \binom{m}{i} &\leqslant \sum_{i=0}^{d} \left(\frac{d}{m}\right)^i \binom{m}{i} && \text{since } \left(\frac{d}{m}\right)^d \leqslant \left(\frac{d}{m}\right)^i \\
&\leqslant \sum_{i=0}^{m} \left(\frac{d}{m}\right)^i \binom{m}{i} && \text{adding stuff from } \frac{d+1}{m} \text{ to } \frac{m}{m} \\
&= \left(1 + \frac{d}{m}\right)^m && \text{by the binomial theorem} \\
&\leqslant \left(e^{d/m}\right)^m = e^d && \text{since } 1 + x \leqslant e^x \text{ for all } x,
\end{aligned}$$

which implies that

$$\sum_{i=0}^{d} \binom{m}{i} \leqslant \left(\frac{em}{d}\right)^d,$$

as desired.

Now let's get back to thinking about the concept class $\mathcal{C}$. We will show that for any concept class $\mathcal{C}$, we have that for all m, $\Pi_{\mathcal{C}}(m) \leqslant \Phi_d(m)$, and we will prove this by jointly inducting on m and d. For the base cases, since the maximum number of ways concepts in a concept class can label a set of zero examples is 1 (by assigning labels to any of them), we have that $\Pi_{\mathcal{C}}(0) = 1 = \Phi_d(0)$ as we have already seen. On the other hand, if $d = \text{VCDIM}(\mathcal{C}) = 0$, then it must be that $|\mathcal{C}| \leqslant 1$, so that $\Pi_{\mathcal{C}}(m) \leqslant 1 = \Phi_0(m)$ for all m. Now for the inductive step. Suppose that for all $m' < m$ and $d' < d$, we have that $\Pi_{\mathcal{C}}(m') \leqslant \Phi_{d'}(m')$. We will show that $\Pi_{\mathcal{C}}(m) \leqslant \Phi_d(m)$ for all m and d.

Let S be any subset of X with $|S| = m$. Suppose x contains concepts $c_1 = \{k_1, k_2, k_3\}$, $c_2 = \{x\}$, $c_3 = \{x, k_4, k_5, k_6, k_7\}$, and $c_4 = \{k_7, k_8, k_9, k_{10}\}$, while $c_1 \cap c_2 = c_1 \cap c_3 = \varnothing$, $c_2 \cap c_3 = \{x\}$, and c_4 is pairwise disjoint to all the other concepts. We claim that $\Pi_{\mathcal{C}}(S \setminus \{x\})$ undercounts $\Pi_{\mathcal{C}}(S)$ because of pairs of distinct sets in $\Pi_{\mathcal{C}}(S)$ which differ only on x. Now define $\mathcal{C}' = \{c \in \Pi_{\mathcal{C}}(S) : x \in c_1, c \cup \{x\} \in \Pi_{\mathcal{C}}(S)\}$, the collection of sets "like" c_1. Then $|\Pi_{\mathcal{C}}(S)| = |\mathcal{C}'| + |\Pi_{\mathcal{C}}(S \setminus \{x\})| \leqslant |\mathcal{C}'| + \Phi_d(m-1)$. But notice that $\mathcal{C}' = \Pi_{\mathcal{C}'}(S \setminus \{x\})$, because each set in $\mathcal{C}'$ is a subset of $S \setminus \{x\}$ already, and since $\Pi_{\mathcal{C}'}(S \setminus \{x\})$ is defined to be the set $\{c \cap (S \setminus \{x\}) : c$ ranges over all of $\mathcal{C}'\}$, which is exactly the same as $\mathcal{C}'$ itself. It follows then that

$$\begin{aligned}
|\Pi_{\mathcal{C}}(S)| &= |\mathcal{C}'| + \Pi_{\mathcal{C}}(S \setminus \{x\}) \\
&= |\Pi_{\mathcal{C}'}(S \setminus \{x\})| + \Pi_{\mathcal{C}}(S \setminus \{x\}).
\end{aligned}$$

As $\Pi_{\mathcal{C}}(S \setminus \{x\}) \leqslant \Phi_d(m-1)$, we only need to show that $|\Pi_{\mathcal{C}'}(S \setminus \{x\})| \leqslant \Phi_{d-1}(m-1)$ to finish the proof by the inductive hypothesis. The proof for this is not too complicated. Suppose that $S' \subseteq S \setminus \{x\}$, and notice that S' is shattered by C'. Then $S' \cup \{x\}$ is shattered by $\mathcal{C}$. But we know that $\text{VCDIM}(\mathcal{C}) = d$, so that $|S' \cup \{x\}| \leqslant d$, i.e. $|S'| \leqslant d-1$, which implies that $\text{VCDIM}(\mathcal{C}') \leqslant d-1$. As $|S \setminus \{x\}| = m-1$, it follows then that $|\Pi_{\mathcal{C}'}(S \setminus \{x\})| \leqslant \Phi_{d-1}(m-1)$, as desired.

All in all, we have shown that

1. $\Pi_{\mathcal{C}}(m) \leqslant \Phi_d(m)$,

2. Consequently from the above,
$$\Phi_d(m) = \sum_{i=0}^{d} \binom{m}{i} = \begin{cases} = 2^m & \text{if } m \leqslant d \\ \leqslant \sum_{i=0}^{d} \binom{m}{i} & \text{if } m > d \end{cases} = \begin{cases} = 2^m & \text{if } m \leqslant d \\ \leqslant \left(\frac{em}{d}\right)^d & \text{if } m > d, \end{cases}$$
and therefore the proof of the amazing theorem is done. □

§16 Lecture 16—30th October, 2023

Last time.

- Introduced and proved the amazing theorem—the Sauer-Shelah-Perles lemma—that helps us characterise a bound for the growth function of a concept class via its VC dimension.

Today.

- Finish final part of proof of the upper bound of PAC learning via the VC dimension.
- Application: Efficient PAC learning of LTFs over $\mathbb{R}^n$ or $\{0,1\}^n$.

§16.1 The upper bound on the sample complexity of PAC learning

Recall that we said that any concept (typically a hypothesis) $h \in \mathcal{C}$ is *bad* if,
$$\Pr_{x\sim\mathcal{D}}[h(x) \neq c(x)] \geqslant \varepsilon,$$
where c is the target concept. Also recall our old CHF theorem for finite concept classes $\mathcal{C}$:

Theorem 16.1. *Fix a concept class $\mathcal{C}$ and a hypothesis class $\mathcal{H}$, some unknown $c \in \mathcal{C}$, and some distribution $\mathcal{D}$ over X. Let $\left(x^{(1)}, c\left(x^{(1)}\right)\right), \dots, \left(x^{(m)}, c\left(x^{(m)}\right)\right)$ be m i.i.d. draws from $\mathrm{EX}(c,\mathcal{D})$, where*
$$m = \frac{1}{\varepsilon}\left(\ln\frac{1}{\delta} + \ln|\mathcal{H}|\right).$$
Let us say that h is bad if $\mathrm{err}_{\mathcal{D}}[h,c] > \varepsilon$. Then
$$\Pr\left[\textit{any bad } h \in \mathcal{H} \textit{ is consistent with draws } \left\{\left(x^{(i)}, c\left(x^{(i)}\right)\right)\right\}\right] \leqslant \delta.$$

Remember that the key to the proof of this theorem is developing a union bound over all the hypotheses $h \in \mathcal{H}$.

Our main gripe with this theorem is that it is very limiting; we want something for infinite concept classes so that we can do more things, e.g. learn intervals on the real line, etc. We want an infinite analogue, and we will indeed get one:

Theorem 16.2 (Key Theorem)**.** *Fix any possibly infinite concept class $\mathcal{C}$, any target $c \in \mathcal{C}$, and any distribution $\mathcal{D}$. Given m samples from $\mathrm{EX}(c,\mathcal{D})$, where*
$$m \geqslant \frac{2}{\varepsilon}\left(\ln\left(\Pi_{\mathcal{C}}(2m)\right) + \ln\frac{2}{\delta}\right), \quad m \geqslant \frac{8}{\varepsilon},$$
we can construct an (ε,δ)-PAC learner via a CHF for $\mathcal{C}$ using $\mathcal{C}$ on m examples.

Remark 16.1. *The main thing in the previous theorem was the fact that finite concept classes are learnable because, with enough examples, one may just run a CHF on them. This new theorem is saying that if m satisfies*

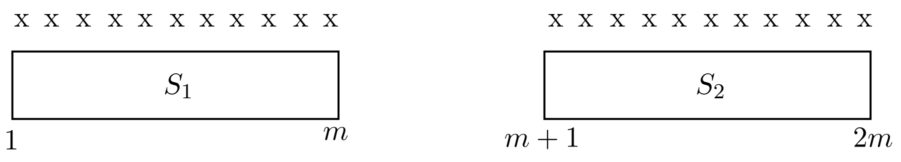
$$m \geqslant \frac{2}{\varepsilon}\left(\ln\left(\Pi_{\mathcal{C}}(2m)\right) + \ln\frac{2}{\delta}\right), \quad m \geqslant \frac{8}{\varepsilon},$$

then we can learn $\mathcal{C}$ with $\mathcal{C}$ on m examples. But why does the first condition on m change? If we didn't have the Amazing Theorem, we'd only know that $\Pi_{\mathcal{C}}(2m) \leqslant 2^{2m}$, so that we'd need $m \geqslant \frac{2}{\varepsilon}\left(2m + \ln\frac{8}{\varepsilon}\right)$, which is a lot worse than the condition we have now. Furthermore if it were something smaller, maybe even $\frac{1}{2}\cdot 2^{2m}$, then the above condition would never hold, as it says

$$m \geqslant \frac{2}{\varepsilon}\left(2m - 1 + \ln\frac{2}{\delta}\right),$$

which will never hold! But thanks to the Amazing Theorem, we have better control; $\Pi_{\mathcal{C}}(2m) \leqslant \left(\frac{e\cdot 2m}{d}\right)^d$, and so Theorem 16.2 holds for m "not too large."

Proof of Theorem 16.2. Consider the following procedure: draw $2m$ samples from the oracle $\mathrm{EX}(c, \mathcal{D})$, call the first m samples S_1, and the second S_2:



Now define two events: event A which specifies whether there exists a bad $h \in \mathcal{C}$ that is consistent with S_1 (our eventual goal in this argument is to show that $\Pr[A]$ is "low"), and event B which specifies whether there is some $h \in \mathcal{C}$ such that h is consistent with S_1, but h makes $\geqslant \varepsilon/2$ mistakes on S_2. The reason we have introduced this event B is because we can bound $\Pr[A]$ by twice $\Pr[B]$ for large enough m; in particular, for $m \geqslant 8/\varepsilon$, $\Pr[A] \leqslant 2\cdot\Pr[B]$. Given this claim, the remainder of our effort will be focused on showing that $\Pr[B] \leqslant \delta/2$. Let's now prove the claim.

Note that

$$\Pr[B] \geqslant \Pr[A \text{ and } B] = \Pr[A]\cdot\Pr[B \mid A],$$

so that it is enough to show that $\Pr[B \mid A] \geqslant 1/2$—this is true because since A happens, there is some bad $h^* \in \mathcal{C}$ consistent with S_1, so that since

$$\mathbb{E}\left[\text{number of mistakes of the bad } h^* \text{ on } S_2\right] \geqslant \varepsilon m,$$

we have that for $\gamma = 1 - \gamma = \frac{1}{2}$,

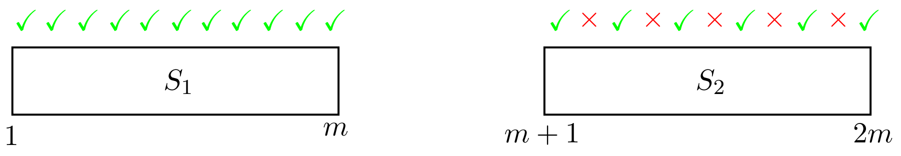
$$\Pr\left[\begin{smallmatrix} h^* \text{ makes fewer than half its expected} \\ \text{number of mistakes on } m \text{ examples}\end{smallmatrix}\right] \leqslant e^{-\varepsilon m\gamma^2/2},$$

which is $\leqslant 1/2$ as long as $m \geqslant 8/\varepsilon$. This then means that

$$\Pr\left[\begin{smallmatrix} \text{bad hypothesis } h^* \text{ makes at least} \\ m \text{ mistakes on a fresh batch of draws } S_2\end{smallmatrix}\right] \geqslant 1 - e^{-\varepsilon m\gamma^2/2} \geqslant \frac{1}{2},$$

which is sufficiently high, so $\Pr[B \mid A] \geqslant 1/2$.

The rest of the argument is just to show that $\Pr[B] \leqslant \delta/2$. Recall that event B is



where the hypothesis has made $\geqslant \varepsilon m/2$ mistakes. We can equivalently view B as follows:

- draw a set S of $2m$ examples,
- randomly separate the examplesinto two equally-sized halves S_1 and S_2.

Then B is the event that there is some $h \in \mathcal{C}$ that is consistent with S_1 and wrong on at least $\frac{\varepsilon}{2} \cdot m$ points in S_2.

Now fix a *particular* labelling $\mathcal{L}$ of the $2m$ points in S which labels at least $\frac{\varepsilon}{2} \cdot m$ of them wrongly, i.e. differently from the concept c. Now we argue that this probability is very small:

$$\Pr_{\substack{\text{split of } S \\ \text{into } S_1 \cup S_2}} [\text{the labelling } \mathcal{L} \text{ puts all the wrong points in } S_2] \leqslant \frac{\delta}{2}.$$

Note that this entire situation is strikingly equivalent to that in which we have $2m$ balls in a barrel with at least $k = \frac{\varepsilon}{2} \cdot m$ black and the rest white. Split the balls in the barrel randomly into bags of size m and m respectively. Then writing $\mathcal{B}_{S_2}$ for the event that all the black balls are in the second bag, we have that

$$\begin{aligned}\Pr[\mathcal{B}_{S_2}] &= \binom{m}{k} \Big/ \binom{2m}{k} = \frac{m!}{(m-k)!k!} \cdot \frac{(2m-k)!k!}{(2m)!} \\ &= \frac{m}{2m} \cdot \frac{m-1}{2m-1} \cdot \ldots \cdot \frac{m-k+1}{2m-k+1} \\ &\leqslant \frac{1}{2^k}.\end{aligned}$$

Carrying our analogy over, we observe that each fixed labelling of $2m$ examples has $\leqslant 1/2^{\varepsilon m/2}$ chance of giving us event B. Applying a union bound over all the $\leqslant \Pi_{\mathcal{C}}(2m)$ labellings of S then gives

$$\Pr[B] \leqslant \Pi_{\mathcal{C}}(2m) \cdot \frac{1}{2^{\varepsilon m/2}} \leqslant \frac{\delta}{2},$$

where the final inequality holds because by taking logarithms base 2,

$$\log\left(\Pi_{\mathcal{C}}(2m)\right) + \log\left(\frac{2}{\delta}\right) \leqslant \frac{\varepsilon m}{2} \iff m \geqslant \frac{2}{\varepsilon}\left(\log\left(\Pi_{\mathcal{C}}(2m)\right) + \log\left(\frac{2}{\delta}\right)\right).$$

This completes the proof of the theorem (note that since $2 < e$, $\log\left(\Pi_{\mathcal{C}}(2m)\right) < \ln\left(\Pi_{\mathcal{C}}(2m)\right)$). $\square$

Now we know that, to construct an (ε, δ)-PAC learner for a concept class $\mathcal{C}$, it suffices to find a CHF for $\mathcal{C}$ on m examples, where m satisfies

$$m \geqslant \frac{2}{\varepsilon}\left(\ln\left(\Pi_{\mathcal{C}}(2m)\right) + \ln\frac{2}{\delta}\right), \quad m \geqslant \frac{8}{\varepsilon}.$$

Recall that from the Amazing Theorem,

$$\Pi_{\mathcal{C}}(2m) \leqslant \left(\frac{2em}{d}\right)^d \iff \log\left(\Pi_{\mathcal{C}}(2m)\right) \leqslant d \cdot \log\left(\frac{2em}{d}\right).$$

So it indeed sufficies to have m examples with m satisfying

$$m \geqslant \frac{2}{\varepsilon}\left(d \cdot \log\left(\frac{2em}{d}\right) + \ln\frac{2}{\delta}\right), \quad m \geqslant \frac{8}{\varepsilon},$$

which always holds provided that

$$m \geqslant k\left(\frac{d}{\varepsilon}\log\frac{1}{\varepsilon} + \frac{1}{\varepsilon}\log\frac{1}{\delta}\right),$$

for some constant k with $|k-8| \approx 0$.

Furthermore, suppose we ignored the δ in the sample complexity obtained from Theorem 16.2, as well as extraneous constants. Then

$$m \geqslant \frac{2d}{\varepsilon}\log\frac{1}{\varepsilon} \implies \frac{m}{d} \geqslant \frac{2}{\varepsilon}\log\frac{m}{d},$$

so that

$$\frac{m/d}{\log(m/d)} \geqslant \frac{2}{\varepsilon} \implies \frac{m}{d} = \frac{2}{\varepsilon}\log\frac{2}{\varepsilon}$$

is enough, via our analysis in Lecture 11.1.1. Thus we have derived a concrete upper bound on the sample complexity of PAC learning.

§16.2 Application: learning linear threshold functions over $\mathbb{R}^n$

Now let's see why this upper bound is helpful. Consider the concept class $\mathcal{C} = \{\text{all LTFs over } \mathbb{R}^n\}$. Note that this concept class is infinite ($X = \mathbb{R}$ itself is indeed infinite).

Fact 16.2. *If $X = \mathbb{R}^n$, then* $\mathrm{VCDIM}(\mathcal{C}) = n + 1$.

Proof. See midterm. □

Why is this important? Thanks to this result we now know that to reason properly about the sample complexity of learning LTFs, we can use the upper bound on the sample complexity of PAC learning via the VC dimension, which helps us develop statistically and computationally efficient algorithms for learning this concept class.

Fact 16.3. *There is a computationally efficient CHF for the concept class $\mathcal{C} = \{$all LTFs over $\mathbb{R}^n\}$ using $\mathcal{C}$ via polynomial-time linear programming.*

Proof. Here's an example for how this works. Say $w_1x_1 + \ldots + w_nx_n \geqslant \theta$ is an unknown LTF. A positive example $[(3, 1, -5, \ldots, 6), +]$ in $\mathbb{R}^n$ gives the constraint

$$3w_1 + w_2 - 5w_3 + \ldots + 6w_n \geqslant \theta.$$

On the other hand, a negative example $[(4, -2, 7, \ldots, 3.3), -]$ gives the constraint

$$4w_1 - 2w_2 + 7w_3 + \ldots + 3.3w_n < \theta.$$

Problem Linear Programming (LP) feasibility:

Given a collection of linear inequalities, is there a solution that satisfies all of them? If so, find such a solution.

There are polynomial-time algorithms for this! Via these algorithms, we can construct computationally efficient PAC-learning algorithms for $\mathcal{C} = \{\text{all LTFs over } \mathbb{R}^n\}$ using $\mathcal{C}$:

- draw m examples where $m \approx \frac{1}{\varepsilon}\left(n \log \frac{1}{\varepsilon} + \ln \frac{1}{\delta}\right)$,
- run a polynomial-time algorithm for LP to get a consistent $w_1, \ldots, w_n, \theta$,
- output hypothesis $\text{sign}(w \cdot x - \theta)$. $\square$

Next time: Boosting.

§17 Lecture 17—1st November, 2023

Last time.

- Finished proof for upper bound on the sample complexity of PAC learning.
- Application: $\mathcal{C} = \{\text{all LTFs over } \mathbb{R}^n\}$ is efficiently PAC-learnable.

Today. Start unit on boosting confidence and accuracy:

- Setup, framework of boosting.
- Proof of concept: 3-stage boosting.

