**CS 598: Theoretical Machine Learning**

Lecturer: Pranjal Awasthi
Scribe: Jeremy Bierema

Lecture #2
9/8/2017

# 1 Administrative

The course webpage, `https://www.cs.rutgers.edu/~pa336/mlt_f17.html`, is now up. It has some final project ideas that you can start thinking about now.

# 2 PAC model review

## 2.1 Setup

In the last class, we introduced the PAC model, which provides the mathematical setup and parameters for quantifying the degrees to which machine learning algorithms are **p**robably **a**pproximately **c**orrect.

Let us recall the setup. We established an instance space $X$ and an unknown function $f : X \to \{+1, -1\}$ that classifies the inputs into two categories. We assume that there is a distribution $D$ over $X$ and a training set $S = \{(x_1, y_1), (x_2, y_2), \ldots, (x_m, y_m)\}$ with each $x_i$ drawn independently according to $D$ and with each $y_i = f(x_i)$.

We want our machine learning algorithm to take $S$ as input and output a function $\hat{f}$ as close as possible to $f$, that is, an $\hat{f}$ with $\text{err}(\hat{f}) = P_{x \sim D}\big(\hat{f}(x) \neq f(x)\big)$ as small as possible. An algorithm $A$ is said to *PAC learn* $f$ if $\forall \epsilon, \delta > 0$, $\forall D$, $A$ uses a finite set of training examples and outputs $\hat{f}$ such that with probability $\geq 1 - \delta$, $\text{err}(\hat{f}) \leq \epsilon$.

Note that in general, we can never achieve zero error, since the number of possible functions $f$ is usually infinite. Similarly, we cannot succeed with probability 1, since there is always a possibility that the training set is unrepresentative and unhelpful.

## 2.2 Threshold function example

We considered the design of a machine learning algorithm for threshold functions to demonstrate the PAC model in action. We let $X = [0, 1]$, and we set

$$f(x) = \begin{cases} +1, & x > \theta \\ -1, & x \leq \theta. \end{cases}$$

Our algorithm was to look at the interval from the greatest negative example in the training set to the least positive example and arbitrarily choose our estimated $\theta$ in the interval. $A$ PAC learns $f$ if $m \geq \frac{1}{\epsilon} \ln \frac{2}{\delta}$. (Check this against the inequality we stated last time.) This expression shows that if you want less error, you need a larger training set. Similarly, if you want a higher success probability, you also need a larger training set.

# 3 ERM, a universal learning algorithm

In the threshold function example, it is important to note that our algorithm depended on our knowledge about the class of functions generating the data. It is natural to ask, then, is there a generic learning algorithm for any type of function? And we will show today that, surprisingly, the answer for functions over finite domains is yes. The algorithm is called *empirical risk minimization.*

## 3.1 Motivation and definitions

In our previous example, we were told the "type" of an unknown function, that is, that it was a threshold function. Our strategy was to find a function of the same type that perfectly fit the training set data. This is the principle of empirical risk minimization: find a function in the class of possible choices that fits the training data as closely as possible and output it. After all, all you have to work with is your training set.

Now we state the ERM algorithm more formally. Suppose the true function describing the data is $f \in H$, where $H$ is some set of functions, which for today is finite. (In our example, $H$ was the set of threshold functions.) Given a training set $S = \{(x_1, y_1), \ldots, (x_m, y_m)\}$, we want to find a function matching the training set, so we choose $\hat{f} = \arg\min_{g \in H} \mathrm{err}_S(g)$, where $\mathrm{err}_S(g)$ denotes the fraction of mistakes that $g$ makes on $S$. In words, $\hat{f}$ is the function making the minimum number of mistakes on $S$.

Intuitively, we want $H$ to be small to minimize the number of examples needed for training. Note that in this setting, we will always output a function with zero error on the training set since $f \in H$. But what is the error on the testing set?

## 3.2 ERM PAC learns finite function classes

We now prove that ERM PAC learns any $f \in H$ provided $m \geq \frac{1}{\epsilon} \ln \frac{|H|}{\delta}$. Note that the inequality is similar to the one we derived for threshold functions.

To prove the statement, suppose we are given a randomized training set $S$ of fixed size $m$ with elements chosen i.i.d. and we choose $\hat{f} = \arg\min_{g \in H} \mathrm{err}_S(g)$. As we observed, $\mathrm{err}_S(\hat{f}) = 0$. We call a function $h \in H$ that satisfies $\mathrm{err}(h) > \epsilon$ a "bad function"; if the ERM algorithm were to output these functions, this would violate our desired accuracy bound. The other functions are good, since they satisfy our accuracy bound. We want to bound $P_S\big(\exists h \in H, \mathrm{err}(h) > \epsilon \text{ and } \mathrm{err}_S(h) = 0\big)$. If $\mathrm{err}_S(h) = 0$, then $h$ has the potential to be chosen by $A$, and we cannot allow this if $h$ is bad (has high error).

