CS787: Advanced Algorithms	
Scribe: Chris Hinrichs and Baris Aydinlioglu	**Lecturer:** Shuchi Chawla
Topic: PAC Learning	**Date:** November 16, 2007

In this lecture we continue our discussion of learning in the PAC (short for Probably Approximately Correct) framework.

25.1 PAC Learning

In the PAC framework, a **concept** is an efficiently computable function on a domain. The elements of the domain can be thought of as objects, and the concept can be thought of as a classification of those objects. For example, the boolean function $f : \{0,1\}^n \to \{0,1\}$ classifies all $0,1$ n-vectors into two categories. A **concept class** is a collection of concepts.

In this framework, a learning algorithm has off-line access to what we can call a "training set". This set consists of $\langle element, value \rangle$ pairs, where each *element* belongs to the domain and *value* is the concept evaluated on that element. We say that the algorithm can learn the concept if, when we execute it on the training set, it outputs a hypothesis that is consistent with the entire training set, and that gives correct values on a majority of the domain. We require that the training set is of small size relative to that of the domain but still is representative of the domain. We also require that the algorithm outputs its hypothesis efficiently, and further that the hypothesis itself can be computed efficiently on any element of the domain.

We used a number of qualitative terms in the above description (e.g., how do we tell if the training set represents the domain?). We formalize these concepts in a probabilistic setting in the following definition.

Definition 25.1.1 (Learnability) *Let C be a class of concepts that are defined over a domain T. We say C is* **PAC-learnable** *if there is an algorithm ALG such that for any distribution D over T, any concept $f \in C$, any $\delta > 0$ and $\epsilon > 0$, when ALG is given a set of examples $S = \{\langle x, f(x)\rangle : x \sim D\}$, of size $|S| = \text{poly}\left(\frac{1}{\epsilon}, \frac{1}{\delta}, n, \log|C|\right)$, ALG produces, in time $\text{poly}(|S|)$, a hypothesis $h \in C$ satisfying:*
(i) h is polytime computable,
(ii) h agrees with f on S,
(iii) $\mathbf{Pr}_{S\sim D}\left[\ \mathbf{Pr}_{x\sim D}\left[h(x) \neq f(x)\right] \leq \epsilon\ \right] \geq 1-\delta$.

In this definition $1-\delta$ captures our confidence in the algorithm and $1-\epsilon$ captures the accuracy of the algorithm. By forming the training set S with respect to a distribution D, and measuring the accuracy of the algorithm by drawing elements from the domain T according to the same distribution, we enforce that the training set represents the domain. The condition on the size of S captures the intuition that for greater accuracy or confidence we have to train the algorithm with more examples.

There are a number of concept classes that are learnable in the PAC framework. Next we study one of these, Decision Lists.

25.1.1 Decision Lists

A Decision List (DL) is a way of representing certain class of functions over n-tuples. Each n-tuple can be thought of as describing an object through a fixed set of n attributes. A DL consists of a series of if-then-else type rules, each of which references exactly one attribute, except the last rule which references at most one attribute.

For example, suppose we have the concept $f : \{0,1\}^5 \to \{0,1\}$, and we are given a set A of sample objects, $A \subset \{0,1\}^5$, along with the value of f on each element $x = \langle x_1, ..., x_5 \rangle$ of A, as follows:

x_1	x_2	x_3	x_4	x_5	$f(x)$
0	1	1	0	0	1
0	0	0	1	0	0
1	1	0	0	1	1
1	0	1	0	1	0
0	1	1	1	1	0

Then the following is a decision list that is consistent with, or that represents, f restricted to A.

if $x_4 = 1$ then $f(x) = 0$
else if $x_2 = 1$ then $f(x) = 1$
else $f(x) = 0$.

A rough upper bound on the number of all possible boolean decision lists on n boolean variables is $n!4^n = O(n^n)$, since there are $n!$ ways of permuting the attributes (x_i), 2^n ways of specifying conditions $(x_i = j)$ on those permuted attributes, and another 2^n ways of assigning values to the function $(f(x) = k)$ when each condition is true. From this we see that we cannot represent all possible boolean functions on n variables with decision lists as there are 2^{2^n} of them.

25.1.1.1 A PAC-learning algorithm for DLs

Let $f : T \to \{0,1\}$ be a concept that is representable as a decision list, defined over a domain $T = \{0,1\}^n$. Let $S = \{\langle x, f(x)\rangle : x \sim D\}$ be a training set.

