1 Learning conjunctions in a Probably Approximately Correct framework

Narayana Santhanam

1.1 Background:

Discrete probability, expectation, union bound

1.2 Aim:

You should be able to set up a learning problem in the standard probabilistic framework (which we will call Probably Approximately Correct or PAC learning), specifically the notions of *confidence* and *accuracy.* Understand the purpose behind this formulation. For the problem of learning conjunctions, you should understand how to derive the PAC bound on the training sample, and be able to work out similar bounds for simple problems.

1.3 Content: Learning conjunctions

A *literal* is one of the following three binary valued functions of a binary valued variable x: x or $\bar{x}$ or 1, where $\bar{x}$ is the complement of x. A *conjunction* is the logical "AND" of literals corresponding to n binary variables. For example, if $n = 2$, there are 9 possible conjunctions

$$\mathcal{H}_2 = \{x_1 \wedge x_2, \bar{x}_1 \wedge x_2, x_2, x_1 \wedge \bar{x}_2, \bar{x}_1 \wedge \bar{x}_2, \bar{x}_2, x_1, \bar{x}_1, 1\}$$

In general, when you have n variables, there are 3^n possible conjunctions, and we will denote this set of functions by $\mathcal{H}_n$.

Now suppose we have binary random variables $X_1, \ldots, X_n$, and each X_i can take on a value of 0 or 1. We let the sample space (denoted by Ω) to be the set of all values the vector $(X_1, \ldots, X_n)$ can take (so the sample space is the set of 2^n length-n binary vectors)

A slightly pedantic note: to just represent the function arguments, we have used lower case letters of the corresponding random variables. In general, we reserve capitals for random variables. We will, of course, evaluate the function at points indicated by the random variables in the future—then the result of the evaluation is itself another random variable.

Hypothesis or concept class This is usually the first step of modeling any problem in learning—we come up with a set of possible hypothesis/concepts, where each concept is a function on Ω. In our first example, the hypothesis class is $\mathcal{H}_n$. Note how each conjunction in $\mathcal{H}_n$ is a function from Ω to $\{0, 1\}$.

Learning conjunctions is set up as follows: Imagine some hypothesis $c \in \mathcal{H}_n$ is true. Using the evaluations of c on some of the elements of Ω, our goal is to infer the function c.

A look at some concrete numbers will show us interesting this premise is. First, even for a small-ish $n = 50$, the size of Ω is 2^{50}, really large. The number of conjunctions is 3^{50}, even larger. Of these 3^{50} conjunctions, one is true, and we are going to see the evaluation of that function on a few points, no more than a 1000, let us say. From the evaluation on these few points, we have to estimate the function "reasonably". Needless to say, there may be many, many conjunctions that are consistent with the evaluations on these few points. To make this happen, we set up the problem probabilistically.

Training examples Fix some distribution D on this Ω. D need not be such that the components of the vector, X_i, are independent, it can be any valid probability assignment on Ω. In class, we reviewed basic probability definitions here, you will find them in (Section 1.2).

A *training example* is an element from Ω sampled according to the distribution D, and the value of the true hypothesis on this vector. For instance, assume $n = 2$, a training example could be the observation $X_1 = 0, X_2 = 1$, $c(X_1, X_2) = 0$.

Training sample For $i = 1, \ldots, m$, $X^{(1)}, \ldots, X^{(m)}$ are picked independently according to distribution D from Ω. c is the true concept from $\mathcal{H}_n$ (which we do not know but are trying to learn). Now let $Z^{(i)} = (X^{(i)}, c(X^{(i)}))$ be the *training sample* (basically, m binary vectors, independently chosen according to D, and the value of the concept on those vectors).

Once again, pay attention to the notation here, each of $X^{(i)}$ is a vector of n binary random variables, *i.e.,* $X^{(i)} = (X_1^{(i)}, \ldots, X_n^{(i)})$, and that vector is chosen according to D. While the $X^{(i)}$ are independent vectors (so, the training vector $X^{(1)}$ carries no information about $X^{(2)}$), the components of each of $X^{(i)}$ need not be (so, for instance, the first component of $X^{(1)}$ may tell us something about the third component of $X^{(1)}$) depending on what D is chosen.