§17.1 Boosting

Motivation. What does it mean to (strongly) PAC learn a concept class $\mathcal{C}$? We have always said that an algorithm A PAC learns a concept class $\mathcal{C}$ if for every distribution $\mathcal{D}$, any $c \in \mathcal{C}$, and *any* $\varepsilon, \delta > 0$, given access to ε, δ and an oracle $\text{EX}(c, \mathcal{D})$, with probability $\geqslant 1 - \delta$, A outputs h such that $\Pr_{x \sim \mathcal{D}}[h(x) \neq c(x)] \leqslant \varepsilon$. Note that the requirements on ε and δ are quite strong here—in fact, we will come to know this as *strong PAC learning*—we will not be able to learn this concept class at all if they are not entirely met.
What happens if we weaken the definition? As it turns out, there are no concept classes that are weakly learnable but not strongly learnable. We will explore all of this in this unit.

§17.1.1 Boosting the confidence

Let's start by attempting to weaken the requirement on δ.

Definition 17.1 (Low-confidence PAC learning)**.** *An algorithm A'* low-confidence PAC-learns *a concept class $\mathcal{C}$ if, for all distributions $\mathcal{D}$ and $c \in \mathcal{C}$, for all $\varepsilon > 0$ with fixed $\delta = 0.1$, given ε and access to $\mathrm{EX}(c, \mathcal{D})$, with probability $\geqslant 0.9$, A outputs a hypothesis h such that $\Pr_{x\sim\mathcal{D}}[h(x) \neq c(x)] \leqslant \varepsilon$.*

Obviously, if you have a regular PAC learner, then you get a low-confidence PAC learner for free(pick our favourite δ). The interesting fact here is that the opposite also holds; given a low-confidence PAC learner A, we can also get a regular PAC learner that works like this, broadly speaking:

- run A' repeatedly k times (so that with probability $1 - \frac{1}{10^k}$, some run gives good hypotheses); we can ensure that we have a $1 - \delta/2$ failure chance here, and
- test the k hypotheses on fresh examples, and output the one that performs the best, where just like above, we can ensure that we have a $1 - \delta/2$ failure chance here.

Because of this, δ is a boring parameter for us. We'll sometimes ignore δ in our analyses—since we now know that PAC learning is robust to our choice of δ—particularly when we have bigger fish to fry in the form of weakening ε.

§17.1.2 Boosting the accuracy via Schapire's boosting algorithm

Indeed, the bigger question for us which many once believed to be impossible is that of *boosting the accuracy ε.*

Definition 17.2 (Weak PAC learning)**.** *An algorithm A is a* weak PAC learner *for $\mathcal{C}$ with advantage γ if, for all distributions $\mathcal{D}$ and $c \in \mathcal{C}$, and for all $\delta > 0$, with probability $\geqslant 1 - \delta$, the algorithm A granted access to $\mathrm{EX}(c, \mathcal{D})$ outputs a hypothesis h such that*

$$\Pr_{x\sim\mathcal{D}}[h(x) \neq c(x)] \leqslant \frac{1}{2} - \gamma.$$

When $\gamma = 0$, then our classification is trivial and we can simply toss coins, so we must do better than that. This notion of weak PAC learning initially seems weaker than that of strong PAC learning, but as we will see, it surprisingly isn't.

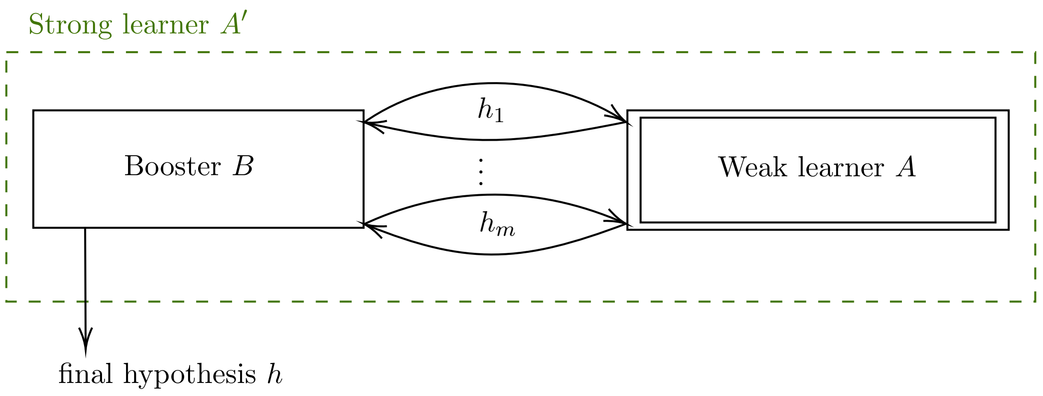
Theorem 17.1. *Let $\mathcal{C}$ be any concept class. If there is an efficient weak PAC learning algorithm A for $\mathcal{C}$, then there is an efficient strong PAC learning algorithm A' for $\mathcal{C}$, where A' runs in time* $\mathrm{poly}\left(\frac{1}{\gamma}, \frac{1}{\varepsilon}, \log(\frac{1}{\delta}), T\right)$*, where T is the running time of A required to achieve confidence* 0.9.

Our proof of this theorem will be by giving efficient *boosting algorithms* B that take a weak PAC learner A and run it $\mathrm{poly}\left(\log(\frac{1}{\varepsilon}), \frac{1}{\gamma}\right)$ times—with clever changes along the way—to get a strong PAC learner.

The high-level idea of boosting is as follows. To learn a concept $c \in \mathcal{C}$ to accuracy ε given access to $\mathrm{EX}(c, \mathcal{D})$, as well as a weak learner A with advantage γ, our boosting algorithm (which we will also simply call a *booster*):

1. Runs the weak learner A repeatedly using *different* distributions on the examples, that is, using $\mathrm{EX}(c, \mathcal{D}_1), \mathrm{EX}(c, \mathcal{D}_2), \ldots, \mathrm{EX}(c, \mathcal{D}_m)$, and get hypotheses $h_1, h_2, \ldots, h_m$.

2. Combines the hypotheses $h_1, h_2, \ldots, h_m$ to get a final hypothesis h.



What does this really mean? Let's flesh out some of the specifics of this algorithm.

- *What is the idea behind the distributions $\mathcal{D}_1, \mathcal{D}_2, \ldots, \mathcal{D}_m$?* The booster will cook these distributions up so as to force the weak learner A to serve as a reluctant ally, to give some new information each time it is run. This is a bit of a black art, and we will see how to do this in time.
- *How can we run A on the* $\mathrm{EX}(c, \mathcal{D}_1), \mathrm{EX}(c, \mathcal{D}_2), \ldots, \mathrm{EX}(c, \mathcal{D}_m)$, *when we only have access to* $\mathrm{EX}(c, \mathcal{D})$*?* This is a bit of a trick; we will see two possible approaches:
  - Filter/adapt examples to change $\mathcal{D}$ into $\mathcal{D}_2, \mathcal{D}_3, \ldots, \mathcal{D}_m$, or
  - Draw a fixed sample F from $\mathcal{D}$, and then focus all our learning on that finite set of examples, reweighting them as we go along while explicitly maintaining the same distribution over F.
- *How do we combine the hypotheses $h_1, h_2, \ldots, h_m$?* We will typically use a majority vote or a weighted majority vote, but there are other approaches as well (e.g. branching programs).
- *Why does it all work? Why is* $\Pr_{x\sim\mathcal{D}}[h(x) \neq c(x)] \leqslant \varepsilon$*?* This is the most important question, and we will see a proof that a booster can indeed achieve this.

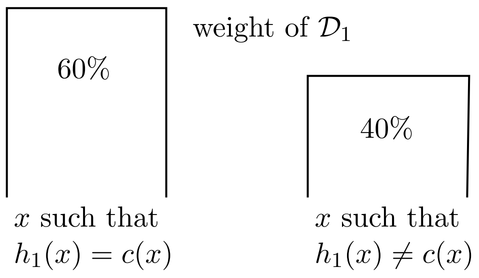
We will see a concrete and simple example of boosting in action, which is the 3-stage boosting algorithm of Schapire. This algorithm is recursive, but we will only do one step for simplicity.

First some setup. Let A be a weak PAC learning algorithm with advantage $\gamma = 0.1$; in other words, for any distribution $\mathcal{D}$ and any $c \in C$, A outputs a hypothesis h such that $\Pr_{x\sim\mathcal{D}}[h(x) \neq c(x)] \leqslant 0.4$. We will see how to efficiently boost the accuracy to achieve error $\leqslant 0.352$ by doing three runs of A with different distributions $\mathcal{D}_1, \mathcal{D}_2, \mathcal{D}_3$ to get hypotheses h_1, h_2, h_3, and then combining them to get a final hypothesis $h = \mathsf{MAJ}(h_1, h_2, h_3)$. We will make two simplifying assumptions, namely that at every run, A gives a hypothesis with error exactly equal to 0.4, and that this happens with probability exactly equal to 1 (ignoring the $\delta = 0$).

Notation 17.3. For a distribution $\mathcal{D}$ and an event E, we write $\mathcal{D}(E)$ for $\Pr_{\mathcal{D}}[\mathbb{E}\text{ holds}]$. For instance we might write $\mathcal{D}(h(x) \neq c(x))$ for $\Pr_{x\sim\mathcal{D}}[h(x) \neq c(x)]$.

Then the booster does this:

1. Run the weak learner A on $\mathrm{EX}(c, \mathcal{D} = \mathcal{D}_1)$ to get a hypothesis h_1, which we know to have, by our assumptions, $\mathcal{D}_1(h_1(x) \neq c(x)) = 0.4$.

   
   weight of $\mathcal{D}_1$
   60%
   40%
   x such that $h_1(x) = c(x)$
   x such that $h_1(x) \neq c(x)$

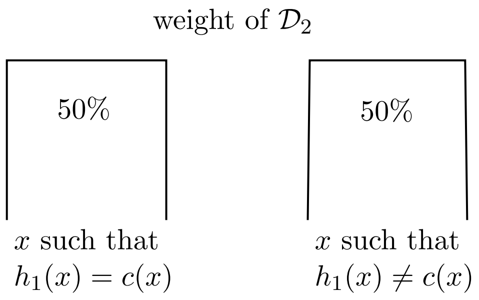
   (If we get an example from the oracle and we run the booster to get h_1, we can run h_1 on the example and see if it agrees with the oracle.)

2. What is a new distribution $\mathcal{D}_2$ such that running A on examples from $\mathrm{EX}(c, \mathcal{D}_2)$ would tell me something new that h_1 didn't already tell me? (One suboptimal approach towards this distribution is to filter by discarding all the x such that $h_1(x) = c(x)$, so that $\overline{h_1}$ has perfect accuracy—h_1 is wrong on every example from the distribution—and the weak learner can give this filtered distribution to us as $\mathcal{D}_2$. But this doesn't help us, because we already know that h_1 is wrong on every example from $\mathcal{D}_2$! We need to be more clever.)

   So here's what the booster does. Let $\mathcal{D}_2$ be the distribution which:

   - scales up the weight of each $x \in X$ such that $h_1(x) \neq c(x)$ by a factor of $\frac{5}{4}$, and
   - scales down the weight of each $x \in X$ such that $h_1(x) = c(x)$ by a factor of $\frac{5}{6}$.

   Now the two events have probabilities $\mathcal{D}_2(h_1(x) = c(x)) = \frac{5}{6} \cdot \mathcal{D}_1(h_1(x) = c(x)) = \frac{5}{6} \cdot 0.6 = 0.5$ and $\mathcal{D}_2(h_1(x) \neq c(x)) = \frac{5}{4} \cdot \mathcal{D}_1(h_1(x) \neq c(x)) = \frac{5}{4} \cdot 0.4 = 0.5$. So with this $\mathcal{D}_2$, the probability $\Pr_{x \sim \mathcal{D}_2}[h_1(x) \neq c(x)] = \frac{1}{2}$, and so we must get a different hypothesis h_2 from A.

   
   weight of $\mathcal{D}_2$
   50%
   50%
   x such that $h_1(x) = c(x)$
   x such that $h_1(x) \neq c(x)$

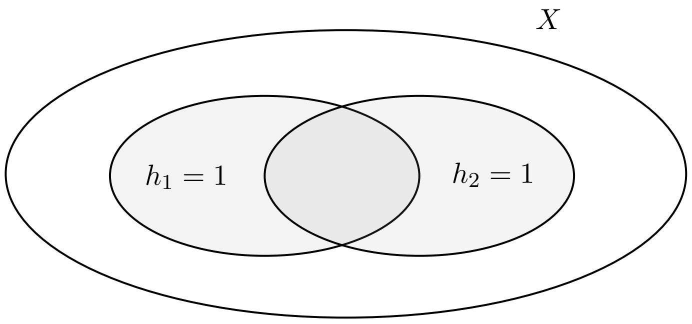
   One question we may have is how exactly we do the rescaling. One way to do this is to draw an example from the original distribution $\mathcal{D}_1$, and if it lives on the right block, keep it for sure, and if it lives on the left block, discard it with probability $\frac{1}{3}$. A different way is to toss a fair coin; if it comes up heads, draw from $\mathrm{EX}(c, \mathcal{D})$ until we get an x such that $h_1(x) = c(x)$, and if it comes up tails, draw from $\mathrm{EX}(c, \mathcal{D})$ until we get an x such that $h_1(x) \neq c(x)$.[1] For the second approach, the expected number of repetitions even in the worst case is

   $$\mathbb{E}[\text{number of repetitions}] = \frac{1}{0.4} = \frac{5}{2} = 2.5,$$

[1]Both of these methods are efficient because each block has quite a bit of weight in it, and so we don't have to wait too long to get an example from the right block.

and so we can expect to get an example from the right block in 3 repetitions with probability $\geqslant 0.9$.

3. We will only motivate the third distribution $\mathcal{D}_3$ and present its analysis next time. Consider the following picture, with the perspective that our final hypotheses $h = \mathsf{MAJ}(h_1, h_2, h_3)$.



Since h_3 only matters on x if $h_1(x) \neq h_2(x)$, we can take $\mathcal{D}_3$ to be the distribution $\mathcal{D}_1$ conditioned on the event $\{x \in X : h_1(x) \neq h_2(x)\}$. To do this (i.e. simulate $\mathrm{EX}(c, \mathcal{D}_3)$), we can draw examples from $\mathrm{EX}(c, \mathcal{D}_1)$ until we get an x such that $h_1(x) \neq h_2(x)$, and then use that $(x, c(x))$ as an example from $\mathrm{EX}(c, \mathcal{D}_3)$.

We then run A on $\mathrm{EX}(c, \mathcal{D}_3)$ to get a hypothesis h_3, and then output $h = \mathsf{MAJ}(h_1, h_2, h_3)$ as our final hypothesis.

Question: What if $\Pr_{x \sim \mathcal{D}_1}[h_1(x) \neq h_2(x)] \coloneqq \tau$ is tiny?
$\triangleright$ Then we can't expect to get a different hypothesis h_3 from A efficiently, as the runtime overhead to simulate 1 draw from $\mathrm{EX}(c, \mathcal{D}_3)$ is $1/\tau$ in expectation. However, if τ is tiny, then with probability $\geqslant 1 - \tau$, $h_1 = h_2 = h$, and so we can output h as our final hypothesis regardless of what h_3 is. Therefore in this situation with tiny τ, we can output $h = h_1 = h_2$ as our final hypothesis (implicitly using anything for h_3), and then the accuracy $\Pr_{x \sim \mathcal{D}}[h(x) \neq c(x)]$ will only change by something tiny (at most τ and at least $-\tau$).

Next time, we will analyze the accuracy of the final hypothesis $h = \mathsf{MAJ}(h_1, h_2, h_3)$, and see that it is at least $(1 - 0.352)$-accurate.

§18 No Lecture—Election Day, 7th November, 2023

Today was a holiday—no class. Instead, we have a bad joke about learning theory.

Why did the PAC learning algorithm get invited to all the parties? Because it was great at making predictions in uncertain environments—just don't ask it to dance without a confidence interval.

§19 Lecture 19—8th November, 2023

Last time. Started the unit on boosting confidence and accuracy:

- Gave setup and framework of boosting.
- Gave a proof of concept:
  - 3-stage boosting to go from 40% error to 35.2% error.
  - described the distributions $\mathcal{D}_1, \mathcal{D}_2, \mathcal{D}_3$ to simulate the 3-stage boosting.

Today. Start unit on boosting confidence and accuracy:

- Analysis of 3-stage boosting and a justification for why it gives 35.2% error h with respect to the distribution $\mathcal{D}$.
- Discuss the extension to the "full" booster (with $1 - \varepsilon$ accuracy).
- Discuss boosting over a fixed sample.
- Introduce the AdaBoost algorithm, a simple, practical booster over a fixed sample.

§19.1 Boosting by filtering: an analysis of the 3-stage booster

Remember from last time that we obtained a combined final hypothesis $h = \mathsf{MAJ}(h_1, h_2, h_3)$ with $\mathcal{D} = \mathcal{D}_1$. We want to show that

$$\Pr_{x \sim \mathcal{D}}[h(x) \neq c(x)] \leqslant 0.352.$$

Indeed we will show the strict equality (due to the simplifying assumptions we made last time).

Divide the domain X into four regions according to h_1, h_2, and c, and define $\rho = \mathcal{D}_2(R_2)$, carrying over the notation from last time:

R_1	R_2	R_3	R_4
all x such that $h_1(x) = c(x)$, $h_2(x) = c(x)$. Here, $\mathcal{D}_2(R_1) = 0.5 - \rho$ $\mathcal{D}_1(R_1) = \frac{6}{5}(0.5 - \rho)$	all x such that $h_1(x) = c(x)$, $h_2(x) \neq c(x)$. Here, $\mathcal{D}_2(R_2) = \rho$ $\mathcal{D}_1(R_1) = \frac{6}{5}\rho$	all x such that $h_1(x) \neq c(x)$, $h_2(x) \neq c(x)$. Here, $\mathcal{D}_2(R_3) = 0.4 - \rho$ $\mathcal{D}_1(R_1) = \frac{4}{5}(0.4 - \rho)$	all x such that $h_1(x) \neq c(x)$, $h_2(x) = c(x)$. Here, $\mathcal{D}_2(R_1) = 0.1 + \rho$ $\mathcal{D}_1(R_1) = \frac{4}{5}(0.1 + \rho)$

Recall that $\rho := \mathcal{D}_2(R_2)$, and we have the following. Since $\mathcal{D}_2(\{x : h_2(x) \neq c(x)\}) = 0.4$, we can guarantee that $\mathcal{D}_2(R_1) = 0.5 - \rho$ and $\mathcal{D}_2(R_3) = 0.4 - \rho$. We also have that $\mathcal{D}_2(\{x : h_1(x) \neq c(x)\}) = 0.5$ by our choice of $\mathcal{D}_2$, so $\mathcal{D}_2(R_4) = 0.1 + \rho$ and $\mathcal{D}_2(R_1) = 0.5 - \rho$. We also have that

$\mathcal{D}_1(R_1) = \frac{6}{5}(0.5 - \rho)$, $\mathcal{D}_1(R_2) = \frac{6}{5}\rho$, $\mathcal{D}_1(R_3) = \frac{4}{5}(0.4 - \rho)$, and $\mathcal{D}_1(R_4) = \frac{4}{5}(0.1 + \rho)$, by the way we engineered $\mathcal{D}_2$ from last class—just reverse-engineer the distributions to get these values.