The following algorithm takes as input S, and outputs a decision list h that is consistent with S:

repeat until $|B| = 1$, where $B = \{f(x) : \langle x, f(x)\rangle \in S\}$
  repeat until a rule is output
    let x_i be an attribute that doesn't appear in a rule yet
    let $B_j = \{f(x) : \langle x, f(x)\rangle \in S$ and $x_i = j\}$, where $j \in \{0,1\}$
    if $|B_j| = 1$ then
      output rule "if $x_i = j$ then $f(x) = k$ ", where $k \in B_j$
      output partial rule "else"
      $S \leftarrow S - \{\langle x, f(x)\rangle : x_i = j\}$
output rule "$f(x) = l$" where $l \in B$.

Since we know f is representable as a decision list, it follows that this algorithm always terminates. Clearly it runs in polynomial time. It is also clear that for any x in the domain, $h(x)$ is efficiently computable.

What remains to be shown, in order to prove that this algorithm can learn in the PAC framework, is the following. If we want our algorithm to output, with probability at least $1-\delta$, a hypothesis h that has an accuracy of $1-\epsilon$, it suffices to provide as input to our algorithm a training set S of size as specified in definition 1.

Lemma 25.1.2 *Let f be the target concept that our algorithm is to learn, and let h be its output. Suppose S is of size $|S| \geq \frac{1}{\epsilon}\left(N + \frac{1}{\delta}\right)$, where $N = O(n \log n)$ is the natural logarithm of the number of different decision lists. Then $\mathbf{Pr}_{S\sim D}[h \text{ is } 1-\epsilon \text{ accurate}] \geq 1-\delta$.*

Proof: Suppose $|S|$ is as in the statement of the claim. It suffices to show

$$\mathbf{Pr}_{S\sim D}[h \text{ is more than } \epsilon \text{ inaccurate}] \leq \delta\,. \quad (*)$$

Consider what it means for h to have an inaccuracy ratio more than ϵ. One way to interpret this is that our algorithm is "fooled" by the training set $S \sim D$, in the sense that even though the target concept f and the hypothesis h agree on all of S, they disagree on too-large a fraction of the domain. (We view the domain through D.) So we can rewrite (*) as

$$\mathbf{Pr}_{S\sim D}[\text{our algorithm is fooled}] \leq \delta\,. \quad (**)$$

For a given S, partition the set of all concepts that agree with f on S into a "bad" set and a "good" set. The bad set contains those hypotheses which, if our algorithm outputs one of them, would be fooled.

$$\mathcal{H}_{bad} = \{h \in C : h(x) = f(x)\,\forall x \in S\,, \text{ but } \mathbf{Pr}_{x\sim D}[h(x) \neq f(x)] \geq \epsilon\}\,.$$

Here C is the set of all decision lists. In order to succeed, our algorithm should output a member of $\mathcal{H}_{good}$.

$$\mathcal{H}_{good} = \{h \in C : h(x) = f(x)\,\forall x \in S\,, \text{ and } \mathbf{Pr}_{x\sim D}[h(x) \neq f(x)] \leq 1-\epsilon\}\,.$$

Notice that for a given S, $\mathcal{H}_{good}$ is never empty (contains f at least) but $\mathcal{H}_{bad}$ may be, in which case our algorithm definitely does not get fooled. Now we can rewrite (**) as

$$\mathbf{Pr}_{S\sim D}[\mathcal{H}_{bad} \text{ is non-empty and our algorithm outputs a member of it}] \leq \delta\,. \quad (***)$$

To show (***), it suffices to show

$$\mathbf{Pr}_{S\sim D}[\mathcal{H}_{bad} \text{ is non-empty}] \leq \delta\,,$$

which we shall do in the rest of the proof.

Consider any $h \in C$ such that $\mathbf{Pr}_{x\sim D}[h(x) \neq f(x)] \geq \epsilon$. The probability that we pick S such that h ends up in $\mathcal{H}_{bad}$ is

$$\mathbf{Pr}_{S\sim D}[h(x) = f(x)\,\forall x \in S] \leq (1-\epsilon)^m\,,$$

where $m = |S|$. Here we used the fact that S is formed by independently picking elements from the domain according to D.

What we have just obtained is the bound on a particular inaccurate $h \in C$ ending up in $\mathcal{H}_{bad}$. There are at most $|C|$ such h, so by a union bound,

$$\mathbf{Pr}_{S\sim D}\left[\mathcal{H}_{bad} \text{ is non-empty}\right] \leq |C|\,(1-\epsilon)^m\ .$$

All that is left is to show that the right hand side of the last inequality is at most δ.