We need to use $Z^{(1)}, \dots, Z^{(m)}$ in any way possible to learn the concept c.

Probability of error Consider any algorithm $\mathcal{A}$ that generates, using only the training data, an estimate $\hat{C}$ of the true concept $c \in \mathcal{H}_n$. Generally, $\hat{C}$ also belongs to $\mathcal{H}_n$, but here we will allow in addition to functions in $\mathcal{H}_n$, the zero function that assigns to all elements of Ω the value 0. We write $\hat{C}$ (capitals) because the output is a function of the randomly generated training sample, and is therefore a random variable itself. If we have not seen all members of Ω and their labels, it is possible that $\hat{C} \neq c$. Let $E_{\hat{C}}$ be the event that $\hat{C}(\mathbf{x}) \neq c$, ie

$$E_{\hat{C}} = \Big\{ \mathbf{x} \in \Omega : \hat{C}(\mathbf{x}) \neq c(\mathbf{x}) \Big\}.$$

Now, we don't care if $E_{\hat{C}}$ is a small set or a big set, whether it contains just one element of Ω or even constains most of the elements of Ω. What matters is its probability under the distribution D (perfectly possible that sets with large number of elements have small probability and vice versa). We focus on the probability $D(E_{\hat{C}})$, which we call the *probability of error.*

While we may not get the concept c exactly, we would at least want our estimate $\hat{C}$ to have small probability of error.

Algorithm: keep-them-consistent The simplest algorithm (call this $\mathcal{A}$) starts with $x_1 \wedge \bar{x}_1 \wedge x_2 \wedge \bar{x}_2 \cdots \wedge x_n \wedge \bar{x}_n$ (such a function would evaluate to 0 everywhere). We weed out any literal inconsistent with the training data, and output what remains at the end.

Can you come up with a sequence of steps that accomplishes this? Write this as a psuedocode.

Note how the algorithm $\mathcal{A}$ takes in a training sample, and outputs a hypothesis in a deterministic manner given the training sample. Therefore if the training sample is $Z^{(1)}, \dots, Z^{(m)}$ and output is $\hat{C}$, we write $\mathcal{A}(Z^{(1)}, \dots, Z^{(m)}) = \hat{C}$. Randomized algorithms are often used in practice as well (and those cannot generally be expressed as a function like above), but for this simple problem, it suffices to stick to deterministic algorithms as above.

Since there is at least one concept (the true c) that is consistent with the training data, we will be able to output some $\hat{C}$ using this approach. But in general, the training sample is far smaller than the size of Ω, so we will have not observed the function at all possible elements of Ω. So it is quite possible that multiple concepts are consistent with the limited observations we have seen. This scenario is almost always the case with learning problems.

If $\hat{C} \neq c$, what has happened is that $\hat{C}$ and c have the same labels on the points of Ω we have seen (but obviously differ on other points since they are different functions). As defined above, the set of all points on which they differ is precisely the set $E_{\hat{C}}$, and we measure the quality of $\hat{C}$ with the probability of error.

Misleading training samples The problem with a lot of learning is that the training data misleads us. It fools us into choosing a hypothesis that looks good on the training data but performs poorly in general (has high probability of error).

Let us see how likely it is that we end up with such a bad hypothesis. There could be different such misleading hypothesis, but let us begin by fixing any one and think about that.

Formally, fix some output $\hat{C}$ the algorithm can have yielded, and assume the probability of error of $\hat{C}$ against the true hypothesis exceeds a level ϵ (a number between 0 and 1) we are comfortable with, *i.e.,* $D(E_{\hat{C}}) > \epsilon$.

This means of course that the event $\hat{C}(X) = c(X)$, which corresponds to the set $E^c_{\hat{C}}$ (the complement of the set $E_{\hat{C}}$) has probability $\leq 1 - \epsilon$.

Let the event that the algorithm chooses $\hat{C}$ be $A_{\hat{C}}$. Now $A_{\hat{C}}$ happens to be the intersection of these two events (*i.e.,* both conditions must be true)

1. Each training vector $X^{(i)} \in E^c_{\hat{C}}$. We will denote this event as $\left(E^c_{\hat{C}}\right)^m$.

2. The algorithm may not really output $\hat{C}$ when presented with training vectors all in $E^c_{\hat{C}}$, namely $X^{(1)}, \ldots, X^{(m)} \in \left(E^c_{\hat{C}}\right)^m$. So in addition, $X^{(1)}, \ldots, X^{(m)}$ must be such that the algorithm outputs $\hat{C}$ on $\hat{C}$ on them. This really depends on the algorithm and hard to quantify exactly, but as we will see, it will not matter.