Now $\mathcal{D}_3$ has nonzero weight only on R_2 and R_4 by our analysis from last time, and on these, $\mathcal{D}_3$ is merely a rescaling of $\mathcal{D}_1$. Now the final hypothesis h is correct on all of R_1 and incorrect on all of R_3, so on R_2 and R_4, h is correct exactly when h_3 is correct, i.e. $h = c$ if and only if $h_3 = c$. Now $\mathcal{D}_3$ is such that h is correct 60% of the time, and incorrect 40% of the time, so

$$\Pr_{x\sim\mathcal{D}_3}[h(x) \neq c(x)] = 0.4$$
$$\Pr_{x\sim\mathcal{D}_3}[h(x) = c(x)] = 0.6.$$

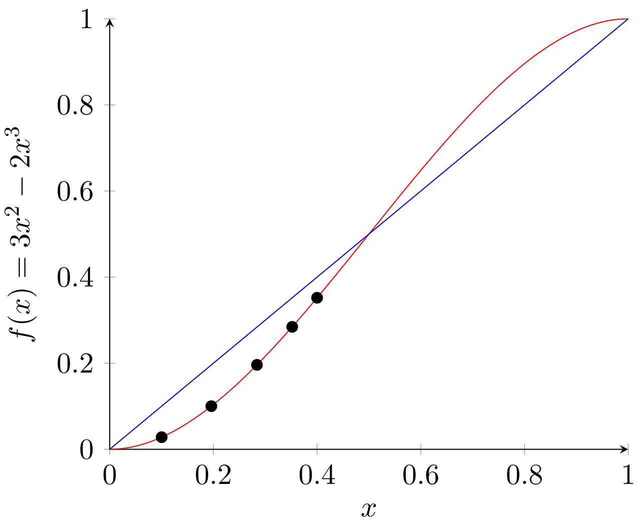
So under $\mathcal{D} = \mathcal{D}_1$, h is wrong 40% of the time, and correct 60% of the time on $R_2 \cup R_4$. Therefore, all in all, we make no errors on R_1, we always make errors on R_3, and we make errors 40% of the time on $R_2 \cup R_4$, so

$$\begin{aligned}\Pr_{x\sim\mathcal{D}=\mathcal{D}_1}[h(x) \neq c(x)] &= 1 \cdot \mathcal{D}_1(R_3) + 0.4 \cdot \mathcal{D}_1(R_2) + 0.4 \cdot \mathcal{D}_1(R_4)\\ &= \frac{4}{5}(0.4 - \rho) + 0.4 \cdot \left(\frac{6}{5}\rho + \frac{4}{5}(0.1 + \rho)\right)\\ &= 0.352,\end{aligned}$$

since $\frac{4}{5}(-\rho) + 0.4 \cdot \frac{6}{5} \cdot \rho + 0.4 \cdot \frac{4}{5} \cdot \rho = 0$.

We have done something truly remarkable here: we have gone down from 40% error to 35.2% error, so we have improved the accuracy by 4.8% by using a 3-stage boosting algorithm. So we can view this 3-stage booster as some other weak learner which achieves 35.2% error, and we can use this weak learner to boost it further to achieve error even < 35.2%.

In general, as we will see on a problem set, if the weak learner has error $\beta < \frac{1}{2}$, then the hypothesis h coming from this three stage procedure will have error $\leqslant 3\beta^2 - 2\beta^3$, and indeed we can recurse on this.



In fact we get a doubly-exponential rate of convergence towards a very small error—in particular, if we apply $f(f(\cdots f(0.4)\cdots))$, we require $O(\log\log\frac{1}{\varepsilon})$ iterations to ger an error $\leqslant \varepsilon$. So our recursive tree of calls has height $O(\log\log\frac{1}{\varepsilon})$, and a branching factor 3, so that if our target error is $\leqslant \varepsilon$, we require

$$3^{O(\log\log\frac{1}{\varepsilon})} = \text{poly}\left(\log\frac{1}{\varepsilon}\right)$$

calls to the weak learner overall.

Unfortunately this 3-stage boosting algorithm is not practical for any number of reasons, including but not limited to the facts that it doesn't take advantage of the fact we might have fluctuations in the error rate, it doesn't take advantage of the quality of the weak learner, it is recursive, it requires discarding a lot of data, etc. It's purpose in the literature is to show that boosting is possible, and to give a proof of concept that we have hope of boosting a weak learner. That hope is realized in the AdaBoost algorithm, which we will discuss next.

§19.2 Boosting over a fixed sample: the AdaBoost algorithm

Boosting over a fixed sample is much more practical, and saves data. The key themes are as follows:

1. Draw a fixed sample S of m data points $(x_1, y_1 = c(x_1)), \ldots, (x_m, y_m = c(x_m))$. This is the only data ever used by the boosting algorithm we will present.

2. We can now think of a distribution as a weighting scheme applied to each point. Initially, we have $\mathcal{D}(x_i) = \frac{1}{m}$ for all $i \in [m]$. We will update these weights as we go along to get a new distribution $\mathcal{D}'$ such that $\mathcal{D}'(x_i) \geqslant 0$ for all $i \in [m]$ and $\sum_{i=1}^{m} \mathcal{D}'(x_i) = 1$.

3. In this setting, the weak learning guarantee is as follows: for any weighting $w_1, \ldots, w_m$ such that $w_i \geqslant 0$ and $\sum_{i=1}^{m} w_i = 1$, if we run the weak learning algorithm using this distribution, we get a hypothesis h such that
$$\sum_{\substack{i\in[m]\\ h(x_i)\neq y_i}} w_i \leqslant \frac{1}{2} - \gamma,$$
where γ is the advantage of the weak learner. This is the same as the guarantee we had before, but now we have a weighted sum.

4. The desideratum of the booster is then to find a final hypothesis h with small error, that is, we want to find a hypothesis h such that
$$\sum_{\substack{i\in[m]\\ h(x_i)\neq y_i}} \frac{1}{m} \leqslant \varepsilon,$$
where ε is the target error rate (sometimes we can even shoot for $\varepsilon = 0$).

To accomplish this, we present the AdaBoost algorithm, which is a simple, powerful, and practical boosting algorithm with nice theoretical guarantees that works for boosting over a fixed sample. The algorithm takes in as input the m labelled examples $S = \{(x_1, y_1), \ldots, (x_m, y_m)\}$, where each label y_i is either $+1$ or -1, and the number of rounds T of boosting to perform. In the algorithm which we will soon present, we denote by $\mathcal{D}_t$ the t-th distribution AdaBoost constructs over the m examples, by $\mathcal{D}_t(i)$ the probability $\Pr_{x\sim\mathcal{D}_t}[x = x_i]$, and by ε_t the error rate $\Pr_{x\sim\mathcal{D}_t}[h_t(x) \neq y_i]$, where h_t is the hypothesis produced by the weak learner at round t under the distribution $\mathcal{D}_t$.

The algorithm proceeds as follows:

Algorithm: AdaBoost [FS+96]

1. Initialise the distribution $\mathcal{D}_1(i) = \frac{1}{m}$ for all $i \in [m]$.
2. For $t = 1, 2, \ldots, T$ do
   - Run the weak learner L on $\mathcal{D}_t$ to get a hypothesis $h_t : X \to \{-1, 1\}$ which has error rate $\varepsilon_t = \Pr_{x \sim \mathcal{D}_t}[h_t(x) \neq y_i]$ with respect to the distribution $\mathcal{D}_t$.
   - Set $\alpha_t = \frac{1}{2} \ln\left(\frac{1-\varepsilon_t}{\varepsilon_t}\right)$.
   - Update
   $$\mathcal{D}_{t+1}(i) = \frac{\mathcal{D}_t(i) \cdot \exp\left(-\alpha_t y_i h_t(x_i)\right)}{Z_t},$$
   where Z_t is a normalising constant chosen so that $\mathcal{D}_{t+1}$ is a distribution, i.e. the sum of all the probability weights $\sum_{i=1}^m \mathcal{D}_{t+1}(i) = 1$.
3. The final hypothesis is the weighted majority vote of weak hypotheses, i.e. the signed weighted sum
$$h(x) = \mathsf{sign}\left(\sum_{t=1}^T \alpha_t h_t(x)\right),$$
where the weight α_t is the weight of the t-th weak hypothesis.

Here are some key ideas we must keep in mind when thinking about the AdaBoost algorithm:

1. Given h_t and $\mathcal{D}_t$, we construct $\mathcal{D}_{t+1}$ by increasing the weight of the examples on which h_t made mistakes, and decreasing the weight of the examples on which h_t made no mistakes—just like we did when we constructed $\mathcal{D}_2$ from $\mathcal{D}_1$ in the 3-stage boosting algorithm. The point is to make sure that h_t is not incorrect on more than half the examples under $\mathcal{D}_{t+1}$.
2. At the end, we combine the weak hypotheses $h_1, \ldots, h_T$ into a single hypothesis h by taking a weighted majority vote, where the weight of h_t is α_t. Here, the weight of h_t is proportional to its accuracy under the distribution $\mathcal{D}_t$, so our weighting scheme is at least somewhat clever; it does better than the original booster that didn't even take into account the quality of the weak learners. This is why it's called AdaBoost: it adapts automatically to the different accuracies we might obtain from each of the weak learners.

   Example 19.1. Recall that
   $$\alpha_t = \frac{1}{2} \ln\left(\frac{1-\varepsilon_t}{\varepsilon_t}\right).$$
   If $\varepsilon_t = 0.4$, then $\alpha_t = 0.202$, and if $\varepsilon_t = 0.01$, then $\alpha_t = 2.29$. So in this sense we are giving more weight to the weak learners that are more accurate. In fact, if we have one weak hypothesis which is 99% accurate, it has ten times as much voting power as a weak learner which is only 60% accurate.

3. The AdaBoost update rule is quite analogous to the weighted majority algorithm:

Weighted majority	AdaBoost
Expert i	i-th example for AdaBoost
Prediction of expert i at trial t	Hypothesis $h_t(i)$
Trial t	t-th run of the weak learner
Weight of expert i at trial t	$\mathcal{D}_t(i)$

One interesting distinction though is that with weighted majority, we punish the experts who make mistakes (decrease their weights), but with AdaBoost, that's a sign we should increase its weight, essentially deploying more of the weak learner's power to the examples on which it made mistakes.

Next time, we will justify many of the claims we have made about AdaBoost, and prove some nice theory about its performance.

§20 Lecture 20—13th November, 2023

Last time. Started the unit on boosting confidence and accuracy:

- Analysed of 3-stage boosting and a justification for why it gives a 35.2% error h with respect to the distribution $\mathcal{D}$.
- Discussed the extension to the "full" booster (with $1-\varepsilon$ accuracy).
- Discussed boosting over a fixed sample.
- Introduced the AdaBoost algorithm, a simple, practical booster over a fixed sample, as well as its main ideas:
  - reweighting examples based on the current hypothesis's errors to ensure that h_t is 50% accurate under $\mathcal{D}_{t+1}$.
  - the clever weighted majority vote to combine the hypotheses.

Today.

- Analysis of AdaBoost: state and prove a theorem about its performance.
- Start a new unit on PAC learning in the presence of noise: general framework for noise as well as some particular noise models.

§20.1 Analysis of AdaBoost

Let us start by considering once again the reweighting rule for examples in AdaBoost. For the t-th iteration, we have a distribution $\mathcal{D}_t$ over the examples, and we want to define a new distribution $\mathcal{D}_{t+1}$ over the examples like:

$$\mathcal{D}_{t+1}(i) = \frac{\mathcal{D}_t(i)}{Z_t} \cdot \exp\left(-\alpha_t y_i h_t(x_i)\right),$$

for h_t and y_i taking values in $\{\pm 1\}$ where $\alpha_t = \frac{1}{2}\ln\left(\frac{1-\varepsilon_t}{\varepsilon_t}\right)$. Let's decode this a bit:

- If $h_t(x_i) = y_i$—that is, if h_t gets the example right—then

  $$\exp\left(-\alpha_t y_i h_t(x_i)\right) = \exp(-\alpha_t) = \sqrt{\frac{\varepsilon_t}{1-\varepsilon_t}}.$$

  In this case, ignoring the renormalisation, if the total weight on i such that $h_t(x_i) = y_i$ is $1 - \varepsilon_t$ (which is the accuracy of h_t for error rate ε_t), then this is getting multiplied by $\exp\left(-\alpha_t y_i h_t(x_i)\right)$, and the total weight goes from $1 - \varepsilon_t$ to

  $$\sqrt{\frac{\varepsilon_t}{1-\varepsilon_t}} \cdot (1-\varepsilon_t) = \sqrt{\varepsilon_t(1-\varepsilon_t)}.$$

- On the other hand, if $h_t(x_i) \neq y_i$, then

  $$\exp\left(-\alpha_t y_i h_t(x_i)\right) = \exp(\alpha_t) = \sqrt{\frac{1-\varepsilon_t}{\varepsilon_t}}.$$

  Again ignoring the renormalisation, if the total weight on i such that $h_t(x_i) \neq y_i$ is ε_t, then this is getting multiplied by $\exp\left(-\alpha_t y_i h_t(x_i)\right)$, and the total weight goes from ε_t to

  $$\sqrt{\frac{1-\varepsilon_t}{\varepsilon_t}} \cdot \varepsilon_t = \sqrt{\varepsilon_t(1-\varepsilon_t)}.$$

So $\mathcal{D}_{t+1}$ is a renormalisation of $\mathcal{D}_t$ that keeps the total weight on the examples that h_t gets right and the examples that h_t gets wrong the same, and it increases the weight on the examples that h_t gets wrong and decreases the weight on the examples that h_t gets right. We see this calculation again in more generality when we prove the following theorem on the performance of AdaBoost:

Theorem 20.1 (AdaBoost performance)**.** *Suppose AdaBoost is run for T stages. Then its final hypothesis $H\colon \{x_1, \ldots, x_m\} \to \{\pm 1\}$ makes errors on at most a*

$$\prod_{t=1}^{T} \sqrt{1 - 4\gamma_t^2} \leqslant \exp\left(-2\sum_{t=1}^{T} \gamma_t^2\right)$$

fraction of the m examples, where the advantage $\gamma_t = \frac{1}{2} - \varepsilon_t$.

This theorem basically says that if we can always have a decent weak hypothesis, then AdaBoost does a really good job at driving the error rate down. This theorem also emphasises how AdaBoost is adaptive—the better the weak hypotheses are, the larger the γ_t are, and the better the final hypothesis is. Indeed we have the following corollary:

Corollary 20.1. *If each $\gamma_t \geqslant \gamma$, then we can run AdaBoost for*

$$T = \frac{1}{2\gamma^2} \ln\left(\frac{1}{\varepsilon}\right)$$

stages, and the final hypothesis H makes errors on at most an ε fraction of the m examples, and this is optimal in a strong sense. In particular, if $\varepsilon < 1/m$, then we converge to perfect classification on $x_1, \ldots, x_m$ given a γ-advantage weak learner.

Notice further that Theorem 20.1 is not saying anything about the generalisation error—the error AdaBoost makes on whatever distribution $\mathcal{D}$ the examples come from—but we have already spent quite a while talking about how, eg. a consistent hypothesis finder gives us a good generalisation error and so on and so forth, and we can combine those results with some of our newer machinery to combine statements about boosters over a fixed sample such as AdaBoost and statements about generalisation error, etc.

Now we prove Theorem 20.1, via the help of the following three lemmas.

Lemma 20.2. *The fraction of points i such that $H(x_i) \neq y_i$ is at most an exponential loss function over the examples:*

$$\frac{1}{m} |\{i \in [m] : H(x_i) \neq y_i\}| \leqslant \frac{1}{m} \cdot \sum_{i=1}^{m} \exp\left(-y_i \cdot f(x_i)\right),$$

where $f(x) = \sum_{t=1}^{T} \alpha_t h_t(x)$ and $H(x) = \operatorname{sign}(f(x))$.

Proof. Suppose that i is such that $H(x_i) \neq y_i$. Then $\operatorname{sign}(f(x_i)) \neq y_i$, so $y_i \cdot f(x_i) < 0$, so that $-y_i \cdot f(x_i) > 0$, and so $\exp(-y_i \cdot f(x_i)) > 1$. On the other hand, if i is such that $H(x_i) = y_i$, then $|\{i \in [m] : H(x_i) \neq y_i\}| = 0$, and so the inequality holds trivially (since $\exp(\cdot) \geqslant 1$). So the inequality holds trivially. Summing over all i gives the result (the left-hand side's contribution from each i is at most the right-hand side's contribution from that i). □

Lemma 20.3. *The exponential loss function over the examples is at most the product of the renormalisation factors over the stages:*

$$\frac{1}{m} \cdot \sum_{i=1}^{m} \exp\left(-y_i \cdot f(x_i)\right) \leqslant \prod_{t=1}^{T} Z_t.$$

Proof. Recall the definition of the very last distribution constructed, $\mathcal{D}_{T+1}$:

$$\begin{aligned}
\mathcal{D}_{T+1}(i) &= \frac{\exp\left(-\alpha_T y_i h_T(x_i)\right)}{Z_T} \cdot \mathcal{D}_T(i) \\
&= \frac{\exp\left(-\alpha_T y_i h_T(x_i)\right)}{Z_T} \cdot \frac{\exp\left(-\alpha_{T-1} y_i h_{T-1}(x_i)\right)}{Z_{T-1}} \cdot \mathcal{D}_{T-1}(i) \\
&= \frac{\exp\left(-\alpha_T y_i h_T(x_i)\right)}{Z_T} \cdot \ldots \cdot \frac{\exp\left(-\alpha_1 y_i h_1(x_i)\right)}{Z_1} \cdot \mathcal{D}_1(i) \\
&= \frac{\exp\left(-\sum_{t=1}^{T} \alpha_t y_i h_t(x_i)\right)}{\prod_{t=1}^{T} Z_t} \cdot \mathcal{D}_1(i) \\
&= \frac{\exp\left(-\sum_{t=1}^{T} \alpha_t y_i h_t(x_i)\right)}{\prod_{t=1}^{T} Z_t} \cdot \frac{1}{m},
\end{aligned}$$

where the last equality follows from the fact that $\mathcal{D}_1$ is a distribution equidistributed over the examples. Summing over all $i \in [m]$, we get, since $\sum_{i=1}^{m} \mathcal{D}_{T+1}(i) = 1$,

$$\frac{1}{m} \cdot \sum_{i=1}^{m} \exp\left(-\sum_{t=1}^{T} \alpha_t y_i h_t(x_i)\right) = \prod_{t=1}^{T} Z_t.$$

Since $f(x_i) = \sum_{t=1}^{T} \alpha_t h_t(x_i)$,

$$\frac{1}{m} \cdot \sum_{i=1}^{m} \exp\left(-y_i \cdot f(x_i)\right) = \prod_{t=1}^{T} Z_t,$$

as required. □

Notice that in the proofs of the last two lemmas, we haven't used anything about the specific setting of α_t; we merely carried the proofs on. However, it will be crucial for the following lemma.

Lemma 20.4. *The product of the renormalisation factors over the stages is*

$$\prod_{t=1}^{T} Z_t \leqslant \prod_{t=1}^{T} \sqrt{1 - 4\gamma_t^2}.$$

Proof. Define

$$Z_t = \sum_{i=1}^{m} \mathcal{D}_t(i) \cdot \exp\left(-\alpha_t y_i h_t(x_i)\right) = A + B,$$

where

$$\begin{aligned} A &= \sum_{i\,:\,h_t(x_i)=y_i} \mathcal{D}_t(i) \cdot \exp\left(-\alpha_t y_i h_t(x_i)\right) \\ B &= \sum_{i\,:\,h_t(x_i)\neq y_i} \mathcal{D}_t(i) \cdot \exp\left(-\alpha_t y_i h_t(x_i)\right). \end{aligned}$$

Now we saw at the start of lecture that $\varepsilon_t = \sum_{i\,:\,h_t(x_i)\neq y_i} \mathcal{D}_t(i)$ and $1 - \varepsilon_t = \sum_{i\,:\,h_t(x_i)=y_i} \mathcal{D}_t(i)$, and we also saw that for each of the i in the A-sum, $\exp\left(-\alpha_t y_i h_t(x_i)\right) = \sqrt{\frac{1-\varepsilon_t}{\varepsilon_t}}$, and for each of the i in the B-sum, $\exp\left(-\alpha_t y_i h_t(x_i)\right) = \sqrt{\frac{\varepsilon_t}{1-\varepsilon_t}}$. So

$$A = \sqrt{\frac{1-\varepsilon_t}{\varepsilon_t}} \cdot (1 - \varepsilon_t) = \sqrt{\varepsilon_t(1-\varepsilon_t)},$$

and

$$B = \sqrt{\frac{\varepsilon_t}{1-\varepsilon_t}} \cdot \varepsilon_t = \sqrt{\varepsilon_t(1-\varepsilon_t)}.$$

Therefore $Z_t = A + B = 2\sqrt{\varepsilon_t(1-\varepsilon_t)} = \sqrt{4\varepsilon_t(1-\varepsilon_t)}$, and we are nearly done. Indeed, since $\gamma_t = \frac{1}{2} - \varepsilon_t$, we have

$$\varepsilon_t = \frac{1}{2} - \gamma_t \iff 2\varepsilon_t = 1 - 2\gamma_t \iff \begin{cases} 2\varepsilon_t = 1 - 2\gamma_t, \\ 2(1-\varepsilon_t) = 1 + 2\gamma_t, \end{cases}$$

so that $Z_t = \sqrt{4\varepsilon_t(1-\varepsilon_t)} = \sqrt{(1-2\gamma_t)(1+2\gamma_t)} = \sqrt{1-4\gamma_t^2}$, as required. The result follows by taking the product over all t. □

The proof of Theorem 20.1 now follows by combining the three lemmas.