Note that by definition $|C| = e^N$, and using $1-\epsilon \leq e^{-\epsilon}$, we have $\frac{1}{\epsilon} \geq \frac{1}{\log\left(\frac{1}{1-\epsilon}\right)}$. Putting together, we get

$$\begin{aligned} m &\geq \frac{1}{\log\left(\frac{1}{1-\epsilon}\right)}\left(\log|C| + \log\frac{1}{\delta}\right) \\ -m\log\left(1-\epsilon\right) &\geq \log|C| + \log\frac{1}{\delta} \\ \log\delta &\geq \log|C| + m\log\left(1-\epsilon\right), \end{aligned}$$

which implies that

$$\mathbf{Pr}_{S\sim D}\left[\mathcal{H}_{bad} \text{ is non-empty}\right] \leq |C|\,(1-\epsilon)^m \leq \delta\,,$$

proving the lemma. ■

25.1.2 Other Concept Classes

Notice that in the above lemma we used the concept class being DLs only when we substituted $n\log n$ with $\log|C|$. This suggests the following:

Theorem 25.1.3 (Consistency ⇒ Learnability) *Let C be a concept class. Suppose that there is an algorithm that, given a sample set S, can efficiently (in time* $\text{poly}\,|S|$*) solve the consistency problem (i.e. produce a hypothesis $h \in C$ that is consistent with S). Then C is PAC-learnable with sample complexity $|S| = \frac{1}{\epsilon}\left(\log|C| + \log\frac{1}{\delta}\right)$.*

Proof: Once we show such an algorithm exists, the PAC-learnability of C follows by an invocation of the above lemma with $|S| \geq \frac{1}{\epsilon}\left(\log|C| + \log\frac{1}{\delta}\right)$. ■

We apply this theorem to some other concept classes below.

25.1.2.1 Decision Trees

A Decision Tree (DT) is an extension of DLs where each condition can refer to more than one attribute. If DLs and DTs are represented as graphs with rules as vertices and connections between rules as edges, then a DL has a linked list structure, and a DT has a tree structure.

DTs are powerful enough to represent any boolean function. For example, $f : \{0,1\}^n \to \{0,1\}$ can be represented by a DT with root node labeled x_1, two children of the root node both labeled x_2 at the second level, and so on where at the last (nth) level there are 2^n nodes all labeled x_n.

Given a sample set S, we can efficiently produce a consistent hypothesis by essentially "fitting" a DT to S. For each sample in S, we add a branch that encodes that sample to the DT. The final hypothesis will look like a DT as described in the previous paragraph, though only with $|S|$ leaves. Clearly this procedure runs in time $\text{poly}\,|S|$.

It follows by the Theorem 25.1.3 that DTs are PAC-learnable. But usually we are interested in representing data with small decision trees, and not decision trees of arbitrary size. So one might ask, can we learn decision trees with depth at most k, or decision trees with at most k nodes, etc. Unfortunately solving the consistency problem for such questions is NP-hard.

For example, suppose our target concept is a DT with number of nodes at most $g(|S|)$, where $g(|S|) = o(|S|)$. The concept class in this case is all DTs with less than $g(|S|)$ nodes. So $|C| = n^{g(|S|)}$ and we can PAC-learn C with $|S| \geq \frac{1}{\epsilon}\left(g(|S|)\log n + \log\frac{1}{\delta}\right)$, provided that we come up with an efficient algorithm that can solve the consistency problem on S.

The problem is that such an algorithm is unlikely to exist, because it is an NP-hard problem to produce, given S, a decision tree with $o(|S|)$ nodes that is consistent with the entire sample set.

So DTs of restricted size are not PAC-learnable, although those of arbitrary size are.

25.1.2.2 AND-formulas

An AND-formula is a conjunction of literals, where a literal is an attribute or its negation. For example, $x_1 \wedge \neg x_2 \wedge x_n$ is an AND-formula. There are 2^{2n} AND-formulas on n literals.

Given a training set S we can efficiently produce a consistent hypothesis as follows. Start out with the formula $x_1 \wedge \neg x_1 \wedge ... \wedge x_n \wedge \neg x_n$, then for each $x \in S$ with $f(x) = 1$, if $x_i = 0$ then eliminate x_i from the formula, otherwise eliminate $\neg x_i$; do this for $i = 1..n$.

So this class is PAC-Learnable.

25.1.2.3 OR-formulas

Similar to AND-formulas. In this case we construct a hypothesis by looking at $x \in S$ with $f(x) = 0$. So this class is also PAC-Learnable.

