15-850(C) Machine Learning Theory

Lecturer: Avrim Blum
Scribe: Matt Glickman

Lecture 3: September 26, 1995

---

1 PAC Learning and Consistent Hypotheses

In the last lecture, when analyzing our algorithm for learning conjunctions, we proved the following important result about the PAC model as a byproduct. The result is that if the learning algorithm selects a sample of data of size

$$m \geq \frac{1}{\epsilon}\left[\ln|H| + \ln\left(\frac{1}{\delta}\right)\right],$$

then with high probability ($\geq 1-\delta$) there will be no "bad" hypotheses (error $\geq \epsilon$) in H consistent with the data. Thus, so long as $\ln|H|$ is polynomial in our parameters, it suffices to solve the consistency problem in order to achieve PAC learning.

To understand this statement, it is worth noting that as there are 2^n members of $\{0,1\}^n$, there are 2^{2^n} boolean functions over $\{0,1\}^n$. Thus, this requires H to be much smaller than the set of all boolean functions.

In fact, PAC learning and solving the consistency problem are equivalent in the following setting:

Theorem 1 *Suppose that concept class C satisfies $|C| \leq 2^{poly(n)}$, and all $c \in C$ are poly-time computable. Then, PAC-learning C by C is equivalent to solving the consistency problem for C in randomized polynomial time (i.e., in the case that there exists a hypothesis consistent with the examples, the randomized algorithm is permitted to fail with probability δ).*

Proof:

1. Solving the consistency problem for C in randomized polynomial-time implies PAC-learning:

   We can show this by splitting our confidence parameter δ over both the failure probability of the algorithm and the chance of a bad hypothesis being consistent with the data. Let L be a randomized polynomial-time algorithm to solve the consistency problem for C with a failure probability of $\delta/2$. We run L on a sample of size $m \geq \frac{1}{\epsilon}[\ln|C| + \ln(2/\delta)]$, which as we noted above implies that there is at most a $\delta/2$ chance of any bad hypothesis (error $> \epsilon$) being consistent with the data. (Note that m is polynomial in the PAC parameters because $|C| \leq 2^{poly(n)}$.) Therefore, after seeing at most m examples, our algorithm will have a total chance of at most δ (summing the chance of a bad, consistent hypothesis and the randomized failure probability) of outputing a hypothesis with error $> \epsilon$, qualifying it as a PAC algorithm.

2. PAC-learning C using C implies solving the consistency problem for C in randomized polynomial-time:

   Let S be the set of data for which we would like to solve the consistency problem. Drawing examples for our learner from a distribution $\mathcal{D}$ in which each example in S has probability $1/|S|$, we can force the PAC-learning algorithm to produce a consistent hypothesis with probability $1-\delta$ by setting $\epsilon = 1/(|S|+1)$. Because concepts in C are polynomial-time computable, we can then verify whether or not our hypothesis was in fact consistent with the data before producing our result. ■

So the good news is that in order to demonstrate that concept class C can be PAC-learned by L using C, we now know that it is sufficient to show that L solves the consistency problem for C in randomized polynomial time. We go on to employ this knowledge to demonstrate PAC-learnability in a new domain in the next section.

2 Decision Lists

Decision lists can be seen as a restricted form of decision trees in which each decision node has two branches, at least one of which leads to a leaf (value) node. This amounts to a chain of if-then-else rules.

Example: (when to play our favorite sport) If rainy then no else if warm then yes else if weekend then yes else if teaching-day then no else yes.

Definition 1 *A* **decision list** *is a list of rules:*

$$A_1 \rightarrow B_1,\ A_2 \rightarrow B_2,\ \ldots,\ A_{l-1} \rightarrow B_{l-1},\ True \rightarrow B_l$$

where A_i are literals and $B_i \in \{0,1\}$. The meaning is: given an example, we look at the first left-hand-side satisfied and output the corresponding right-hand-side.

Extension: *In* k**-decision lists**, *each left-hand-side may be a conjunction of up to k literals.*

2.1 How expressive are decision lists?

While decision lists are not fully expressive (consider the difficulty of representing a hypothesis that requires a full decision tree, e.g., a parity function), they are strictly more expressive than conjunctions and disjunctions. We can see that they are at least as expressive by noting first that any disjunction of literals can be represented by a series of rules in which any of the literals implies `positive`, and the default is `negative`. Now, notice that decision lists are closed under complement (simply complement the right-hand-side of each rule), which

means that decision lists can represent conjunctions too. Here is an example in which there is no consistent conjunction or disjunction, but there *is* a consistent decision list.

1	0	0	1	1	+
0	1	1	0	0	−
1	1	1	0	0	+
0	0	0	1	0	−
1	1	0	1	1	+
1	0	0	0	1	−

While there exists no conjunction or disjunction consistent with this data, the following decision list satisfies the set: $\bar{x}_1 \Rightarrow -$, else $x2 \Rightarrow +$, else $x4 \Rightarrow +$, else $-$.

Employing the same line of reasoning, we may see that k-decision lists are more expressive than both k-CNF and k-DNF.

2.2 How many decision lists are there?

From each variable, we can form four rules by specifying either the variable or its negation on the left-hand side, and either 0 or 1 on the right-hand side. As each rule classifies all those examples to which it applies, there is no point in applying a given rule more than once. These two observations taken together allow us to set an upper bound of $(4n)!$ on the number of decision lists. Further analysis leads us to a somewhat tighter bound of $n!4^n$. In either case, $\ln |C| = O(n \log n)$. Therefore, we know it suffices to find a consistent decision list in order to achieve PAC learning.