Proof of Theorem 20.1. Consider Lemmas 20.2, 20.3, 20.4, and recall that $1 - x \leqslant e^{-x} \implies \sqrt{1-x} \leqslant e^{-x/2}$. Then with $x = 4\gamma_t^2$, we have

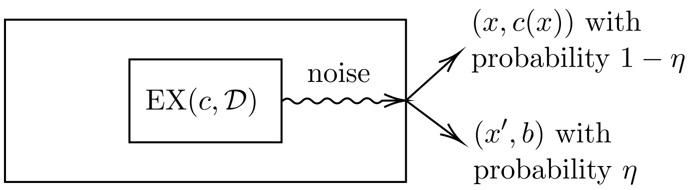
$$\prod_{t=1}^{T} Z_t \leqslant \prod_{t=1}^{T} \sqrt{1 - 4\gamma_t^2} \leqslant \exp\left(-2\sum_{t=1}^{T} \gamma_t^2\right),$$

as required. $\square$

§20.2 PAC Learning in the Presence of Noise

The motivation for this unit of the course is natural: we have been talking about learning where we have perfect data, but in the real world, we often have noisy data. We will start by discussing a general framework for noise, and then we will discuss some particular noise models.

The general framework we consider is as follows. We have a domain set X, a target concept $c\colon X \to \{0,1\}$ residing in a concept class $\mathcal{C}$, and a distribution $\mathcal{D}$ over X, and we still want an ε-accurate hypothesis $h \in \mathcal{H}$ with respect to c under $\mathcal{D}$ with probability at least $1-\delta$. But now the learner has access to a noisy example oracle that gives the learner examples $(x, c(x))$ at noise rate η, where the noise rate $\eta \in (0,1)$ is the probability that the oracle gives the learner a noisy example.


EX$(c, \mathcal{D})$
noise
$(x, c(x))$ with probability $1-\eta$
(x', b) with probability η

Here are two different noise models in this framework which we will discuss in this course:

1. *Random (mis)classification noise* [GS91]: The oracle gives the learner a noisy example $(x, c(x))$ with probability $1-\eta$, and with probability η, the oracle gives the learner a random example (x', b), where $x' = x$ and b is chosen uniformly at random from $\{0,1\}$ so that $b = \overline{c(x)}$.

2. *Malicious noise* [KL88]: The oracle gives the learner a noisy example $(x, c(x))$ with probability $1-\eta$, and with probability η, the oracle gives the learner a random example (x', b), where $x' = x$ and b is chosen arbitrarily at random from $\{0,1\}$ so that $b = c(x)$. We can view this as being generated by an omniscient malicious adversary who knows the target concept, its distribution, the current state of the learning procedure, etc, and is trying to make the learner's life as hard as possible.

For each of these two noise models, here's a high-level takeaway:

1. Random classification noise (RCN) is generally easy to deal with. In fact, we will see a general method that can let us convert mahy PAC learning algorithms into noise-tolerant PAC learning algorithms for RCN with only a small increase in the sample complexity, even at noise rates η that are close to $1/2$. (Notice that if $\eta \geqslant 1/2$, then learning is impossible, since even in the best case, whenever we get a label, it just amounts to discarding the label and receiving the outcome of a coin toss.)

2. Malicious noise is very challenging—think of it as dealing with a psychopath who is out to gaslight and confuse the learner by swarming them with misinformation. In fact, if $\eta \approx 2\varepsilon$,

then we cannot achieve error $\leqslant \varepsilon$. In fact, the best known methods to deal with malicious noise are extremely weak and usually reduce to just hoping that the noise rate is small. ☹

Example 20.5 (Toy scenario for learning with noise)**.** Consider two worlds. In world 1, $\frac{3}{8}$-ths of the examples are positive, and in world 2, $\frac{5}{8}$-ths of the examples are positive, and we want to learn whether we're in world 1 or world 2. If we're dealing with malicious noise at noise rate $\eta = \frac{1}{5}$. Then if an omniscient adversary gets to flip the label of 20% of the examples, then we're in trouble—he can flip the label of all the positive examples and a sufficient number of the negative examples, and so in both worlds, he can make the data look 50% positive and 50% negative. So we're in trouble; there is no hope for learning. On the other hand, if we're dealing with random classification noise at any $\eta < \frac{1}{2}$, then we can still learn. In fact, in world 1, we see positive examples with probability

$$\frac{3}{8} \cdot (1-\eta) + \frac{5}{8} \cdot \eta = \frac{3}{8} + \frac{1}{4}\eta,$$

and similarly in world 2, we see positive examples with probability

$$\frac{5}{8} \cdot (1-\eta) + \frac{3}{8} \cdot \eta = \frac{5}{8} - \frac{1}{4}\eta.$$

So as long as $\eta < \frac{1}{2}$, the probability $\frac{3}{8} + \frac{1}{4}\eta < \frac{1}{2}$ in world 1, and the probability $\frac{5}{8} - \frac{1}{4}\eta > \frac{1}{2}$ in world 2, and so the worlds are distinguishable; in particular, we can learn precisely which world we're in with $\approx (1-2\eta)^{-2}$ examples.

§21 Lecture 21—15th November, 2023

Last time. Finished the unit on boosting confidence and accuracy:

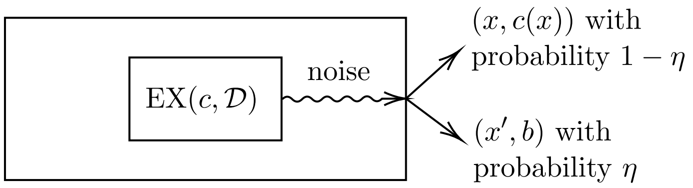
- Analysis of AdaBoost: state and prove a theorem about its performance.
- Start a new unit on PAC learning in the presence of noise:
  - malicious noise and random misclassification noise, both with some noise rate η.
  - a toy scenario: cannot handle malicious noise, but can handle random misclassification noise at any rate $\eta < 1/2$.

Today.

- A lower bound: learning with malicious noise is arbitrarily hard.
- A sad positive result for the hardness of learning with malicious noise.
- Start learning with random classification noise.

§21.1 PAC learning with malicious noise

Recall our setup with malicious noise:

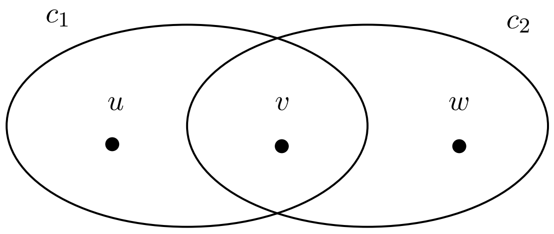


EX$(c, \mathcal{D})$
noise
$(x, c(x))$ with probability $1 - \eta$
(x', b) with probability η

In malicious noise, the labelled example (x', b) is generated with a completely arbitrary label b.

Let's now show an information-theoretic lower bound on the error achievable by any PAC learner in the presence of malicious noise.

Definition 21.1 (Distinct concept classes)**.** *A concept class $\mathcal{C}$ is said to be* distinct *if it contains concepts $c_1, c_2 \in \mathcal{C}$ and the domain X contains three points u, v, w such that $c_1(u) = 1 \neq c_2(u)$, $c_1(v) = c_2(v) = 1$, and $c_1(w) = 0 \neq c_2(w)$.*



This definition is useful for the following theorem:

Theorem 21.1. *Let $\mathcal{C}$ be a distinct concept class. Suppose the malicious noise rate $\eta \geqslant \frac{2\tau}{1+2\tau}$ for some $\tau > 0$. Then there is no algorithm that can PAC learn $\mathcal{C}$ (with malicious noise rate η) to error $< \tau$ and confidence $1 - \delta > \frac{1}{2}$.*

Proof. To start, only consider the concepts $c_1, c_2 \in \mathcal{C}$. We will define a distribution $\mathcal{D}$, as well as adversaries A_1 and A_2, so that if $\eta = \frac{2\tau}{1+2\tau}$, then to the learner, learning the concept c_1 under $\mathcal{D}$ against A_1 is perfectly indistinguishable from learning the concept c_2 under $\mathcal{D}$ against A_2. This is our plan for the proof.

Let $\mathcal{D}$ be the distribution that assigns probability weight τ on u, $1 - 2\tau$ on v, and $1 - 2\tau$ on w. The strategy for adversary A_1 is as follows: of the adversary's η "noise budget", it uses $\eta/2$ of the noise budget to mislabel u as negative (i.e. returns $(u, -)$), and the remaining $\eta/2$ to mislabel w as positive (i.e. returns $(w, +)$). Then the statistics of $(x, c_1(x))$ which the learner sees under $\mathcal{D}$ and adversary strategy A_1 is as follows:

Probability weight...	...on example	Noisy?
$\eta/2$	$(u, -)$	Yes
$(1 - 2\eta)\tau$	$(u, +)$	No
$(1 - \eta)(1 - 2\tau)$	$(v, +)$	No
$(1 - \eta)\tau$	$(w, -)$	No
$\eta/2$	$(w, +)$	Yes

In the other setting where the target concept is c_2 and the adversary is A_2, the goal of the adversary is then to match the statistics of the learner's observations as closely as possible to the table above in as obfuscated a way as possible. The adversary A_2 achieves this by using $\eta/2$ of its noise budget to mislabel u as positive, and the remaining $\eta/2$ to mislabel w as negative. Then the statistics of $(x, c_2(x))$ which the learner sees under $\mathcal{D}$ and adversary strategy A_2 is as follows:

Probability weight...	...on example	Noisy?
$\eta/2$	$(u, +)$	Yes
$(1-2\eta)\tau$	$(u, -)$	No
$(1-\eta)(1-2\tau)$	$(v, +)$	No
$(1-\eta)\tau$	$(w, +)$	No
$\eta/2$	$(w, -)$	Yes

So if, say, the probability weights of $(u, -)$ are the same under the actions of A_1 and A_2, that is, $\eta/2 = (1-2\eta)\tau$, then the weight of $(u, -)$ is the same under both adversaries (and in fact the same for all of the five labelled examples). Solving this gives

$$\eta = 2\tau - 2\tau\eta \implies \eta(1+2\tau) = 2\tau \implies \eta = \frac{2\tau}{1+2\tau},$$

and so if this is the noise rate, both adversaries are indistinguishable to the learner (note that if the noise rate was more than η, then the adversaries can simply use enough noise to make the learner's observations completely random and use the rest of the noise budget to build a spacecraft or something equally weird).

So it is impossible for the learner to get any information about whether the target concept is c_1 or c_2 from the observations. In fact, we cannot do better than random guessing, as we now show. Suppose world 1 or world 2 was chosen randomly with equal probability by a coin toss. Then the learner:

- can say "$h = 1$ always," thereby incurring error τ with probability 1, or
- can say "$h = 1$ on v and $h = 0$ on u and w," thereby incurring error τ with probability 1, or
- can say "$h = c_1$ always," thereby incurring error 2τ with probability 1/2 and error 0 with probability 1/2.
- can say "$h = c_2$ always," thereby incurring error 2τ with probability 1/2 and error 0 with probability 1/2.

But no matter which of these the learner does, for a concept $c \in \mathcal{C}$,

$$\Pr_{(x,c(x))\sim\mathcal{D}}[\mathrm{err}[h,c] \geqslant \tau] \geqslant \frac{1}{2},$$

and so the learner cannot achieve error $< \tau$ with confidence $1 - \delta > \frac{1}{2}$. $\square$

What then can we do about malicious noise? The above theorem shows that malicious noise can be arbitrarily hard to learn from. However, we can still try to learn from it, and in fact we can do so with a positive result (which we will toil to get to). Consider the generic scenario where we have a PAC learner A for $\mathcal{C}$ which works in the noiseless setting, but now is tasked to learn from an oracle with malicious noise. There are only two strategies we have here:

1. hope that there is no noise in the examples received by the algorithm, or
2. repeat multiple (k) runs of the algorithm, hope that for each run the examples are not generated from the malicious noise distribution, and then do hypothesis testing over the k hypotheses returned by the k runs.

The bad news here for the hypothesis testing case is that the labelled examples themselves may be infected with malicious noise, and so hypotheses which look like they're doing the best on the examples may not be doing the best on the true distribution. How well could we then hope to do with this strategy? We provide a rough sketch now.

As always, we want to obtain an (ϵ, δ)-PAC learner. Say that m is the number of examples needed to obtain a $(\frac{\varepsilon}{2}, \frac{\delta}{4})$-PAC learner in the noiseless setting. Then we can run the algorithm on m examples once. Then

$$\Pr[\text{one run has some noisy example}] = 1 - (1-\eta)^m,$$

and so

$$\Pr[\text{each of } k \text{ independent runs has at least one noisy example}] = \left(1 - (1-\eta)^m\right)^k.$$

Now if $\eta = \frac{\ln m}{m}$ and $k = m \cdot \ln(\frac{2}{\delta})$, then

$$\begin{aligned}(1-\eta)^m &= \left(1 - \frac{\ln m}{m}\right)^m \\ &\approx \left(e^{-\frac{\ln m}{m}}\right)^m = \frac{1}{m},\end{aligned}$$

so that the probability that each of k independent runs has at least one noisy example is

$$\begin{aligned}\left(1-(1-\eta)^m\right)^k &\approx \left(1 - \frac{1}{m}\right)^{m\ln(\frac{2}{\delta})} \\ &\approx \left(e^{-\frac{1}{m}}\right)^{m\ln(\frac{2}{\delta})} = \left(\frac{1}{e}\right)^{\ln(\frac{2}{\delta})} \\ &= \frac{1}{2/\delta} \\ &= \frac{\delta}{2}.\end{aligned}$$

So of the hypotheses $h_1, \ldots, h_k$, with probability $\geqslant 1 - \frac{\delta}{2}$, with probability $1 - \frac{\delta}{4}$, some hypothesis h_i has error $\leqslant \frac{\varepsilon}{2}$, and we only need to find such a hypothesis $h \in \{h_1, \ldots, h_k\}$, despite the additional uncertainty in our estimate of h_j from sampling even if there was no noise; the adversary can still skew our perception of what the true accuracy is by up to η:

So we have an additional η error, and this enforces $\eta \leqslant \frac{\varepsilon}{8}$ as well.

Overall, we can handle η up to

$$\approx \min\left\{\frac{\varepsilon}{8}, \frac{\ln m}{m}\right\},$$

and so when all the dust settles, we can handle η up to $\frac{\varepsilon}{8}$, which is not too bad.

In fact, the method just described is the best known generic technique for learning with malicious noise; with more knowledge about what the concept class is or which distribution it is learning over, we can do more, but this is the best we can do in the generic case.

§21.2 PAC learning with random classification noise

Recall how random classification noise works: from an oracle $\mathrm{EX}^\eta(c, \mathcal{D})$, the oracle returns $(x, c(x))$ with probability $1-\eta$ and $\left(x, \overline{c(x)}\right)$ with probability η. In some sense this is a predictable noise process; if noise happens we can still predict what the label should have been by flipping the label. The point though is that the learner does not know when the noise is happening, and still the learner has to learn from the examples it gets.

Definition 21.2 (PAC learning with random classification noise [AL88])**.** *An algorithm A* PAC learns a concept class $\mathcal{C}$ in the presence of random classification noise *if for all $c \in \mathcal{C}$, for all distributions $\mathcal{D}$ over the domain X, for all $\varepsilon, \delta \in (0,1)$, and for all $\eta \in [0, \frac{1}{2})$, if the algorithm A is given ε, δ and η, as well as access to an oracle $\mathrm{EX}^\eta(c, \mathcal{D})$, then with probability $\geqslant 1-\delta$, the algorithm outputs a hypothesis h such that*

$$\Pr_{x\sim\mathcal{D}}[h(x) \neq c(x)] \leqslant \varepsilon.$$

An algorithm of this type is efficient if it runs in time $\mathrm{poly}\left(\frac{1}{\varepsilon}, \log(\frac{1}{\delta}), \mathrm{size}(c), n, \frac{1}{1-2\eta}\right)$. The last quantity captures the idea that as the noise rate gets higher and higher, it should take longer and longer to learn from the examples; in fact, as the noise rate approaches $\frac{1}{2}$, it should take infinitely long to learn from the examples.

Remark 21.3. A few observations about this setup:

1. The goal is to do well on *noiseless* examples; we don't care about achieving high prediction accuracy on noisy examples.
2. The observed error rate of the noisy data is a linear function of the true error rate of the learner on the noiseless data. In particular, suppose that the true error rate of h on c is τ, i.e.

   $$\Pr_{x\sim\mathcal{D}}[h(x) \neq c(x)] = \tau.$$

   Then in the world where we only get examples from the noisy oracle $\mathrm{EX}^\eta(c, \mathcal{D})$, the observed error rate of h on c is

   $$\begin{aligned}\Pr_{(x,y)\sim\mathrm{EX}^\eta(c,\mathcal{D})}[h(x) \neq y] &= (1-\eta)\Pr_{x\sim\mathcal{D}}[h(x) \neq c(x)] + \eta \Pr_{x\sim\mathcal{D}}[h(x) = c(x)] \\ &= (1-\eta)\tau + \eta(1-\tau) \\ &= \eta + \tau(1-2\eta).\end{aligned}$$

   This makes even clear the fact that we only have hope for learning when $\eta < \frac{1}{2}$. In fact, if $\tau_1 < \tau_2$, then the observed error rate of h on c_1 is less than the observed error rate of h on c_2—this is a good thing; it says that bad hypotheses look worse on the noisy data than good hypotheses. We have already done better than in the case of malicious noise.

3. It is assumed that the learner knows the noise rate η. This is not a ridiculous assumption, for if we don't know η, we can:
   - run the algorithm with different guesses $\Delta, 2\Delta, 3\Delta, \dots, \frac{1}{2} - \Delta$ (for $\Delta = 1/\text{poly}(n)$) for η for each of the hypotheses $h_1, h_2, \dots, h_{\frac{1}{2\Delta}}$ returned by the algorithm—at least one of these guesses is close enough.
   - do hypothesis testing over the $h_1, h_2, \dots, h_{\frac{1}{2\Delta}}$ to find the best hypothesis, and then use this hypothesis to estimate η.

Next time we will revisit a (now ancient for us) PAC learner (the elimination algorithm) for learning monotone conjunctions, but now in the model $\text{EX}^{\eta}(c, \mathcal{D})$, and then modify it to only use estimates of noiseless probabilities (which we will show we can estimate), even in the presence of noise. We will then show that this algorithm is a PAC learner in the presence of random classification noise.

§22 Lecture 22—20th November, 2023

Last time.

- A lower bound: learning with malicious noise is arbitrarily hard.
- A sad positive result for the hardness of learning with malicious noise.
- Start learning with random classification noise.

Today. Methods for handling random classification noise.

- Revisit an old PAC learning algorithm (for learning monotone conjunctions) that is not RCN-tolerant, and adapt it to handle random classification noise.
- By so doing, we will introduce a new learning model: *statistical query learning.*
- Show that any SQ learning algorithm automatically yields a PAC learning algorithm that is tolerant to random classification noise.