25.1.2.4 3-CNF formulas

A 3-CNF formula is a conjunction of clauses, each of which is a disjunction of exactly 3 literals. There are $2^{\text{poly}(n)}$ 3-CNF formulas on n literals.

We can reduce the problem of constructing a hypothesis to that in the AND formulas section, by mapping each of the $\binom{2n}{3}$ possible clauses that can be formed from n literals into new variables y_i. Since this reduction is polytime, and the algorithm for the AND-formulas is polytime, we obtain a polytime procedure that outputs a hypothesis.

So this class is PAC-learnable.

25.1.2.5 3-DNF formulas

A 3-DNF formula is a disjunction of clauses, each of which is a conjunction of exactly 3 literals.

The analysis is similar to 3-CNF (in this case it reduces to the OR-formulas).

25.1.2.6 3-term DNF formulas

A 3-term DNF formula is a disjunction of exactly 3 clauses, each of which is an AND-formula.

Since a 3-term DNF formula is functionally equivalent to a 3-CNF formula, we may be tempted to conclude that 3-term DNF formulas are also learnable, as we already showed above an algorithm that can efficiently solve the consistency problem for 3-CNF formulas.

The problem with this is that 3-CNF formulas are a superset of 3-term DNF formulas, and if we reuse the algorithm from the 3-CNF formula case, we may get a hypothesis that does not belong to the class of 3-term DNF formulas. It could take exponential time to go through all possible 3-CNF formulas consistent with the sample set until we find one that is also a 3-term DNF formula—and that is assuming we can quickly tell if a 3-CNF formula is also a 3-DNF formula.

In fact, it turns out that it is an NP-hard problem, given S, to come up with a 3-term DNF formula that is consistent with S. Therefore this concept class is not PAC-learnable—but only for now, as we shall soon revisit this class with a modified definition of PAC-learning.

25.2 PAC-Learning—a revised definition

It seems somewhat unnatural to require that the hypothesis be in the same class of concepts as that of the target concept. If, instead, we allow the hypothesis to be from a larger concept class then we might be able to learn some concepts that would otherwise be not learnable. This leads to the following revised definition of PAC-learning.

Definition 25.2.1 (Learnability—revised) *Let C be a class of concepts that are defined over a domain T. We say C is* **PAC-learnable** *if there is an algorithm ALG such that for any distribution D over T, any concept $f \in C$, any $\delta > 0$ and $\epsilon > 0$, when ALG is given a set of examples $S = \{\langle x, f(x)\rangle : x \sim D\}$, of size $|S| = \text{poly}\left(\frac{1}{\epsilon}, \frac{1}{\delta}, n, \log|C|\right)$, ALG produces, in time $\text{poly}(|S|)$, a hypothesis $h \in C'$— where C' is some (possibly different) concept class—satisfying:*
(i) h is polytime computable,
(ii) h agrees with f on S,
(iii) $\mathbf{Pr}_{S\sim D}\left[\ \mathbf{Pr}_{x\sim D}\left[h(x) \neq f(x)\right] \leq \epsilon\ \right] \geq 1 - \delta$.

With this new definition, Theorem 25.1.3 also needs to be revised.

Theorem 25.2.2 (Consistency ⇒ Learnability—revised) *Let C be a concept class. Suppose that there is an an algorithm that, given a sample set S, can efficiently (in time $\text{poly}\,|S|$) solve the consistency problem (i.e. produce a hypothesis $h \in C'$ that is consistent with S). Suppose further that $\log|C'| = O(\text{polylog}\,|C|)$. Then C is PAC-learnable with sample complexity $|S| = \frac{1}{\epsilon}\left(\log|C'| + \log\frac{1}{\delta}\right)$.*

Proof: Invoke Lemma 25.1.2 with $|S| \geq \frac{1}{\epsilon}\left(\log|C'| + \log\frac{1}{\delta}\right)$. ■

We now turn to the two concept classes that we previously concluded not-learnable based on the original definition, and see if the revised definition makes a difference.

- **3-term DNF formulas**: Now we are allowed to output a 3-CNF formula even if it is not equivalent to a 3-term DNF formula. So we can run the algorithm from the 3-CNF

formulas discussion. Then C' is the class of 3-CNF formulas, and $\log|C'| = \log 2^{\text{poly}\,(n)} = \text{polylog}\, 2^{O(n)} = O(\text{polylog}\,|C|)$, so 3-term DNF formulas are now PAC-learnable.