Why are we looking at the probability that there is even just one bad function with $\mathrm{err}_S(h) = 0$? Remember that ERM does not say which function to output of the ones that match the training set, so we must be ready for the worst of these to be chosen. We are bounding the probability that some bad $h$ fools $A$.

As an aside, if we do not have a sizable, representative training set, we cannot expect to succeed. For example, for the threshold problem, if we just have a single data point and its label in the training set, there are many bad functions that could fool us. As we increase the side of the data set, we hope to catch bad functions with high probability.

Let $E_h$ denote the event that $\mathrm{err}(h) > \epsilon$ and $\mathrm{err}_S(h) = 0$, that is, the event that the function $h$ is bad but it fools $A$ that it is good since its error on the training set is zero. Note that if $\mathrm{err}(h) > \epsilon$, each time we draw a random example, $h$ has a probability greater than $\epsilon$ of making an error, since $P_{x \sim D}\big(h(x) \neq f(x)\big) = \mathrm{err}(h) > \epsilon$. The probability that

$h$ looks good on any particular element of the training set (fooling $A$) is less than $1-\epsilon$. Since the training set elements are chosen independently, the probability that $h$ fools $A$ on all of the training set is $P(E_h) \leq (1-\epsilon)^m \leq e^{-\epsilon m}$. As $m$ increases, this becomes small.

This bound is for a single bad $h$. There may be many bad functions. We can compute

$$P(\text{some bad function fools } A) = P\left(\bigcup_{h \in H} E_h\right) \leq \sum_{h \in H} P(E_h) \leq |H| e^{-\epsilon m}$$

by the union bound. If we want to succeed with probability at least $1-\delta$, we can set

$$\begin{aligned} & |H| e^{-\epsilon m} < \delta \\ \implies & e^{-\epsilon m} < \frac{\delta}{|H|} \\ \implies & m \geq \frac{1}{\epsilon} \ln \frac{|H|}{\delta}. \end{aligned}$$

With this many examples, we can be confident of outputting a good function, so this is the number of examples we need to obtain our desired accuracy.

We can also rearrange this inequality to quantify the uncertainty of our algorithm's success in the case of a small training set. So say we have a fixed $m$ and we run ERM to get an $\hat{f}$ with $\text{err}_S(\hat{f}) = 0$. Then with probability at least $1-\delta$, we have $\text{err}(f) \leq \frac{1}{m} \log \frac{|H|}{\delta}$. Note that this decreases at the rate $\frac{1}{m}$. Also note that, for a larger function class, we expect a larger error.

### 3.3 Limitations

So if we have a universal algorithm, why are we not finished? Well, $H$ could be huge. However, we will see in a future lecture that the size of $H$ is not always the important thing. For example, in the threshold example, $H$ was infinite, but we still got a good bound.

However, there are two other problems that arise in practice. First, $H$ is typically unknown. In the real world, we can observe data, but if we do not have an *a priori* knowledge of the class of possible functions, what can we do? How can we minimize the training set error on $H$ if we do not know what $H$ is? We can sometimes guess, but how do we know whether our guess is right?

There is also an algorithmic challenge: finding $\hat{f}$ might not be easy. ERM tells us to choose $\hat{f} = \arg\min_{g \in H} \text{err}_S(g)$, but it does not specify what algorithm to use to locate it. We often need to look for clever algorithms to solve this ERM problem efficiently.

These are the two challenges that make machine learning difficult.

Another possible complication is that samples might not be chosen independently. However, it turns out that even with some weaker assumptions, the bound will not change too much. A good survey topic for the final project would be how to get these bounds when the data is not i.i.d.

### 3.4 Boolean disjunction example

Suppose $H$ is the class of Boolean disjunctions over $n$ variables $x_1, x_2, \ldots, x_n \in \{0,1\}$. So an $f \in H$ could be $f = x_1 \vee x_2 \vee x_3$. Another example would be $f = x_1 \vee \overline{x_2} \vee x_4$, where $\overline{x_2}$ denotes the complement of $x_2$. So we are told that the unknown function is a Boolean

disjunction, and we want to use the ERM principle to PAC learn $f$. More formally, our algorithm $A$ will receive the training set $S$ as input and will seek to output an $\hat{f}$ which is a disjunction such that $\text{err}_S(\hat{f}) = 0$. We have a promise that $S$ is labeled according to some disjunction, and we want to find $\hat{f}$ with zero error.

The naïve algorithm is simply to check, for each $g \in H$, whether $\text{err}_S(g) = 0$, outputting one for which this is true. There are $|H|$ functions to check, and for each one we do $O(m)$ work to process the $m$ literals in the disjunction, so the run time is $O(|H|m)$. The number of possible clauses is about $3^n$ since for each variable $x_i$, we can either include $x_i$ in the disjunction, include $\overline{x_i}$ in the disjunction, or include neither, three choices for each of the $n$ variables.[1] Thus, the run time is $O(3^n m)$, which is exponential in $n$.