§22.1 Handling RCN for PAC learning: learning monotone conjunctions

Recall the our old OLMB algorithm (we showed that every OLMB algorithm is also a PAC learning algorithm), the elimination algorithm, for learning monotone conjunctions. The algorithm is as follows:

Algorithm (Elimination Algorithm):

1. Set the initial hypothesis $h(x) = x_1 \wedge \ldots \wedge x_n$.
2. Get an example $z \in \{0,1\}^n$, and predict $h(z)$.
3. Given the true label $c(z)$, update the hypothesis h as follows:
   - (False positive mistake.) If $h(z) = 1$ and $c(z) = 0$, remove x_i from h for all i such that $z_i = 1$.

- (False negative mistake.) If $h(z) = 0$ and $c(z) = 1$, stop and fail.
- (Correct hypothesis.) If $h(z) = c(z)$, then keep h the same.

In the PAC framework, this algorithm is the elimination algorithm:

Algorithm (Elimination Algorithm):

1. Set the initial hypothesis $h(x) = x_1 \wedge \ldots \wedge x_n$.
2. Draw a set S of m examples $z_1, \ldots, z_m \in \{0,1\}^n$ from $\mathrm{EX}(c, \mathcal{D})$.
3. For each positive example in S, if $x_i = 0$, remove x_i from $h(x)$.

In the noiseless world we have already shown and proved that this algorithm gives us a consistent monotone conjunction. But what happens when we have random classification noise? In this case the noisy oracle $\mathrm{EX}^\eta(c, \mathcal{D})$ may give us the wrong label for an example (with probability η), and thus we may permanently eliminate a valuable literal x_i in the target concept because the algorithm insists that we make a decision for each new example, whether it is a false positive or a false negative. This is not good—in fact we may be left with the empty conjunction, which is clearly not a good hypothesis.

An intuitive fix for this is to instead look at the aggregate statistical properties of the data set. More precisely, define

$$p_i = \Pr_{(x, c(x)) \sim \mathrm{EX}(c,\mathcal{D})} [c(x) = 1 \text{ and } x_i = 0].$$

If x_i is in the target conjunction, then $p_i = 0$. Intuitively, each x_i that isn't in c but is in a hypothesis h should add $\leqslant p_i$ (due to overlap) to the error of a hypothesis h. So it is good enough to identify (and subsequently eliminate from h) all the x_i for which $p_i \geqslant \varepsilon/n$. If we could do this, then the total error of h is at most the product of the error of each erroneously included x_i and the bound on the number of such x_i, that is, the total error is

$$\leqslant \frac{\varepsilon}{n} \cdot n = \varepsilon.$$

So to PAC-learn monotone conjunctions in the presence of random classification noise, it is sufficient to estimate each p_i to within $\pm\varepsilon/2n$. This is our goal.

If there was no noise, then we're in good shape: we can draw m examples from $\mathrm{EX}(c, \mathcal{D})$, compute

$$\hat{p}_i = \text{fraction of the } m \text{ examples such that } c(x) = 1 \text{ and } x_i = 0,$$

and use $\hat{p}_i$ as an estimate of p_i. By the additive Chernoff bound, we know that

$$\Pr[|\hat{p}_i - p_i| \geqslant \tau] \leqslant 2e^{-2m\tau^2} \implies \Pr[|\hat{p}_i - p_i| \geqslant \tau] \leqslant \delta,$$

when $m \geqslant \frac{1}{2\tau^2} \ln\left(\frac{2}{\delta}\right)$. We have to do this for all the n variables, so we can take $\tau = \varepsilon/2n$, and we want each i to fail with probability $\leqslant \delta/n$. So by a union bound, we need

$$m \geqslant \frac{n}{2\tau^2} \ln\left(\frac{2n}{\delta}\right) = \frac{n^2}{2\varepsilon^2} \ln\left(\frac{2n}{\delta}\right)$$

examples to estimate each p_i to within $\pm\varepsilon/2n$ with probability $\geqslant 1 - \delta$.

But what do we do if there's noise? Write

$$\begin{aligned} p_i &= \Pr_{(x,c(x))\sim \mathrm{EX}(c,\mathcal{D})}[c(x) = 1 \text{ and } x_i = 0] \\ &= \Pr_{(x,c(x))\sim \mathrm{EX}(c,\mathcal{D})}[c(x) = 1 \mid x_i = 0] \cdot \Pr_{(x,c(x))\sim \mathrm{EX}(c,\mathcal{D})}[x_i = 0]. \end{aligned}$$

Observe that since the event "$x_i = 0$" is entirely unaffected by noise, we have

$$\Pr_{(x,c(x))\sim \mathrm{EX}(c,\mathcal{D})}[x_i = 0] = \Pr_{(x,c(x))\sim \mathrm{EX}^\eta(c,\mathcal{D})}[x_i = 0],$$

and so we can estimate it given $\mathrm{EX}^\eta(c, \mathcal{D})$. Now suppose $\Pr_{(x,c(x))\sim \mathrm{EX}(c,\mathcal{D})}[x_i = 0]$ is small; in particular, suppose

$$\Pr_{(x,c(x))\sim \mathrm{EX}(c,\mathcal{D})}[x_i = 0] \leqslant \frac{\varepsilon}{4n}.$$

Then we can estimate that $p_i \lesssim \frac{\varepsilon}{4n}$, and we have a good-enough estimate of p_i.

So suppose that the quantity is not small, that is,

$$\Pr_{(x,c(x))\sim \mathrm{EX}(c,\mathcal{D})}[x_i = 0] \geqslant \frac{\varepsilon}{4n}.$$

Define $q_i := \Pr_{(x,b)\sim \mathrm{EX}(c,\mathcal{D})}[b = 1 \mid x_i = 0]$. Then we have

$$\begin{aligned} n_i &:= \Pr_{(x,b)\sim \mathrm{EX}^\eta(c,\mathcal{D})}[b = 1 \mid x_i = 0] \\ &= q_i \cdot (1-\eta) + (1-q_i)\cdot \eta \\ &= \eta + (1-2\eta)q_i. \end{aligned}$$

So if we can estimate n_i well and we know η, then we can estimate q_i as well, and we can estimate p_i as well, and we'll be done. But there is one caveat: n_i is a conditional probability, so we can only use draws from $\mathrm{EX}^\eta(c, \mathcal{D})$ that have $x_i = 0$ towards estimating n_i; in particular, we need, on average, $\left(\Pr_{(x,c(x))\sim \mathrm{EX}(c,\mathcal{D})}[x_i = 0]\right)^{-1}$ examples from $\mathrm{EX}^\eta(c, \mathcal{D})$ to get one satisfying $x_i = 0$. The stumbling block for efficiency then arises when $\Pr_{(x,c(x))\sim \mathrm{EX}(c,\mathcal{D})}[x_i = 0]$ is small. But we already saw that in this case, we can estimate p_i well enough if it is $\leqslant \varepsilon/4n$. So we can just ignore the x_i for which $\Pr_{(x,c(x))\sim \mathrm{EX}(c,\mathcal{D})}[x_i = 0]$ is small, and estimate p_i for the rest.

So we can efficiently estimate p_i for all i to within $\pm\varepsilon/2n$ with probability $\geqslant 1-\delta$ using enough examples from $\mathrm{EX}^\eta(c, \mathcal{D})$, and then we can use the elimination algorithm to output a hypothesis h that is ε-close to c with probability $\geqslant 1-\delta$. What exactly are the key highlights in what we've just demonstrated?

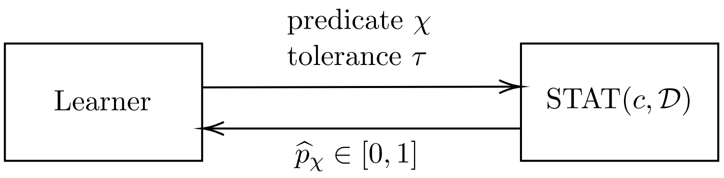
1. We reduced a noisy learning problem to the problem of estimating some (noiseless) probabilities in a noisy world.
2. We decomposed the desired probability into:
   a) a part unaffected by noise (that is, $\Pr_{(x,c(x))\sim \mathrm{EX}(c,\mathcal{D})}[x_i = 0]$), and
   b) a part completely/predictably affected by noise (that is, $\Pr_{(x,b)\sim \mathrm{EX}(c,\mathcal{D})}[b = 1 \mid x_i = 0]$).

   Then we handled each part separately and combined the results.

The next learning model generalises this approach.

§22.2 Introduction to statistical query learning

We now begin to introduce a new learning model due to [Kea98], *statistical query learning*, where we don't think about the regular PAC learning problem but direct our attention to estimating certain statistical properties of the data distribution, like we did in the previous section. In this model the algorithm no longer has access to examples from the oracle $\mathrm{EX}(c, \mathcal{D})$, but can only request estimates of probabilities[2] from a new oracle, $\mathrm{STAT}(c, \mathcal{D})$.



Definition 22.1 (Predicate)**.** *A* predicate *of a labelled example (x, y) is a statement that is either* `true` *or* `false` *for (x, y). Formally, a predicate is a function $\chi : X \times Y \to \{0,1\}$ (we will only consider $Y = \{0,1\}$).*

Example 22.2. An example of a predicate is "(x, y) has $x_1 = x_2 = 1$ and $y = 0$".

Let the true probability of a predicate χ be

$$p_\chi := \Pr_{(x,c(x))\sim \mathrm{EX}(c,\mathcal{D})}[\chi(x, c(x)) = 1].$$

The following explains the actions of the new statistical query oracle $\mathrm{STAT}(c, \mathcal{D})$:

Definition 22.3. *The oracle $\mathrm{STAT}(c, \mathcal{D})$ takes in a predicate χ and a tolerance $\tau \in [0,1]$ as the inputs from the learner, and returns a value $\widehat{p}_\chi \in [0,1]$ such that*

$$|\widehat{p}_\chi - p_\chi| \leqslant \tau.$$

Note that if we had $\mathrm{EX}(c, \mathcal{D})$, we can simulate $\mathrm{STAT}(c, \mathcal{D})$ with failure probability $\leqslant \delta$ by the following procedure: given inputs χ and τ,

- draw m examples from $\mathrm{EX}(c, \mathcal{D})$, and evaluate χ on each example,
- return
$$\widehat{p}_\chi = \frac{\text{number of the } m \text{ examples that satisfy } \chi}{m}.$$

By the Chernoff bound, we have that by taking $m = O\left(\frac{\log(\frac{1}{\delta})}{\tau^2}\right)$, with probability $\geqslant 1-\delta$, the estimate $\widehat{p}_\chi$ is within $\pm\tau$ of p_χ. So we can run statistical query algorithms in the probably approximately correct learning model, and we can also run PAC learning algorithms in the statistical query learning model. Later on we will show that any statistical query learning algorithm automatically yields a PAC learning algorithm that is tolerant to random classification noise.

[2] For now, don't think about noise; we'll get to that later.

Let's now say more about the efficiency of statistical query learning. If a learner makes a call (x, τ) to $\text{STAT}(c, \mathcal{D})$, then the learner should be "charged" an amount of time that would be required to simulate $\text{STAT}(c, \mathcal{D})$ on its own if it had access to $\text{EX}(c, \mathcal{D})$. This is because the learner could have just simulated $\text{STAT}(c, \mathcal{D})$ on its own, and so the learner should not be able to do anything more efficiently with $\text{STAT}(c, \mathcal{D})$ than it could have done with $\text{EX}(c, \mathcal{D})$.

So for a statistical query learning algorithm to be efficient, it should:

- not make too many calls to $\text{STAT}(c, \mathcal{D})$, and
- make each call (χ, τ) to $\text{STAT}(c, \mathcal{D})$ with:
    - a τ that is not too small, and
    - a χ that is efficiently computable and easy to evaluate.

We are now ready to define the statistical query learning model.

Definition 22.4 (Statistical query learning). *A concept class $\mathcal{C}$ is* learnable from statistical queries *(or SQ-learnable) if there's a learning algorithm L such that for any concept $c \in \mathcal{C}$, for any distribution $\mathcal{D}$ over X, and for any $\varepsilon > 0$, if L is given access to* $\text{STAT}(c, \mathcal{D})$*, then L outputs a hypothesis h such that*

$$\Pr_{x \sim \mathcal{D}}[h(x) \neq c(x)] \leqslant \varepsilon.$$

Note that here we do not have a δ parameter, because STAT has no failure probability.

Definition 22.5 (Efficient SQ-learning). *A concept class $\mathcal{C}$ is* efficiently (poly-time) SQ-learnable *by a learning algorithm L if:*

- *every query (χ, τ) the algorithm L makes to* $\text{STAT}(c, \mathcal{D})$ *is such that $\chi(x, y)$ can be evaluated in* $\text{poly}\left(n, \text{size}(c), \frac{1}{\varepsilon}\right)$ *for each x, y, and*

$$\tau \geqslant \frac{1}{\text{poly}\left(n, \text{size}(c), \frac{1}{\varepsilon}\right)},$$

- *L runs in time* $\text{poly}\left(n, \text{size}(c), \frac{1}{\varepsilon}\right)$*, where we view each call to* $\text{STAT}(c, \mathcal{D})$ *as running in unit time.*

All of the above gives us the equivalence we have already hinted at:

Theorem 22.1. *Suppose a concept class $\mathcal{C}$ is efficiently SQ-learnable. Then $\mathcal{C}$ is efficiently PAC-learnable.*

Proof. The SQ-learning algorithm makes, say, $M = \text{poly}\left(n, \text{size}(c), \frac{1}{\varepsilon}\right)$ calls to $\text{STAT}(c, \mathcal{D})$, with each tolerance

$$\tau \geqslant \tau_0 := \frac{1}{\text{poly}\left(n, \text{size}(c), \frac{1}{\varepsilon}\right)}.$$

Each predicate is "reasonably" efficiently computable. Then in the PAC learning scenario, for each call to $\text{STAT}(c, \mathcal{D})$, we can simulate it with $\text{EX}(c, \mathcal{D})$ with failure probability

$$\Pr\left[|\widehat{p}_\chi - p_\chi| \geqslant \tau\right] \leqslant \frac{1}{M}.$$

This is polynomial in the parameters $n, \text{size}(c), \frac{1}{\varepsilon}, \log(\frac{1}{\delta})$. So by a Chernoff bound, we can simulate $\text{STAT}(c, \mathcal{D})$ with $\text{EX}(c, \mathcal{D})$ with failure probability $\leqslant \delta$ by running the simulation M times and taking the majority vote. This gives us a PAC learning algorithm. $\square$

§23 Lecture 23—27th November, 2023

Last time. Methods for handling random classification noise.

- Revisit an old PAC learning algorithm (for learning monotone conjunctions) that is not RCN-tolerant, and adapt it to handle random classification noise.
- By so doing, we motivated a new learning model: *statistical query learning.*
- Argued that any SQ learning algorithm automatically yields a PAC learning algorithm that is tolerant to random classification noise.

Today. More on SQ-learning:

- Efficient SQ learning algorithms yield efficient PAC learning algorithms that are tolerant to random classification noise.
  - many PAC learning algorithms can be rephrased as SQ learning algorithms, so we get RCN-tolerant PAC learning algorithms for free.
  - but not all PAC learning algorithms can be rephrased as SQ learning algorithms...
- Unconditional lower bounds for SQ learning: some concept classes are efficiently PAC learnable, but are provably not efficiently SQ learnable.

§23.1 Statistical query learning algorithms yield RCN-tolerant PAC learners

Remember the theorem we ended with last time:

Theorem 23.1. *Suppose a concept class $\mathcal{C}$ is efficiently SQ-learnable. Then $\mathcal{C}$ is efficiently PAC-learnable.*

Now, we introduce noise.

Theorem 23.2. *Let $\mathcal{C}$ be a concept class that is efficiently SQ-learnable. Then $\mathcal{C}$ is efficiently PAC-learnable in the presence of random classification noise, i.e. for any $0 \leqslant \eta < \frac{1}{2}$, the running time is polynomial in $\frac{1}{\varepsilon}$, $\log\left(\frac{1}{\delta}\right)$, n, $\text{size}(c)$, and $\frac{1}{1-2\eta}$.*

This is great news! Many PAC learning algorithms can be viewed as SQ algorithms, so we can get RCN-tolerant versions of perceptron, the algorithms for learning decision lists, conjunctions, disjunctions, LTFs, etc, for free.

Proof of Theorem 23.2. The key step for proving this theorem is to simulate a call to $\text{STAT}(c, \mathcal{D})$ given $\text{EX}^{\eta}(c, \mathcal{D})$, the noisy example oracle. That is, given access to a particular statistical query

(χ, τ) and access to $\mathrm{EX}^\eta(c, \mathcal{D})$, we want to estimate

$$p_\chi := \Pr_{x\sim\mathcal{D}}[\chi(x, c(x)) = 1]$$

accuracy up to $\pm\tau$. The challenge though is that we're in a noisy world, so it's not clear how to do this. The key idea for breaking through this challenge is to break up X into two disjoint sets: those where noise matters and those where it doesn't.

$$\begin{aligned} X_1 &= \{x \in X : \chi(x, 0) \neq \chi(x, 1)\} \\ X_2 &= \{x \in X : \chi(x, 0) = \chi(x, 1)\}. \end{aligned}$$

Clearly noise matters only for $x \in X_1$. Our key observation is then that, given x, we can check if $x \in X_1$ or $x \in X_2$ by computing $\chi(x, 0)$ and $\chi(x, 1)$ and checking if they agree or disagree (if they do, then $x \in X_2$; if they don't, then $x \in X_1$).

Example 23.1. Suppose $\chi(x, b) =$ "$b = 1$ and $x_1 = 0$". Then

$$\begin{aligned} X_2 &= \{x \in X : x_1 = 1\} \\ X_1 &= \{x \in X : x_1 = 0\}. \end{aligned}$$

Now some notation. Let

$$p_1 := \Pr_{x\sim\mathcal{D}}[x \in X_1],$$

and $p_2 = 1 - p_1$. Let $\mathcal{D}_1$ be the distribution $\mathcal{D}$ conditioned on X_1, i.e. for some set S,

$$\Pr_{x\sim\mathcal{D}_1}[x \in S] = \Pr_{x\sim\mathcal{D}}[x \in S \mid x \in X_1].$$

We want to argue that in the noisy world we can estimate the probability p_χ up to $\pm\tau$, and we will do so via the following two lemmas:

Lemma 23.2. *We have that $p_\chi = A \cdot B + C$, where*

$$\begin{aligned} A &= \Pr_{x\sim\mathcal{D}}[x \in X_1] \\ B &= \frac{\Pr_{(x,b)\sim\mathrm{EX}^\eta(c,\mathcal{D}_1)}[\chi(x, b) = 1] - \eta}{1 - 2\eta} \\ C &= \Pr_{(x,b)\sim\mathrm{EX}^\eta(c,\mathcal{D})}[\chi(x, b) = 1 \textit{ and } x \in X_2]. \end{aligned}$$

Proof. We have that

$$\begin{aligned} p_\chi &= \Pr_{x\sim\mathcal{D}}[\chi(x, c(x)) = 1] \\ &= \Pr_{x\sim\mathcal{D}}[x \in X_1 \text{ and } \chi(x, c(x)) = 1] + \Pr_{x\sim\mathcal{D}}[x \in X_2 \text{ and } \chi(x, c(x)) = 1] \\ &= \Pr_{x\sim\mathcal{D}}[x \in X_1] \Pr_{(x,b)\sim\mathrm{EX}^\eta(c,\mathcal{D})}[\chi(x, b) = 1 \mid x \in X_1] + \Pr_{x\sim\mathcal{D}}[x \in X_2 \text{ and } \chi(x, c(x)) = 1] \\ &= \underbrace{p_1}_{A} \cdot \Pr_{x\sim\mathcal{D}}[\chi(x, c(x)) = 1 \mid x \in X_1] + \underbrace{\Pr_{(x,b)\sim\mathrm{EX}^\eta(c,\mathcal{D})}[\chi(x, b) = 1 \text{ and } x \in X_2]}_{C}, \end{aligned}$$

where the last equality holds since noise does not matter for $x \in X_2$. We then need to show that

$$B = \Pr_{x \sim \mathcal{D}}[\chi(x, c(x)) = 1 \mid x \in X_1].$$

Note that

$$\Pr_{x \sim \mathcal{D}}[\chi(x, c(x)) = 1 \mid x \in X_1] = \Pr_{x \sim \mathcal{D}_1}[\chi(x, c(x)) = 1].$$

Then since noise is always a factor in X_1 (i.e. under $\mathcal{D}_1$), we have that

$$\begin{aligned}\Pr_{(x,b) \sim \mathrm{EX}^{\eta}(c, \mathcal{D}_1)}[\chi(x, b) = 1] &= (1 - \eta) \cdot \Pr_{x \sim \mathcal{D}_1}[\chi(x, c(x)) = 1] + \eta \cdot \Pr_{x \sim \mathcal{D}_1}[\chi(x, c(x)) = 0] \\ &= (1 - \eta) \cdot \Pr_{x \sim \mathcal{D}_1}[\chi(x, c(x)) = 1] + \eta \cdot (1 - \Pr_{x \sim \mathcal{D}_1}[\chi(x, c(x)) = 1]) \\ &= \eta + (1 - 2\eta) \cdot \Pr_{x \sim \mathcal{D}_1}[\chi(x, c(x)) = 1].\end{aligned}$$

We only need to solve for $\Pr_{x \sim \mathcal{D}_1}[\chi(x, c(x)) = 1]$ to get:

$$\Pr_{x \sim \mathcal{D}_1}[\chi(x, c(x)) = 1] = \frac{\Pr_{(x,b) \sim \mathrm{EX}^{\eta}(c, \mathcal{D}_1)}[\chi(x, b) = 1] - \eta}{1 - 2\eta} = B,$$

and the proof of the lemma is complete. □

The second lemma gives us the rest of the proof.