- **Decision Trees of Small Size**: Suppose we run the algorithm from the above discussion regarding unrestricted DTs. Then C' is the class of all DTs, and we can conclude that DTs of small size are PAC-learnable if it is the case that $\log|C'| = \text{polylog}\,|C|$. Unfortunately it is usually not the case. For example, if we are interested in learning decision trees with $\text{poly}\, n$ nodes, then $\log|C'| = \log 2^{2^n} \gg \text{poly}\, n = \text{polylog}\,(n^{\text{poly}\, n}) = \text{polylog}\,|C|$. Therefore, even with the revised definition, the class of DTs of small size are not learnable—at least when C' is the class of all DTs.

25.3 Occam's Razor

One use of the theory of PAC learnability is to explain the principle Occam's Razor, which informally states that simpler hypotheses should be preferred over more complicated ones. Slightly more formally we can say that

$\text{size}(f) \leq s \forall f \in C \implies |C| \leq 2^s$

where $\text{size}(f)$ refers to the amount of information (usually measured in bits) sufficient to represent f. In other words, if all functions $f \in C$ have descriptions which fit into s bits then there are at most only 2^s such functions. If we can choose a hypothesis belonging to a smaller concept class C, and having description length s, then we can show the relationship between s and expected error rate ϵ if we treat δ as fixed.

Since $\delta \geq |C|(1-\epsilon)^m$ we can say that

$$\delta \geq 2^s(1-\epsilon)^m$$

$$\log(\delta) \geq s - \epsilon m$$

$$\epsilon \approx \frac{s + \log(\frac{1}{\delta})}{m}$$

Thus ϵ is directly related to $\frac{s}{m}$ (where m is the number of training examples).

25.3.1 Learning in the Presence of Noisy Data

We saw in Theorems 25.1.3 and 25.2.2 that if we can efficiently find a hypothesis h that is consistent with a given sample set S then h has, with high probability, the desired accuracy on the domain, provided that S is large enough. We also saw in our discussion on DTs of a certain size that we sometimes don't know how to efficiently find such h. But sometimes it may be even impossible to solve the consistency problem, no matter how much computational power we have, because S may have erroneous data (noise), so that no concept in the target concept class is actually consistent with S. In this section we discuss whether it is possible, even in such a setting, to produce h that has desired accuracy with high probability.

25.3.1.1 Uniform Convergence Bounds (a.k.a. Generalization Error Bounds)

In this noisy setting our examples take the form $S = \{\langle x, f(x)\rangle_{\sim D}\}$, where f is not necessarily in C, (though h is). We seek a function in C that has the least error with respect to f.

The least such error is:

$$\min_{h \in C} \mathbf{Pr}_{x \sim D}[h(x) \neq f(x)]$$

Now we train our classifier h on the data S, but there is one subtle difference: we choose h which minimizes the error on examples x drawn from S, (i.e. empirical error,) rather than the error on examles drawn from D, (i.e. the true error).

$$h = \arg\min_{h \in C} \mathbf{Pr}_{x \in S}[h(x) \neq f(x)]$$

We will show that for large enough S, an h with low empirical error will have low true error, with high probability.

We define ϵ' as the true error of h on f.

$$\epsilon' = \mathbf{Pr}_{x \sim D}[h(x) \neq f(x)]$$

then since the samples x are drawn independently from D, we can apply Chernoff's bound:

$$\mathbf{Pr}\left[\frac{|x \in S : h(x) \neq f(x)|}{|S|} < \epsilon' - \epsilon\right] < e^{-(2\epsilon^2 m)}$$

This means that with probability $1 - e^{-2\epsilon^2 m}$, the true error of h is no more than ϵ plus its empirical error on S. Now if we define

$$\delta = |C| e^{-(2\epsilon^2 m)}$$
$$\frac{\delta}{|C|} = e^{-(2\epsilon^2 m)}$$

and take the log, then

$$2\epsilon^2 m = \log(|C|) + \log(\frac{1}{\delta})$$

and so

$$\epsilon = \sqrt{\frac{\log(|C|) + \log(\frac{1}{\delta})}{2m}}$$

which gives us ϵ and δ. Now we know that $\mathbf{Pr}[\exists h \in C |$true error of h $>$ empirical error of h on S $+ \epsilon] < \delta$

In other words, with probability $1 - \delta$, $\forall h \in C$, the true error of $h \leq$ empirical error $+ \sqrt{\frac{log(|C|)+\frac{1}{\delta}}{2m}}$
Note that this bound is not quite as good as the bound on the case where we are drawing both f and h from C, (different by a square root factor, and m has a 2 in front of it).

References

[1] Wikipedia. http://en.wikipedia.org/wiki/William_of_Ockham