2.3 How can you find a consistent decision list?

Here's an algorithm:

1. Find some rule consistent with the current set of examples that applies to at least one of them. If no such rule exists, halt with failure.
2. Put the rule at the bottom of the hypothesis.
3. Throw out those examples classified by the hypothesis so far.
4. If there are any examples left, repeat from the beginning.

It is clear that this algorithm runs in polynomial time, since at any stage there are at most $4n$ rules to compare with the data, and the algorithm can run for at most $4n$ iterations. We now show that if there exists a consistent list, this algorithm will find one.

Suppose there exists some consistent decision list L^*, and suppose that some examples still remain in our data set. Let R be the highest rule in L^* that applies to at least one example in our current data set. Notice that R must be consistent with the data because, by the

definition of a decision list, any inconsistent example must cause a higher rule in L^* to fire, which contradicts our definition of R. Therefore, our algorithm has at least one legal choice (namely, rule R) in step 1.

3 What About More Expressive Hypotheses?

In general, the more expressive the hypothesis language, the greater the number of hypotheses that may, by chance, be consistent with a random set of training examples. Thus, as algorithms employ increasingly expressive hypothesis languages, it becomes increasingly difficult to place general bounds on the accuracy of the hypotheses they may produce. For example, an algorithm using the hypothesis class of general DNF formulas can easily output a maximally specific hypothesis that merely amounts to the disjunction of those postive examples in the training set. What kind of guarantee can we give for algorithms that use decision trees or DNF formulas as hypotheses? More generally, when does data justify a hypothesis?

4 Occam's Razor

The following statement is attributed to William of Occam ($\sim$ 1320 AD):

> "Entities should not be multiplied unnecessarily" (in Latin)

which is taken to mean (presumably as opposed to other more complex interpretations), "prefer simpler explanations".

This maxim is often cited to justify the choice of one scientific hypothesis over others, but what reason might we have to believe that it will lead us to hypotheses of lesser error?

One reason follows from the observation that there are fewer simple explanations than complex ones. Therefore, we might expect simple explanations to be less likely to fit data just by chance.

Definition 2 *A* **legal size function** *on concepts is one such that at most* 2^s *concepts have size* s*.*

Theorem 2 *Given* m *examples, with probability* $\geq 1-\delta$ *there are no bad (error* $\geq \epsilon$*) consistent hypotheses of size less than* s*, where:*

$$m \ > \ \frac{1}{\epsilon}\left(s \ln 2 + \ln \frac{1}{\delta}\right)$$

or

$$s \ < \ \frac{1}{\ln 2}\left(\epsilon m - \ln \frac{1}{\delta}\right).$$

Proof: The proof follows simply by noticing that the total number of hypotheses of size less than s is at most 2^s, and plugging in our earlier bound. ■

The above analysis indicates that by favoring smaller hypotheses over larger, we are less likely to run across bad hypotheses, reinforcing the view that

"Learning is data compression"

Reasoning from this perspective, we define an "Occam algorithm" to be one that produces hypotheses consistent with the data that are of size polynomial in the size of the concept, and grow only slowly in size with number of examples in the training set.

Definition 3 *$\mathcal{A}$ is an* Occam *algorithm for learning C by H if given any set S of m examples, if there exists a consistent $c \in C$ then $\mathcal{A}$ finds a consistent $h \in H$ with $size(h) \leq size(c)^\alpha m^\beta$, where α, β are constants and $\beta < 1$.*

Happily, as it turns out, those algorithms which satisfy the criteria for Occam algorithms turn out to be PAC-learners.

Theorem 3 *An Occam algorithm is a PAC-learning algorithm.*

Proof: By Theorem 2, we just need the number of examples m to satisfy:

$$m/2 \geq \frac{1}{\epsilon}(s \ln 2) \quad \text{and} \quad m/2 \geq \frac{1}{\epsilon} \ln \frac{1}{\delta}.$$

Using $s = size(c)^\alpha m^\beta$, this works out to the statement that:

$$m \;\geq\; \max\left\{ \left[\frac{2 \ln 2}{\epsilon} size(c)^\alpha \right]^{\frac{1}{1-\beta}}, \quad \frac{2}{\epsilon} \ln \frac{1}{\delta} \right\}$$

examples suffice. ■

Therefore, even if an algorithm employs a very expressive hypothesis space, as long as it is an Occam algorithm, we can make PAC-learning guarantees about it.

5 An Example: Learning a Convex Region in a Plane

Consider learning the concept class of convex regions in the XY plane, defined by their boundary segments. That is, a concept is a convex polygon, and points inside the polygon are positive examples and points outside are negative examples. In order to talk about the "size" of a concept as above, we assume that all points are given to n bits of precision, and therefore any boundary segment of the target concept can be described using $O(n)$ bits. Later on, when we talk about VC-dimension, we will be able to produce better bounds that do not require this assumption.

We can specify an algorithm to learn such a concept as follows. Given a set of data:

1. Find a line with the positive examples all on one side and as many negatives as possible on the other.

2. Throw out those negatives, and if any remain, repeat Step 1.

3. The end result is a convex polygon defined by the lines used.

If there exists some concept consistent with the training data, how many iterations will this algorithm require at maximum? If $s = |\text{boundary segments in the target}|$, then each line separates at least a $1/s$ fraction of negatives remaining. So, after L lines, at most $m(1-1/s)^L$ negative examples remain. Therefore

$$L = s \ln m$$

lines, or iterations, suffice. Because there are at most m^2 ways of splitting m examples using lines, there are $O(m^2)$ lines to consider during each iteration, so the algorithm runs in polynomial time. Also, because each line found requires only $O(n)$ bits to describe it, the size of the hypothesis found is $O(sn \ln m)$, and therefore this is an Occam algorithm.