Lemma 23.3. *It is possible to estimate $A \cdot B + C$ up to $\pm\tau$ using $\mathrm{EX}^{\eta}(c, \mathcal{D})$, where A, B, and C are as defined in the previous lemma.*

Proof. We will find estimates for $A \cdot B$ and C separately, and then add them up. Given a noisy oracle, we can get an estimate of the probability of an element falling into the x_1 region as follows:

- Call $\mathrm{EX}^{\eta}(c, \mathcal{D})$ to get (x, b) where $x \sim \mathcal{D}$.
- Call $\chi(x, 0)$ and $\chi(x, 1)$ to determine if $x \in X_1$ or $x \in X_2$. If they are not equal, then $x \in X_1$.

So we are in a coin-toss type scenario: toss a coin, if it comes up heads, then this corresponds to the case where $\chi(x, 0) \neq \chi(x, 1)$, and so $x \in X_1$, and if it comes up tails, then $x \in X_2$. Then by the Chernoff bound we can estimate A with $\frac{1}{\tau^2} \log\left(\frac{1}{\delta}\right)$ coin tosses to get a good estimate of $p_1 = A$ up to $\pm\tau$ with probability at least $1 - \delta$.

But we want an efficient estimate of $A \cdot B$, not just A. We already know that

$$B = \frac{\Pr_{(x,b) \sim \mathrm{EX}^{\eta}(c, \mathcal{D}_1)}[\chi(x, b) = 1] - \eta}{1 - 2\eta} \leqslant \frac{1}{1 - 2\eta}.$$

Now if $p_1 < \gamma \cdot (1 - 2\eta)$, then $A \cdot B < \gamma$, and 0 is a $\pm\gamma$-approximation to $A \cdot B$ for some γ. So we're done unless our estimate of p_1 is too high, in particular $p_1 \geqslant \gamma \cdot (1 - 2\eta)$. Now, B involves random samples from $\mathcal{D}_1$, but we only have access to $\mathrm{EX}^{\eta}(c, \mathcal{D})$. We will try to attack this issue by making draws (x, b) from the oracle $\mathrm{EX}^{\eta}(c, \mathcal{D})$ and then checking if $x \in X_1$ or $x \in X_2$. If $x \in X_1$ (with probability p_1), then we have a draw from $\mathrm{EX}^{\eta}(c, \mathcal{D}_1)$ for free. Since $p_1 > \gamma \cdot (1 - 2\eta)$, then the slowdown of this filtering process is modest—it is $\leqslant (\gamma \cdot (1 - 2\eta))^{-1}$. So we can get an accurate estimate of B, and stepping back, we can get an accurate estimate of $A \cdot B$.

It then remains to estimate C. We can estimate C by drawing (x, b) from $\mathrm{EX}^{\eta}(c, \mathcal{D})$ and then computing $\chi(x, 0)$ and $\chi(x, 1)$ to check if $x \in X_2$ (which happens when $\chi(x, 0) = \chi(x, 1)$). We can then use the Chernoff bound to get that we need $\approx \frac{1}{\tau^2} \log\left(\frac{1}{\delta}\right)$ samples to get a good estimate of C up to $\pm\tau$ with probability at least $1 - \delta$. □

Combining Lemma 23.2 and Lemma 23.3, we can estimate p_χ up to $\pm\tau$ using $\mathrm{EX}^{\eta}(c, \mathcal{D})$, and this completes the proof of the theorem. □

So we have now seen that SQ learning algorithms yield RCN-tolerant PAC learning algorithms. This is great news; it motivates the following question: is every concept class that is efficiently PAC learnable also efficiently SQ learnable? The answer is a resounding NO, and we present the following example to illustrate this.

Example 23.4 (Impossibility of SQ learning for parity)**.** Recall the problem of learning parities: we are in the domain $X = \{0, 1\}^n$, we have a subset $S \subseteq [n]$ which is the parity function on S defined as

$$\mathrm{PAR}_S = \sum_{i \in S} x_i \mod 2.$$

The concept class of parity functions is $\mathcal{C}_{\mathrm{PAR}} = \{\text{all } 2^n\ \mathrm{PAR}_S \text{ functions}\}$. We have already seen in the homework that there exists a PAC learning algorithm for $\mathcal{C}_{\mathrm{PAR}}$ that is obtainable by Gaussian elimination modulo 2. In fact, it is provably impossible for any SQ learning algorithm to efficiently learn the concept class $\mathcal{C}_{\mathrm{PAR}}$, and this is intuitively true because it very much feels like PAR_S "looks at individual examples in the concept class". We will show this on the next homework by showing that $\mathcal{C}_{\mathrm{PAR}}$ contains arbitrarily many uncorrelated functions, and so cannot be efficiently learned by any SQ learning algorithm.

Next time, we will:

- discuss *uncorrelated* Boolean functions and why they are intuitively a hard case for learning in generic models that use statistical properties of a distribution to learn.
- show that if any concept class $\mathcal{C}$ contains arbitrarily many uncorrelated functions, then $\mathcal{C}$ is not efficiently SQ learnable by any algorithm.
- start a new unit on the computational/cryptographic hardness of learning.

§24 Lecture 24—29th November, 2023

Last time. Said more about SQ-learning:

- Efficient SQ learning algorithms yield efficient PAC learning algorithms that are tolerant to random classification noise.
  - many PAC learning algorithms can be rephrased as SQ learning algorithms, so we get RCN-tolerant PAC learning algorithms for free.
  - but not all PAC learning algorithms can be rephrased as SQ learning algorithms to learn, e.g., parity functions.
- Recalled that the concept class of all 2^n parity functions over n Boolean variables is PAC-learnable.

Today.

- Sketch a proof that if a concept class $\mathcal{C}$ and a distribution $\mathcal{D}$ are such that $\mathcal{C}$ contains many (about $n^{\omega(1)}$ many) functions that are all pairwise uncorrelated under $\mathcal{D}$, then there is no efficient SQ learning algorithm for $\mathcal{C}$ under $\mathcal{D}$.
  - problem five on the homework set: there is no efficient SQ learning algorithm for the class of all n-variable parity functions, DNFs, and decision trees under the uniform distribution.
- Start a unit on the cryptographic hardness of learning "rich" concept classes.

§24.1 Lower bounds for statistical query learning

Definition 24.1. *Let $\mathcal{D}$ be a distribution over $\{0,1\}^n$. We say that two concepts $c_1, c_2 \colon \{0,1\}^n \to \{0,1\}$ are* ε-uncorrelated *under $\mathcal{D}$ if*

$$\Pr_{x\sim\mathcal{D}}[c_1(x) = c_2(x)] \in [1/2-\varepsilon, 1/2+\varepsilon] \ni \Pr_{x\sim\mathcal{D}}[c_1(x) \neq c_2(x)].$$

In the homework set we will see a case where the concepts are *perfectly* uncorrelated under $\mathcal{D}$, that is, $\varepsilon = 0$. In particular, on the homework we will show that if $\mathcal{D}$ is the uniform distribution, then any two distinct parity functions $\mathrm{PAR}_{S_1}, \mathrm{PAR}_{S_2} \colon \{0,1\}^n \to \{0,1\}$ are perfectly uncorrelated under $\mathcal{D}$ if $S_1 \neq S_2$.

We will not rigourously prove the following fact, but we will give a proof sketch that should give a general idea for the proof.

Fact 24.2 (SQ algorithms cannot learn uncorrelated functions)**.** *Let $\mathcal{C}$ be a concept class over $X = \{0,1\}^n$. Let $\mathcal{D}$ be a distribution over $\{0,1\}^n$ such that $\mathcal{C}$ contains N concepts $c_1, \ldots, c_N$ such that, for all $1 \leqslant i < j \leqslant N$, the concepts c_i and c_j are perfectly uncorrelated under $\mathcal{D}$. Then any SQ learning algorithm for $\mathcal{C}$, even only achieving weak accuracy $\frac{1}{2} + \frac{1}{N^{1/3}}$, must either:*

- *make $\geqslant N^{1/3}$ queries, or*
- *make some SQ query with tolerance $\tau \leqslant \frac{1}{N^{1/3}}$.*

For example, if $\mathcal{C}$ is the class of all n-variable parity functions, then $\mathcal{C}$ contains $N = 2^n$ concepts, and so we would need to make $\geqslant 2^{n^{1/3}}$ queries or make some query with tolerance $\tau \leqslant \left(2^{n^{1/3}}\right)^{-1}$. This is too much to hope for in general, so we cannot hope to learn parity functions with an SQ algorithm (again, we show this more formally in the homework set).

Here's a sketch for why Fact 24.2 should be true. Establish the following analogies:

- A concept over $\{0,1\}^n \longleftrightarrow$ a unit vector in 2^n-dimensional space.

> **Example 24.3.** Suppose $n = 2$ and $c = \pm 1$ is a concept. Then we have the following truth table:
>
> | x_1 | x_2 | $c(x)$ |
> |---|---|---|
> | 0 | 0 | $+1$ |
> | 0 | 1 | -1 |
> | 1 | 0 | -1 |
> | 1 | 1 | $+1$ |
>
> Then c is the unit vector $\frac{1}{2}(-1,1,1,-1)$ in 4-dimensional space.

- Uncorrelated concepts $c_1, \ldots, c_N \longleftrightarrow$ orthogonal unit vectors $e_1, \ldots, e_N$ in 2^n-dimensional space.
- The predicate $\chi(x, c(x)) \longleftrightarrow$ a unit vector v in 2^n-dimensional space.
- The value $p_\chi = \Pr_{x\sim\mathcal{D}}[\chi(x, c(x)) = 1] \longleftrightarrow$ the inner product $\langle v, u\rangle$.
- The tolerance $\tau \longleftrightarrow$ the accuracy of the estimate $\langle v, u\rangle$.

With these analogies in mind, then the problem of SQ learning a concept class c is analogous to the problem of trying to estimate u by making successive queries of the form "what is $\langle v, u\rangle \approx_\tau$ to?" for different unit vectors v. The (sketch of the) proof of Fact 24.2 is then done, once we have established the following fact from high-school geometry:

Fact 24.4. *Let v be any unit vector in $\mathbb{R}^N$. Then, there are at most L^2 many coordinates i for which $|v_i| \geqslant \frac{1}{L}$.*

Proof. Let $v = (v_1, \ldots, v_N)$ be a unit vector in $\mathbb{R}^N$, meaning that $\|v\|^2 = 1$, where $\|v\|^2 = \sum_{i=1}^N v_i^2$. Suppose, for the sake of contradiction, that there are more than L^2 coordinates i for which $|v_i| \geqslant \frac{1}{L}$. Let M be the number of such coordinates, so that we have $M > L^2$.

For those M coordinates, the sum of their squares is at least $\frac{M}{L^2}$. This is because each $|v_i|^2 \geqslant \frac{1}{L^2}$ for these coordinates, and so:

$$\sum_{\text{those } i} v_i^2 \geqslant M \cdot \frac{1}{L^2}.$$

Since $M > L^2$, it follows that the sum of the squares of these coordinates is greater than 1. But this is a contradiction because the sum of the squares of all coordinates of v must equal 1, as v is a unit vector. Thus, it is not possible to have more than L^2 coordinates i for which $|v_i| \geqslant \frac{1}{L}$, proving the fact. $\square$

Given these facts then, here's the lower bound for SQ learning. Suppose that every statistical query is made to tolerance $\tau = 1/N^{1/3}$. Then the target concept is one of basis vectors $e_1, \ldots, e_N$ (i.e. the target vector u). The first SQ query made to the distribution $\mathcal{D}$ will return some vector $v^{(1)}$. Then the learner asks for the correlation of $v^{(1)}$ with the target unknown unit vector u, and the adversary's (again, remember that we are learning with noise) response will be "$\langle v^{(1)}, u\rangle \approx_\tau 0$." By Fact 24.4, there are at most $N^{2/3}$ coordinates i for which $\left|v_i^{(1)}\right| \geqslant 1/N^{1/3}$. So the target u cannot be any of those $N^{2/3}$-many i. The learner then makes a second query to $\mathcal{D}$, and the adversary's response will be "$\langle v^{(2)}, u\rangle \approx_\tau 0$." By Fact 24.4, there are at most $N^{2/3}$ coordinates i for which $\left|v_i^{(2)}\right| \geqslant 1/N^{1/3}$. So the target u cannot be any of those $N^{2/3}$-many i. And so on.

In fact, the learner can repeat this process $N^{1/3} - 1$ times, and still have

$$\begin{aligned} N - \left(N^{1/3} - 1\right) \cdot N^{2/3} &= N - \left(N - N^{2/3}\right) \\ &= N^{2/3} \end{aligned}$$

possible e_i left, and as long as we even have *two* e_i left that are orthogonal, the learner did not learn successfully.

Here are two concrete takeaways from this discussion:

1. For learning DNFs, we have $N = n^{\log n}$ concepts, and so if we want to achieve accuracy $\frac{1}{2} + \frac{1}{N^{1/3}}$, then we need to make $\geqslant n^{(\log n)^{1/3}}$ queries, or make some query with tolerance $\tau \leqslant \frac{1}{n^{(\log n)^{1/3}}}$. So, provably, we cannot learn DNFs with an SQ algorithm in polynomial time.
2. For learning decision trees, we have $N = 2^{O(n)}$ concepts, and so if we want to achieve accuracy $\frac{1}{2} + \frac{1}{N^{1/3}}$, then we need to make $\geqslant 2^{n^{1/3}}$ queries, or make some query with tolerance $\tau \leqslant \frac{1}{2^{n^{1/3}}}$. So, provably, we cannot learn decision trees with an SQ algorithm in polynomial time.

So we can indeed prove *unconditional hardness* for the impossibility of efficiently learning these concept classes—we don't need to assume $\mathsf{P} \neq \mathsf{NP}$ or any other complexity-theoretic conjecture; everything is entirely information-theoretic.

§24.2 Hardness of learning

In this course, we cover three types of hardness of learning:

1. **Information-theoretic/unconditional hardness**: Here we are concerned with the hardness of learning no matter how much computational resources we have, regardless of whatever assumptions we make about the complexity of problems. We have already seen several examples of this kind of hardness, for example:
    - There is no PAC learning algorithm for monotone conjunctions using $\sqrt{\frac{n}{\varepsilon}}$ examples (since the VC dimension of monotone conjunctions is n, and so we need $\geqslant \Omega\left(\frac{n}{\varepsilon}\right)$ many examples to PAC-learn them; the square root is not enough).
    - There is no poly(n)-time SQ algorithm for learning n-term DNFs (like we showed just above).
    - Define the concept class $\mathcal{C}_{\text{ALL}}$ to be the class of all 2^{2^n} functions $f\colon \{0,1\}^n \to \{0,1\}$. Then there is no poly$(n)$-time PAC learning algorithm for $\mathcal{C}_{\text{ALL}}$ (since the VC dimension of

$\mathcal{C}_{\text{ALL}}$ is 2^n, and so we need $\geqslant \Omega\left(\frac{2^n}{\varepsilon}\right)$ many examples to PAC-learn $\mathcal{C}_{\text{ALL}}$; any polynomial in n is not enough).

2. **Representation-dependent hardness of learning**: Here we are concerned with the computational hardness of learning a concept class $\mathcal{C}$ when the learner is only allowed to represent the target concept using a certain type of representation taken from a hypothesis class $\mathcal{H}$. For example, we have seen that unless graph 3-colourability is efficiently solvable (i.e. unless $\mathsf{NP} = \mathsf{RP}$), we cannot learn 3-term DNFs using the hypothesis class of 3-term DNFs. This kind of learning is vaguely unsatisfying; it insists on a certain type of representation, and so it is not clear how much it tells us about the inherent hardness of learning the concept class $\mathcal{C}$.

3. **Representation-independent hardness of learning**: This is much more fundamental than the previous two; it is hardness based on some inherent unpredicatability in the concept class. The general sort of statements in this category are of the form "unless some computational problem P has an efficient algorithm, then no efficient algorithm can PAC-learn $\mathcal{C}$ using any polynomially-evaluatable hypothesis class $\mathcal{H}$." In some sense, these concept classes are inherently hard to learn *on their own*, and so this kind of hardness is much more satisfying for us.

Now types (2) and (3) of the above make computational assumptions (we are assuming that some other problem is inherently hard to solve), but it would be much nicer if we could *unconditionally* prove that certain concept classes are inherently hard to learn. But unless we prove that $\mathsf{P} \neq \mathsf{NP}$, we will not get these results, and so we don't know how to proceed any further.

Example 24.5. If we could show, unconditionally, that

$$\mathcal{C} = \{\text{all } n^2\text{-term DNFs over } \{0,1\}^n\}$$

isn't PAC-learnable in polynomial time, then it would follow immediately that $\mathsf{P} \neq \mathsf{NP}$. Indeed, if $\mathsf{P} = \mathsf{NP}$, then it is possible to verify, in $\text{poly}(n, m)$ time, that given

- a data set S of m labelled examples from $\{0,1\}^n$, and
- $g = T_1 \vee \cdots \vee T_{n^2}$, an n-term DNF,

that g is consistent with S. Then, given the data set S of m labelled examples from $\{0,1\}^n$, we can use the verifier to find the n^2-term DNF g (a consistent n^2-term DNF for a data set labelled by an n^2-term DNF) that is consistent with S, in $\text{poly}(n, m)$ time. Then if $\mathsf{P} = \mathsf{NP}$, we always have an efficient consistent hypothesis finder for $\mathcal{C}$ using $\mathcal{C}$, and hence we can always PAC-learn $\mathcal{C}$ efficiently, and the whole course is a waste of time (it's not ☺).

The above example suggests that we need (complexity-theoretic) assumptions to prove hardness of learning in cases (2) and (3). We have already seen some worst-case assumptions, where we assume that no efficient algorithm exists for some problem P (e.g. graph 3-colourability):

> "There is no $\text{poly}(n)$-time algorithm which, for *every* n-node graph, correctly determines whether it is 3-colourable or not."

But for representation-independent hardness, we can make stronger "hard-on-average" or *average-*

case hardness assumptions, where the basic setup is as follows: we have a distribution $\mathcal{D}$ on problem instances, and we're saying that no efficient algorithm can succeed on a non-negligible fraction of instances drawn from $\mathcal{D}$:

> "For a suitable distribution $\mathcal{D}$, no poly(n)-time algorithm can succeed on 1% of the instances drawn from $\mathcal{D}$, for whatever the hard problem is."

Indeed it turns out that all known results for the hardness of learning are based on average-case hardness assumptions. We will see some of these results based on pseudorandomness and (public-key) cryptography in the next lecture.

§25 Lecture 25—4th December, 2023

Last time. Said more about SQ-learning and started talking about the hardness of learning concept classes.