We are looking for an upper bound on the probability of $A_{\hat{C}}$, *i.e.,* $D(A_{\hat{C}})$. Now for any two sets S and T, $D(S \cap T) \leq D(S)$, so if we compute (an upper bound on) $D(\left(E^c_{\hat{C}}\right)^m)$, we can then claim $D(A_{\hat{C}}) \leq D(\left(E^c_{\hat{C}}\right)^m)$.

Computing an upper bound on $D(\left(E^c_{\hat{C}}\right)^m)$ is not too hard. Since the $X^{(i)}$ are all independently chosen from D,

$$D(\left(E^c_{\hat{C}}\right)^m) = \left(D(E^c_{\hat{C}})\right)^m \leq (1 - \epsilon)^m$$

where the last inequality above follows because $\hat{C}$ is a bad hypothesis ($D(E_{\hat{C}}) > \epsilon$). Thus the probability of choosing any one bad hypothesis $\hat{C}$ by the algorithm is

$$D(A_{\hat{C}}) \leq D(\left(E^c_{\hat{C}}\right)^m) \leq (1-\epsilon)^m.$$

But how many such bad hypotheses are there? We cannot know a precise number without more details on the algorithm and the true concept, but we do know that it is $\leq 3^n$, since there are only $3^n + 1$ possible outputs, one of which is correct. Surprisingly, we don't need to pursue a better answer to come up with something interesting. We are going to use the union bound, a seemingly trivial, but very useful observation:

$$\mathbb{P}(\cup_{i\geq 1} A_i) \leq \sum_{i\geq 1} \mathbb{P}(A_i).$$

We will use the union bound above as follows. Consider all bad hypothesis $\hat{C}_1, \ldots, \hat{C}_k$, where, as we observed trivially $k \leq 3^n$. Let $A_{\hat{C}_i}$ be the event we choose the hypothesis $\hat{C}_i$. Then the probability we choose a bad hypothesis is

$$D(\cup_{i=1}^k A_{\hat{C}_i}) \leq \sum_{i=1}^k D(A_{\hat{C}_i}) \leq k(1-\epsilon)^m$$

where the last inequality follows because the probability of any single bad hypothesis being chosen is $\leq (1-\epsilon)^m$. Since $k \leq 3^n$, the probability of choosing a bad hypothesis is

$$\leq 3^n (1-\epsilon)^m.$$

This is interesting, because the probability above drops exponentially as m increases. Indeed if (ln is logarithm to the natural base e)

$$m \geq \frac{1}{\epsilon}\left(\ln\frac{1}{\eta} + n\ln 3\right),$$

the probability of choosing a bad hypothesis is $< \eta$. Prove the above by using $1-\epsilon \leq e^{-\epsilon}$.

Probably Approximate Learning So the big picture is this. We output a conjunction consistent with the training data (and while you have to provide an algorithm above, you can obtain an algorithm that runs in time polynomial in n and m). We know that so long as $m \geq \frac{1}{\epsilon}\left(\log\frac{1}{\eta} + n\log 3\right)$, we will be assured that the our hypothesis is probably (with *confidence* $\geq 1 - \eta$ over choices of training samples), approximately correct (prob of error of output hypothesis $< \epsilon$).

This requirement on the training size is way way smaller than the size of Ω — 2^n —for example, when $n = 100$, m can be a little more than 100, but the size of Ω is 2^{100}, greater than the number of atoms in the universe.

In summary, having looked at a very small number of elements of Ω, very likely (ie with probability $\geq 1 - \eta$ over choices of training samples), we have chosen an approximately correct (prob of error of the output hypothesis $< \epsilon$) algorithm. We can set η and ϵ to be values as small as we want.

This is called "PAC" learning (PAC for probably approximately correct). Almost all learning has similar guarantees, since the training data is fundamentally randomly generated.