- Sketch a proof that if a concept class $\mathcal{C}$ and a distribution $\mathcal{D}$ are such that $\mathcal{C}$ contains many (about $n^{\omega(1)}$ many) functions that are all pairwise uncorrelated under $\mathcal{D}$, then there is no efficient SQ learning algorithm for $\mathcal{C}$ under $\mathcal{D}$.
  - problem five on the homework set: there is no efficient SQ learning algorithm for the class of all n-variable parity functions, DNFs, and decision trees under the uniform distribution.
- Start a unit on the cryptographic hardness of learning "rich" concept classes.

Today.

- Discussion on the computational hardness of learning:
  $$\mathcal{C} = \{\text{all poly}(n)\text{-size Boolean circuits}\}$$
  based on the existence of pseudorandom function families (note that we have hope to learn $\mathcal{C}$, since $|\mathcal{C}| = 2^{\text{poly}(n)}$ and by the Occam's razor principle, we can learn with $\frac{1}{\varepsilon}(\ln|\mathcal{H}| + \ln\frac{1}{\delta})$ examples, which is polynomial in n).
- Mapping the boundary of efficient learnability.
- Hopefully start the hardness of learning based on public-key cryptography (trapdoor 1-way permutations).

We discussed last time that in order to prove that concept classes are hard to learn, we need conditional assumptions of the form "if there is no efficient learning algorithm for $\mathcal{C}$ under $\mathcal{D}$, then some complexity-theoretic assumption holds" or "if there is no efficient learning algorithm for $\mathcal{C}$ under $\mathcal{D}$, then some algorithm has no efficient solution," because if we can prove unconditionally that a concept class is hard to learn, then we have proved that $\mathsf{P} = \mathsf{NP}$ because then we would have an efficient consistent hypothesis finder for any such concept class. Notice that average-case hardness assumptions, i.e. those of the form

> "For a suitable distribution $\mathcal{D}$, no poly(n)-time algorithm can succeed on 1% of the instances drawn from $\mathcal{D}$, for whatever the hard problem is."

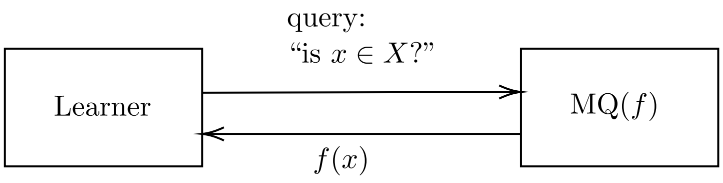
are stronger than worst-case hardness assumptions, i.e. those of the form

> "There is no poly(n)-time algorithm which, for *every* n-node graph, correctly determines whether it is 3-colourable or not."

For instance, the worst-case hardness assumption that there is no poly(n)-time algorithm for 3-colourability essentially says that there is no worst-case poly(n)-time algorithm for 3-colourability on *any* graph, but in fact there is a very easy algorithm[3] for graph 3-colourability which works for $\gg 1 - \frac{1}{n^{100}}$, and so there's a sense in which graph 3-colourability is easy in the average-case, but hard in the worst case. So to get hardness of learning, we will make average-case (strong) hardness assumptions to get hardness of learning.

§25.1 Hardness of learning based on pseudorandomness

We need to introduce a new learning model, that of *membership queries* (MQs). In the PAC model, the learner gets to see examples, but in the MQ model, the learner gets to ask the teacher whether a particular point is in the concept class or not—it has some "black-box oracle access." Unlike the statistical query model where the learner had less power (it could only ask about the data distribution of the target concept), in the membership query model the learner has more power (it can ask about the target concept itself).



If A is an algorithm (a learner), then we write A^f to mean "A with MQ oracle access to f."

Now let us introduce the notions of a (truly) *random* function, versus a *pseudorandom* function. A truly random function (over $X = \{0,1\}^n$) is a function picked uniformly at random from the set of all functions $f : X \to \{0,1\}$. That is, if f is a random Boolean function, then $f \sim_{\text{unif}} \mathcal{C}_{\text{ALL}}$, where $\mathcal{C}_{\text{ALL}}$ is the concept class of all 2^{2^n} Boolean functions $f : X \to \{0,1\}$. Note that, in fact, to pick such an f, we need to toss 2^n coins to pick such an f (we'd have to write down all 2^n entries of the truth table of f, and for each one, we need a coin flip to decide whether it is 0 or 1). The idea behind pseudorandomness is that we want to be able to simulate a truly random function using a much shorter description. A pseudorandom function is a function that "looks random" to any efficient algorithm that is given black-box access to it. We are thinking here of randomness as a "valuable resource for computation," and we don't want to spend too much of it lest we run out; instead, we will use a much shorter (but almost-as-equally-efficient) description (called a *seed*) to simulate a truly random function. More formally, a function $f : X \to \{0,1\}$ is pseudorandom if it is uniformly drawn from a pseudorandom function family, defined below:

[3]The algorithm just says "no" everytime—almost every graph is not 3-colourable. In particular, if we build an Erdös-Rényi random graph $G(n, 1/2)$, then with high probability, the graph is not 3-colourable, as—and this is an easy probability problem—we can show that there are not-too-many complete graphs K_4, i.e. the chances that all six edges are present in K_4 is $\frac{1}{2^6}$, and the probability that it is impossible to legally colour the graph with three colours is at least this. The probability that a graph is 3-colourable is then at most $\left(1 - \frac{1}{2^6}\right)^{\frac{n}{4}} = \frac{1}{2^{\Theta(n)}}$, and so the probability that a graph is not 3-colourable is at least $1 - \frac{1}{2^{\Theta(n)}}$, which is large especially for large n.

Definition 25.1 (Pseudorandom function family (PRFF)). *Let $\mathcal{F}$ be a set of 2^n Boolean functions, i.e.*

$$\mathcal{F} := \{f_s : s \in \{0,1\}^n\} \text{ for each } f_s \colon \{0,1\}^n \to \{0,1\},$$

where s is called the seed of f_s. We say that $\mathcal{F}$ is a pseudorandom function family *(abbreviated* PRFF*) if:*

1. *(Efficient computability.) Each f_s is computable by polynomial-size Boolean circuit families $\{C_n\}_{n\in\mathbb{N}}$; in fact, there is a* $\mathrm{poly}(n)$*-time algorithm that, given s and x, computes $f_s(x)$.*
2. *(Indistinguishability to* $\mathrm{poly}(n)$*-time observers from truly random functions.) Let* DIST*, a distinguisher, be any* $\mathrm{poly}(n)$*-time algorithm which gets oracle access to a Boolean function $f : \{0,1\}^n \to \{0,1\}$ and outputs either* `pseudorandom` *or* `random`*. Then for $f_s \in \mathcal{F}$, we have*

   $$\left| \Pr_{f\sim\mathcal{C}_{\text{ALL}}}\left[\text{DIST}^f \textit{ outputs } \texttt{pseudorandom}\right] - \Pr_{s\sim\{0,1\}^n}\left[\text{DIST}^{f_s} \textit{ outputs } \texttt{pseudorandom}\right]\right| < \frac{1}{p(n)}$$

   for all polynomials $p(n)$. (Here the sampling $f \sim \mathcal{C}_{\text{ALL}}$ means that f is a random function uniformly drawn from the set of all 2^{2^n} Boolean functions and $s \sim \{0,1\}^n$ means that s is a random seed uniformly drawn from $\{0,1\}^n$ for a pseudorandom function f_s.)

A pseudorandom Boolean function is a function uniformly drawn from a pseudorandom function family (note that the number of coin tosses needed to pick such an f is much less than 2^n; in fact it is only n bits).

Shafi Goldwasser and Silvio Micali won the Turing Award in 2012 for their work on pseudorandomness, this ingenious and powerful notion of randomness as being relative to the perspective of an observer. Indeed, a major cryptographic hardness assumption is the existence of pseudorandom function families. In particular, two cornerstones of modern cryptography is the fact that if one-way functions exist, then pseudorandom function families exist, and that if factoring is average-case hard, then pseudorandom function families exist.

Fortunately (although unfortunate for learners!), a pseudorandom function family is a hard-to-learn concept class:

Theorem 25.1. *Suppose $\mathcal{F}$ is a pseudorandom function family.*

Then there is no $\mathrm{poly}(n)$*-time PAC learning algorithm A for $\mathcal{F}$, using any polynomially-evaluatable hypothesis $\mathcal{H}$, even if:*

(i) we only require the algorithm to succeed under the uniform distribution on $\{0,1\}^n$;

(ii) the learner has membership query access to the target function f_s.

The idea for the proof of this theorem is very simple: we already know that we cannot learn truly random functions $f \sim \mathcal{C}_{\text{ALL}}$ because the class $\mathcal{C}_{\text{ALL}}$ has VC dimension 2^n, so the label of every point we haven't seen is merely a coin toss. On the other hand, if we have a learning algorithm for a pseudorandom function family, then we can use it to distinguish between truly random functions and pseudorandom functions, which is impossible by the definition of a pseudorandom function family.

Proof of Theorem 25.1. Let A be an efficient PAC learning algorithm for $\mathcal{F}$ as in Theorem 25.1. We will use A to get a distinguisher as follows: given oracle access to some unknown c(perhaps $c = f$ with $f \sim \mathcal{C}_{\text{ALL}}$, or $c = f_s$ for some $s \sim \{0,1\}^n$):

- run A using the membership query oracle with $\varepsilon = 0.01$ and $\delta = 0.01$ to get a hypothesis $h\colon \{0,1\}^n \to \{0,1\}$ that is poly(n)-time evaluatable;
- draw a uniform $z \sim \{0,1\}^n$, call MQ(c) on z to get $c(z)$, and evaluate $h(z)$;
- if $c(z) = h(z)$, output `pseudorandom`; otherwise, output `random`.

But this our purported distinguisher should not exist!

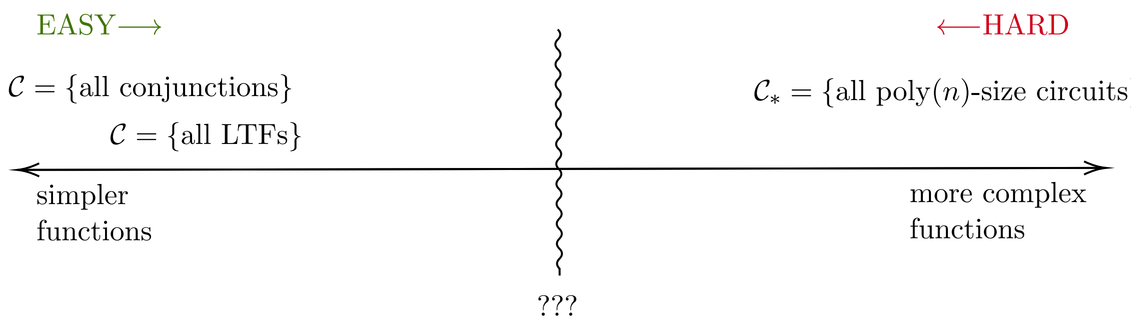
Claim. This distinguisher violates Property 2 of Definition 25.1.

This claim is true by the following argument.

- Suppose $c = f_s$ for some $f_s \in \mathcal{F}$. Then with probability $\geqslant 0.99$, the hypothesis h output by A has error $\leqslant 0.01$, and hence overall with probability $\geqslant 0.98$, $h(z) = f_s(z)$ and so the distinguisher outputs `pseudorandom`.
- Suppose $c = f$ for some $f \sim \mathcal{C}_{\text{ALL}}$, i.e. f is a random function. Then unless z is in the support of the examples seen by A (with probability $\leqslant 2^{-n} \cdot \text{poly(n)} \ll 0.01$), the decision $h(z) = c(z)$ is just a coin toss with probability $\frac{1}{2}$, and so the overall probability that $h(z) = c(z)$ is $\leqslant \frac{1}{2} + 2^{-n} \cdot \text{poly(n)} \leqslant 0.51$, and so the distinguisher outputs `random`.

So we have an efficient distinguisher, which is a contradiction. So we couldn't have had an efficient PAC learning algorithm A for $\mathcal{F}$ in the first place. □

So Theorem 25.1 tells us that there *are* computational (as opposed to information-theoretic) barriers to efficient learning. The big question in learning theory then is: what are the boundaries of efficient learnability?



EASY⟶
⟵HARD
$\mathcal{C} = \{\text{all conjunctions}\}$
$\mathcal{C} = \{\text{all LTFs}\}$
$\mathcal{C}_* = \{\text{all poly}(n)\text{-size circuits}\}$
simpler functions
more complex functions
???

(Here we're using the circuit complexity to measure the "simplicity" of a concept class, and we're using the uniform distribution on $\{0,1\}^n$ as the "hard" distribution under which we're trying to learn.)

One way by which we can map this boundary is to try to show that there are "simpler" concept classes that are still hard to learn compared to $C_* = \{\text{all poly}(n)\text{-size circuits}\}$. We know now that a circuit class that can compute pseudorandom functions is hard to learn, so we can adjust our goal to that of designing simpler and simpler pseudorandom function families. There has been some progress in this area; for instance, under some plausible cryptographic assumptions, we can show that there are pseudorandom function families that are computable as depth-5 majority circuits. So if we can come up with a learning algorithm for the class of all depth-5 majority circuits, then we can break some cryptographic assumptions. This is still a very active area of research.

Another way to try to map out this boundary is to explore cryptographic reasons for why hardness of learning is inherent in certain concept classes. Indeed, there's some juice to be squeezed here.

§25.2 Hardness of learning based on trapdoor one-way permutations

We will now discuss a (public-key) cryptographic hardness assumption that is relevant for our hardness of learning discussion.

Definition 25.2. *A* permutation *of a finite set X is a bijection $f\colon X \to X$.*

Informally, a permutation is a "scrambling" of the elements of a set. Perhaps less informally, a "one-way permutation" on $X = \{0,1\}^n$ is a permutation $f\colon \{0,1\}^n \to \{0,1\}^n$ such that:

- there is a poly(n)-time algorithm to compute f, but
- any poly(n)-time algorithm cannot compute f correctly even on a $1/\text{poly}(n)$ fraction of the inputs.

Next time we will see more about how one-way permutations connect to the hardness of learning.

§26 Lecture 26—6th December, 2023

Last time.

- Discussion on the computational hardness of learning:

$$\mathcal{C} = \{\text{all poly}(n)\text{-size Boolean circuits}\}$$

  based on the existence of pseudorandom function families (note that we have hope to learn $\mathcal{C}$, since $|\mathcal{C}| = 2^{\text{poly}(n)}$ and by the Occam's razor principle, we can learn with $\frac{1}{\varepsilon}(\ln|\mathcal{H}| + \ln\frac{1}{\delta})$ examples, which is polynomial in n).
- Mapping the boundary of efficient learnability.
- Hopefully start the hardness of learning based on public-key cryptography (trapdoor 1-way permutations).

Today.

- More hardness of learning based on public-key cryptography (trapdoor 1-way permutations).
  - Our hardness assumption: discrete cube roots are hard to compute.
- Using this to show that even "simple" poly(n)-size Boolean circuits (equivalently, $O(\log n)$-depth Boolean circuits) are hard to learn.

Reminder: the final is next Friday, 15th December, 2023, 9:00am-10:30am. It will be closed-book and closed-notes, no calculators or electronic devices, cumulative, and will be held in-person in Uris 141.

---

§26.1 Cryptographic hardness of learning based on trapdoor one-way permutations

Recall the definitions we introduced last time:

Definition 26.1. *A* permutation *of a finite set X is a bijection $f\colon X \to X$.*

Informally, a permutation is a "scrambling" of the elements of a set. Perhaps less informally, a "one-way permutation" on $X = \{0,1\}^n$ is a permutation $f\colon \{0,1\}^n \to \{0,1\}^n$ such that:

- there is a poly(n)-time algorithm to compute f, but
- any poly(n)-time algorithm cannot compute f correctly even on a 1/poly(n) fraction of the inputs.

Then a trapdoor permutation is simply a one-way permutation with a "trapdoor" (a hidden secret) that allows one to invert the permutation efficiently. More formally:

Definition 26.2 (Trapdoor one-way permutations)**.** *A tuple of polynomial-time algorithms, denoted as* $(\mathsf{Gen}, \mathsf{Samp}, f, \mathsf{Inv})$ *is a* trapdoor permutation *if:*

- *The probabilistic parameter generation algorithm* Gen*, on input* 1^n*, outputs* (I, td) *with* $|I| \geqslant n$*. Each value of i defines a set S_I that constitutes the domain and range of a permutation $f_I\colon S_I \to S_I$.*
- *Let* Gen_1 *denote the algorithm that results by running* Gen *and outputting only I. Then* $(\mathsf{Gen}_1, \mathsf{Samp}, f, \mathsf{Inv})$ *is a family of one-way permutations.*
- *Let* (I, td) *be an output of* $\mathsf{Gen}(1^n)$*. The deterministic inverting algorithm* Inv*, on input* td *and $y \in S_I$, outputs $x \in S_I$. We denote this by* $x \coloneqq \mathsf{Inv}_{\mathsf{td}}(y)$*. It is required that with all but negligible probability over the choice of I and* td *output by* $\mathsf{Gen}(1^n)$*, for all $y \in S_I$, we have*

$$\mathsf{Inv}_{\mathsf{td}}(f_I(y)) = y.$$

> **Example 26.3.** An example of trapdoor information is the prime factorisation of a number $N = p \cdot q$. If we know p and q, we can easily compute N, but if we only know N, it is hard to compute p and q (in fact there is no known polynomial-time algorithm to do this).

Public-key cryptography Public-key cryptography is a scheme that allows two parties to communicate based on the existence of trapdoor permutations. The problem is as follows: we want to find a way for Bob to send a message securely to Alice in the presence of Eve (an eavesdropper) who has access to the communication channel (and ostensibly can read the message). The setup is as follows:

- Alice creates a trapdoor one-way permutation (e.g. picks primes p and q) f—which we can think of as an encryption function—and she knows how to compute f^{-1} (the decryption function) using the trapdoor information.

- Alice then publishes the algorithm for computing f in the forward direction (e.g. publishes $N = p \cdot q$), and keeps the trapdoor information secret.

Then to communicate:

- To send a message y to Alice, Bob computes $f(y)$ and sends it to Alice.
- Alice applies f^{-1} to $f(y)$—decrypts the message—and reads y.
- Eve, who only knows f and the algorithm to compute f, cannot decrypt the message without the trapdoor information, and so cannot read the message.

The connection to learning for us is as follows. Assuming that trapdoor one-way permutations exist, the decryption function must be hard-to-learn (under the uniform distribution on domain $X = \{0,1\}^n$). Here's why:

1. In the *eavesdropping world*, Eve sees $f(y)$ and wants to compute $f^{-1}(f(y)) = y$. (This is hard for her because she doesn't have the trapdoor information.)
2. In the *learning world*, some supposed learner A wants to learn f^{-1} from labelled examples $(x, f^{-1}(x))$, where x is drawn from the uniform distribution on $X = \{0,1\}^n$. A comes up with a hypothesis h of high-accuracy for f^{-1}.

So the only difference between both worlds is that in the learning world, the learner has access to labelled examples. But even in the eavesdropping world, Eve can simulate these labelled examples $(x, f^{-1}(x))$ for herself! In particular, she can pick z uniformly at random from X, and in that case, since z is uniform (and consequently $f(z)$ is uniform, since f is a permutation),

$$(f(z), z) \text{ is distributed exactly like } (x, f^{-1}(x)).$$

So if a poly(n)-time learning algorithm A existed, Eve could use it to get a high-accuracy hypothesis for f^{-1}, and eavesdrop on Alice and Bob's communication, thereby breaking the security of the public-key cryptography scheme. This is impossible, so we conclude that f^{-1} is hard to learn.

§26.2 A hard-to-learn concept class $\mathcal{C}$ based on discrete cube root hardness

We now consider a concrete instantiation of the above idea. We will show that a specific concept class $\mathcal{C}$ is hard to learn, assuming that discrete cube roots are hard to compute. First we will need quite a bit of setup from number theory.

Let $N = p \cdot q$, where p and q are large ($\frac{n}{2}$-bit) primes, both congruent to $2 \mod 3$.

Definition 26.4 (Ring of integers modulo N)**.** *The ring of integers modulo N is defined as*

$$\mathbb{Z}_N = \{0, 1, \dots, N-1\}.$$

Definition 26.5 (Group of units modulo N)**.** *The group of units modulo N is defined as the set of all $j \in \mathbb{Z}_N$ such that j is relatively prime to N:*

$$\mathbb{Z}_N^* = \{j \in \mathbb{Z}_N : \gcd(j, pq) = 1\}.$$

Definition 26.6 (Euler's totient function). *Euler's totient function $\varphi(N)$ is defined as the number of elements in $\mathbb{Z}_N^*$:*

$$\varphi(N) = |\mathbb{Z}_N^*| = (p-1)(q-1).$$

The following facts are encountered in an introductory number theory course:

Fact 26.7 ($\mathbb{Z}_N^*$ is a group under $\times$ mod N). *$\mathbb{Z}_N^*$ is a group under multiplication modulo N; that is:*

- *if $a, b \in \mathbb{Z}_N^*$, then $a \cdot b \in \mathbb{Z}_N^*$,*
- *if $a \in \mathbb{Z}_N^*$, then there is a unique element $a^{-1} \in \mathbb{Z}_N^*$ such that $a \cdot a^{-1} = 1 \mod N$.*

Fact 26.8 (Corollary of Lagrange's theorem). *For any finite group G and any $g \in G$, $g^{|G|} = 1$.*

So for any $a \in \mathbb{Z}_N^*$, since $|\mathbb{Z}_N^*| = \varphi(N) = (p-1)(q-1)$, we have $a^{\varphi(N)} = 1 \mod N$ and in particular, $a^{\varphi(N)-1} = a^{-1} \mod N$.

Claim 26.9. *For $N = p \cdot q$, where p and q are large primes, and $p, q \equiv 2 \mod 3$, we have that $2\varphi(N) + 1 \equiv 0 \mod 3$.*

Proof. Since $p \equiv 2 \mod 3$, we have $p - 1 \equiv 1 \mod 3$, and similarly $q - 1 \equiv 1 \mod 3$. So

$$\varphi(N) = (p-1)(q-1) \equiv 1 \cdot 1 \equiv 1 \mod 3.$$

Then $2\varphi(N) + 1 \equiv 2 \cdot 1 + 1 \equiv 0 \mod 3$. □

The following claim is less boring:

Claim 26.10. *For $N = p \cdot q$ as above, the function*

$$f_N(x) = x^3 \mod N$$

is a permutation of $\mathbb{Z}_N$.

Proof. The inverse f_N^{-1} is $f_N^{-1}(y) = y^d \mod N$. Then we can check that for any $x \in \mathbb{Z}_N$,

$$\begin{aligned}
f_N(x)^d \mod N &= \left(x^3 \mod N\right)^d \mod N \\
&= \left(x^{3d} \mod N\right) \mod N = x^{3d} \mod N = x^{2\varphi(N)+1} \mod N \\
&= \left(x^{2\varphi(N)} \mod N\right) \cdot (x \mod N) \\
&= \left(\left(x^2 \mod N\right)^{\varphi(N)} \mod N\right) \cdot (x \mod N) \\
&= (1 \mod N) \cdot (x \mod N) = x \mod N.
\end{aligned}$$

So y^d is indeed $f_N^{-1}(y)$, and so $f_N(x) = x^3$ is a permutation of $\mathbb{Z}_N^*$ (it maps $\mathbb{Z}_N^*$ to $\mathbb{Z}_N^*$ and has a well-defined unique inverse). □

So f_N is a suitable permutation of $\mathbb{Z}_N^*$; p, q are the trapdoor information since given p, q, it is easy to compute

$$d = \frac{2(p-1)(q-1)+1}{3},$$

and then $f_N^{-1}(y) = y^d \mod N$. We however need an extra hardness assumption to get one-way-ness. We present it below.

Problem: Discrete Cube Root Problem [RSA78]

Input. Two numbers N and y, where:

- $N = p \cdot q$ for large primes $p, q \equiv 2 \mod 3$,
- $y \in \mathbb{Z}_N^*$ (so $y \equiv x^3 \mod N$ for some unique $x \in \mathbb{Z}_N^*$).

Output. The inverse $f_N^{-1}(y)$, i.e. the unique $x \in \mathbb{Z}_N^*$ such that $x^3 \equiv y \mod N$.

Proposition 26.11 (Discrete cube root hardness assumption (DCRHA)). *For any polynomial $p(n)$, there is no $p(n)$-time (classical) algorithm A such that, if A is given N and y, where*

- *$N = p \cdot q$ for large uniformly random $\frac{n}{2}$-bit primes $p, q \equiv 2 \mod 3$,*
- *y a uniformly random element of $\mathbb{Z}_N^*$,*

then A outputs x with probability $\geqslant 1/p(n)$, where $x^3 \equiv y \mod N$.

Note that if we could factor $N = p \cdot q$ quickly, then this problem is easy, since then we could compute x by computing $x = y^d \mod N$, where

$$d = \frac{2(p-1)(q-1)+1}{3}.$$

Let's now tie all of this back to the Boolean function hardness of learning. Define the concept class $\mathcal{C} = \{\text{all functions } f_N^{-1}\}$, where $N = p \cdot q$ for large primes $p, q \equiv 2 \mod 3$ and $f_N^{-1} \colon \{0,1\}^n \to \{0,1\}^n$. For each n, let $\mathcal{D}_N$ be the uniform distribution over $\mathbb{Z}_N^*$.

Now suppose that there is a poly-time algorithm A which PAC-learns $\mathcal{C}$. If so, then for any $c \in \mathcal{C}$ and for any distribution $\mathcal{D}$ over $\{0,1\}^n$, this algorithm works (gives ε-accurate hypothesis with $\geqslant 1-\delta$ probability), and in particular it succeeds if the target function is f_N^{-1} and the distribution is $\mathcal{D}_N$. So the algorithm A, given N and access to random examples $(x, f_N^{-1}(x))$ where x is drawn from $\mathcal{D}_N$, outputs a hypothesis h such that $\Pr_{x\sim\mathcal{D}_N}[h(x) \neq f_N^{-1}(x)] \leqslant \varepsilon$ with probability $\geqslant 1-\delta$.

But then such an algorithm A would contradict DCRHA! Given N and y (as inputs to the discrete cube roots problem), we can run A using $\mathcal{D}_N$ as the distribution as follows: draw a uniformly random $z \sim \mathcal{D}_N^*$, compute $f_N(z) = z^3 \mod N$, and use $(f_N(z), z)$ as our desired $(x, f_N^{-1}(x))$ example. Then A outputs a hypothesis h such that $h(y) = f_N^{-1}(y)$ with overall probability $\geqslant 1-\delta-\varepsilon$. But this contradicts DCRHA, so we conclude that A cannot exist.

One last point of note is that we can "Booleanise" the problem of learning f_N^{-1} by considering a bonafide concept class $\widehat{\mathcal{C}}$ by looking at the ith bit of f_N^{-1} for each $i \in [n]$. We will see why this representation is useful next time.

§27 Lecture 27—11th December, 2023

Last time.

- More hardness of learning based on public-key cryptography (trapdoor 1-way permutations).
  - Our hardness assumption: discrete cube roots are hard to compute.

Today.

- Using this to show that even "simple" poly(n)-size Boolean circuits (equivalently, $O(\log n)$-depth Boolean circuits) are hard to learn.
- Peek at some other topics in COMS W4252 that we didn't get to cover this semester.

Reminder: today is the last lecture; the final is next Friday, 15th December, 2023, 9:00am-10:30am. It will be closed-book and closed-notes, no calculators or electronic devices, cumulative, and will be held in-person in Uris 141, and not in Pupin 428. The final will cover all material from the entire semester, with a focus on material since the midterm.

§27.1 Cryptographic hardness of learning simple circuits based on discrete cube roots

Last time, we ended on an optimistic note: we have at least some hope for learning if we can Booleanise our hard-to-learn concept class $\mathcal{C}$ of functions $f_N^{-1} = x^d \mod N$ of functions by considering the ith bit of $f_N^{-1}(x)$; that is, we construct the bonafide concept class $\widehat{\mathcal{C}}$ by looking at the ith bit of f_N^{-1} for each $i \in [n]$.

Claim 27.1. *If DCRHA (Proposition 26.11) is true, then $\widehat{\mathcal{C}}$ is not efficiently learnable in the PAC model.*

Proof. Suppose we have an efficient PAC learning algorithm for $\widehat{\mathcal{C}}$, and let A be the algorithm. Then given N, y, we can generate random examples for $f_{N,1}^{-1}, f_{N,2}^{-1}, \ldots, f_{N,n}^{-1}$ as follows:

- Pick a random $z \in \mathbb{Z}_N^*$.
- Compute $f_N(z)$.
- Use $(f_N(z), z)$ as the examples corresponding to $(x, f_N^{-1}(x)_i)$.

So we are indeed getting the labelled examples we crave for $\widehat{\mathcal{C}}$. So if we have our algorithm A, we can run it n times for $f_{N,1}^{-1}, f_{N,2}^{-1}, \ldots, f_{N,n}^{-1}$, each time with accuracy $1/n^2$ and confidence $1/n^2$. So we have a $1 - \frac{2}{n^2}$ chance that the hypothesis h_i is right on y for each $i \in [n]$. So we can take a union bound over all n hypotheses, and we get that the probability that all n hypotheses are wrong is at most $2/n$. So we can run A exactly n times, and with probability at least $1 - 2/n$, we will get a hypothesis that is correct on all n bits of $f_N^{-1}(x)$. So we can use this to compute $f_N^{-1}(x)$, and we can use this to compute x, in which case Proposition 26.11 is false—a contradiction. □

We now want to use this result to say something about the hardness of a relatively simple class of Boolean circuits, so we first make some statements introducing circuit complexity.

Definition 27.2. *A* Boolean circuit *is a directed acyclic graph with the following properties:*

- *Each node is either an input node, an output node, or a gate node.*
- *Each input node is labelled with a variable x_i.*
- *Each gate node is labelled with a Boolean function $f : \{0,1\}^k \to \{0,1\}$ for some $k \geqslant 1$.*
- *The singular output node is labelled with a variable y (the output of the circuit).*

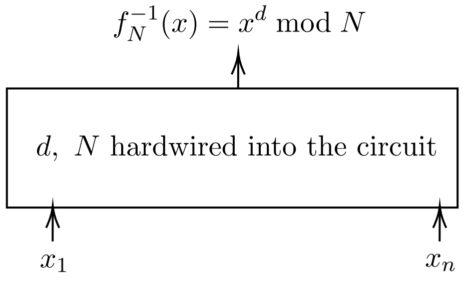
The size *of a circuit is the number of nodes in the circuit. The* depth *of a circuit is the length of the longest directed path from an input node to an output node.*

Circuits are a natural model for computation for Boolean functions, and we can think of them as a generalisation of Boolean formulas (except that we can reuse subcomputations in circuits, but not in formulas).

Our interest then shifts to bounding the circuit complexity of these functions we know are hard to learn. We already know that the functions

$$f_N^{-1}(x) = x^d \mod N = x^{\frac{2(p-1)(q-1)+1}{3}} \mod N = x^d \mod N$$

are hard to learn, and we are interested in what this tells us about the hardness of learning different types of circuits.


$f_N^{-1}(x) = x^d \bmod N$
$d,\ N$ hardwired into the circuit
x_1
x_n

How can we hope to compute $f_N^{-1}(x)$? Recall that N and d are both n-bit numbers, so we can do better by repeated multiplication: bit-shifting by repeated squaring on d in its binary representation, that is, repeat

$$O(n) \text{ many times } \begin{cases} x \mod N \\ x^2 \mod N \\ x^4 \mod N \\ \vdots \\ x^{2^n} \mod N \end{cases}$$

corresponding to the i's in the binary representation of d. So we can compute $f_N^{-1}(x) = x^d \mod N$ in $O(n)$ time by multiplying all of the $x^{2^i} \mod N$'s corresponding to the 1's in the binary representation of d.

With this reasoning, it is not too hard to start understanding the circuit complexity of $f_N^{-1}(x)$. The following easy fact is a first step:

Fact 27.3. *We can multiply two n-bit numbers with a* $\text{poly}(n)$*-size,* $O(\log n)$*-depth circuit.*

The proof is trivial, so we will not do it in lecture. What this does tell us though is that there is a circuit that computes $f_N^{-1}(x)$ that has size $\text{poly}(n)$ and depth $O(n \cdot \log n)$.

But we can do much better than this with a dirty trick[4]! Instead of taking $f_N^{-1}(x) = x^d \mod N$ as the hard function, consider a setup where the input is already the binary representation we used:

$$\underbrace{x, x^2 \mod N, x^4 \mod N, \ldots, x^{2^n} \mod N}_{\text{an } n^2\text{-bit input}}.$$

Consider also the distribution $\mathcal{D}_N^*$ defined as follows: a draw from $\mathcal{D}_N^*$ is:

- pick a random x uniformly from $\mathbb{Z}_N^*$ (which is the same as the old $\mathcal{D}_N$)
- output the n^2-bit string $(x, x^2 \mod N, x^4 \mod N, \ldots, x^{2^{\lfloor \log d \rfloor}} \mod N)$.

Then the function we're learning is $f_N^{-1}(x) = x^{2^b_1} \cdot x^{2^b_2} \cdots x^{2^b_n} \mod N$, where b_i are the locations of the i-th 1's in the binary representation of d. Essentially, we have given a feature-expanded version of the input, and we are learning the function $f_N^{-1}(x)$ on this feature-expanded input, which is the same as learning $f_N^{-1}(x)$ on the original input, except more efficient. So this also has to be a hard learning problem—if this was an easy learning problem, then we would have had an efficient algorithm for the original learning problem (with just x as the input), which we already showed is impossible, so the following algorithm must be hard too:

- On input d, N, let the input to the function be $z = (z_1, z_2, \ldots, z_n)$ where $z_i = x^{2^i} \mod N$ is an n-bit string.
- Output the jth bit of $z_{i_1} \cdot z_{i_2} \cdots z_{i_k} \mod N$, where $i_1, i_2, \ldots, i_k$ are the locations of the 1's in the binary representation of d.

So this algorithm suggests that iterated products modulo n are hard to learn. Now we need only $(\log k)$-depth tree of multiplication gates to compute the product of k numbers (where $k \leqslant n$) and by the fact we stated earlier, each individual multiplication can be done with a $\text{poly}(n)$-size, $O(\log n)$-depth circuit. So putting things together, we have that

$$\mathcal{C} = \{\text{the class of all poly}(n)\text{-size, } O(\log^2 n)\text{-depth circuits}\}$$

is hard to learn, under DCRHA (Proposition 26.11). In fact, something even more awful is true given the following fact:

Fact 27.4. *There is a circuit of depth $O(\log n)$ and size* $\text{poly}(n)$ *computing the product of n many n-bit numbers.*

This is a non-trivial fact, and we will not prove it in lecture. But the upshot is that we can use this to show that

$$\mathcal{C} = \{\text{the class of all poly}(n)\text{-size, } O(\log n)\text{-depth circuits}\}$$

is hard to learn, under DCRHA (Proposition 26.11). So we have shown that even "simple" $\text{poly}(n)$-size Boolean circuits (equivalently, $O(\log n)$-depth Boolean circuits) are hard to learn. The following fact is a corollary of this result:

[4]This trick is due to [BCH86].

Fact 27.5. *Every $O(\log n)$-depth circuit is computed by a Boolean formula of size* $\text{poly}(n)$ *and depth $O(\log n)$.*

§27.2 Beyond COMS W4252

What a semester. We have covered a lot of ground, but there is still so much more to learn. We only worked on:

1. online mistake-bounded learning,
2. PAC learning,
3. techniques for PAC learning: OLMB to PAC conversion, PAC learning via a consistent hypothesis finder, and Occam's razor,
4. sample complexity of PAC learning and the VC dimension,
5. boosting the accuracy of weak learners,
6. boosting the confidence of weak learners and the AdaBoost algorithm,
7. learning in the presence of {malicious, random classification} noise and the statistical query model,
8. representation-independent hardness of learning.

Here are some of the things we didn't get to cover but that learning theory research community is interested in:

1. *Learning rich or expressive concept classes.* How can we PAC-learn decision trees? How about DNF formulas? Constant-depth circuits? A lot of the work in this area is very complexity-theoretic, and we often call on the power of complexity theory to show that certain concept classes are hard to learn or devise efficient learners in the case that they are learnable.
2. *Membership query learning.* We talked about this in a very limited context, but there is a lot more to say about this model of learning. In particular, there are some very interesting results about learning in the presence of malicious noise in the membership query learning model, and in fact we can learn decision trees efficiently in this model.
3. *Agnostic learning.* In this setting, we are given a distribution $\mathcal{D}$ and a concept class $\mathcal{C}$, and we want to find a hypothesis that is close to the best hypothesis in $\mathcal{C}$ with respect to the error of $\mathcal{D}$. This is a much more general setting than PAC learning, and it is much more challenging. There are some very interesting results about agnostic learning for decision trees and DNF formulas, but again, we did not cover them in this course.
4. *Active learning.* In this setting, the learner is allowed to ask the teacher for the label of a point x that the learner chooses. The goal is to learn the concept with as few label queries as possible, because presumably, label queries are expensive.
5. *Distribution learning.* In this setting, the learner is given samples from an unknown distribution $\mathcal{D}$, and the goal is to learn $\mathcal{D}$ as well as possible—there is no target concept class $\mathcal{C}$ to learn.

More generally, there is a huge effort underway to bridge theory and practice in (computational) learning.

References

[AL88] Dana Angluin and Philip Laird. Learning from noisy examples. *Machine learning*, 2:343–370, 1988.

[BCH86] Paul W Beame, Stephen A Cook, and H James Hoover. Log depth circuits for division and related problems. *SIAM Journal on Computing*, 15(4):994–1003, 1986.

[BEHW87] Anselm Blumer, Andrzej Ehrenfeucht, David Haussler, and Manfred K Warmuth. Occam's razor. *Information processing letters*, 24(6):377–380, 1987.

[BEHW89] Anselm Blumer, Andrzej Ehrenfeucht, David Haussler, and Manfred K Warmuth. Learnability and the vapnik-chervonenkis dimension. *Journal of the ACM (JACM)*, 36(4):929–965, 1989.

[Blu05] Avrim Blum. On-line algorithms in machine learning. *Online algorithms: the state of the art*, pages 306–325, 2005.

[FS$^+$96] Yoav Freund, Robert E Schapire, et al. Experiments with a new boosting algorithm. In *icml*, volume 96, pages 148–156. Citeseer, 1996.

[GS91] Sally Ann Goldman and Robert Hal Sloan. *The difficulty of random attribute noise.* Citeseer, 1991.

[Kea98] Michael Kearns. Efficient noise-tolerant learning from statistical queries. *Journal of the ACM (JACM)*, 45(6):983–1006, 1998.

[KL88] Michael Kearns and Ming Li. Learning in the presence of malicious errors. In *Proceedings of the twentieth annual ACM symposium on Theory of computing*, pages 267–280, 1988.

[Lit88] Nick Littlestone. Learning quickly when irrelevant attributes abound: A new linear-threshold algorithm. *Machine learning*, 2:285–318, 1988.

[PV88] Leonard Pitt and Leslie G Valiant. Computational limitations on learning from examples. *Journal of the ACM (JACM)*, 35(4):965–984, 1988.

[Riv87] Ronald L Rivest. Learning decision lists. *Machine learning*, 2:229–246, 1987.

[Ros58] Frank Rosenblatt. The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological review*, 65(6):386, 1958.

[RSA78] Ronald L Rivest, Adi Shamir, and Leonard Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2):120–126, 1978.

[Slo89] Robert Hal Sloan. *Computational learning theory: new models and algorithms.* PhD thesis, Massachusetts Institute of Technology, 1989.

[Val84] Leslie G Valiant. A theory of the learnable. *Communications of the ACM*, 27(11):1134–1142, 1984.