But for this problem, there is also a natural polynomial-time algorithm. We initialize $\hat{f}$, our algorithm's output, to $x_1 \vee \overline{x_1} \vee x_2 \vee \overline{x_2} \vee \cdots \vee x_n \vee \overline{x_n}$. Then we iterate through the training set, checking for each example whether our current $\hat{f}$ is correct for that input. Say we have an input $(1, 0, 0, 1, 1, 0, -1)$, by which we mean that $x_1$, $x_4$, and $x_5$ are set to TRUE; $x_2$, $x_3$, and $x_6$ are set to FALSE; and the output of the original function $f$ is FALSE. Then we can throw away the literals $x_1$, $\overline{x_2}$, $\overline{x_3}$, $x_4$, $x_5$, and $\overline{x_6}$ from $\hat{f}$, since if any of these were in $f$, the output would be TRUE. If we continue throwing away variables like this for negative training examples, we will never throw away variables in $f$. For each mistake we will throw away at least one literal and make $\hat{f}$ correct on that example. Note that we will never make a mistake on a positive example. The run time of the algorithm is $O(nm)$ since we do $O(n)$ work for each of the $m$ training examples.

Thus, we can do efficient ERM for this problem, but only by using a specialized algorithm that depends on the form of the function class $H$.

### 3.5 Boolean conjunction example

Now consider the PAC learning problem when $H$ is the class of Boolean conjunctions. Our algorithm here is analogous to that for disjunctions. We start by initializing $\hat{f}$ to $x_1 \wedge \overline{x_1} \wedge x_2 \wedge \overline{x_2} \wedge \cdots \wedge x_n \wedge \overline{x_n}$. We look at all *positive* examples in the training set, say $(1, 0, 1, 1, \ldots, +1)$. For this example we can remove the literals $\overline{x_1}$, $x_2$, $\overline{x_3}$, $\overline{x_4}$, and so on, from $\hat{f}$. The run time for this problem is the same, $O(nm)$.

These two examples are special cases of linear models. We can use support vector machines as general models to solve these and other problems.

### 3.6 Decision list example

Now suppose the functions in $H$ are decision lists over $n$ Boolean variables. As an example, consider the decision list depicted in Figure 1. To evaluate an input against this function, we proceed from left to right. If the first test, $x_1 = 1$, passes, then we output the function value below that test, $+1$. Otherwise we move on to the next test, $x_2 = 0$, and if that passes, we output the value below it, $-1$. We continue in this way. If no test passes, we output the value at the right side of the diagram, $-1$.

We want an algorithm to find a decision list that matches the training data. The naïve algorithm always works. However, the number of possible decision lists is roughly $n!\, 4^n$, since we have $n!$ orderings of the $n$ variables, and for each of those variables, we have to choose two Boolean values (the value to test the variable against and the value to output

[1] A disjunction that includes both some $x_i$ and its complement $\overline{x_i}$ always evaluates to TRUE, so this brings the total number of disjunctions to $3^n + 1$ when $n > 0$.

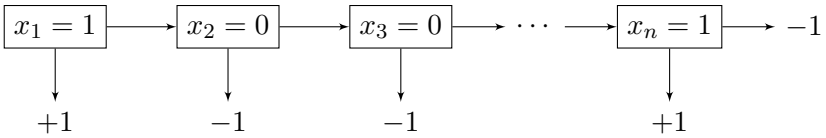


Figure 1: An example decision list.

if the test passes), giving four options per variable.[2] So again, we need to specialize our algorithm if we want to achieve polynomial time.

First, let us observe that since the true function $f$ is a decision list, there is some variable and value such that whenever that variable takes that value, the function is fixed. (These are the variable and value that appear at the front of the decision list for $f$.) So our algorithm should find an $x_i$ and a $v$ such that whenever $x_i = v$, points in $S$ have the same function value. This is easily done in polynomial time (actually in time linear in the input size, which is $O(mn)$). We can immediately put this variable with the value it takes and the output value at the front of our new decision list. Then we can reduce the training set by removing the examples that match this variable and value and by removing this variable from the remaining examples. This generates a new training set $S'$, and we recurse on $S'$. Note that if $S$ can be classified by a decision list, so can any subset, and since $x_i$ is constant on all of the training examples that were not removed, it can be removed from the examples safely. So there is a decision list that fits $S'$. We can recurse until we get a decision list that fits all of the training set.

The total run time is $O(nm^2)$ since we are at most doing linear work $m$ times, so this is a polynomial-time algorithm.

### 3.7 Decision tree example

Now consider the PAC learning problem where $H$ is the class of functions representable by decision trees. An example decision tree is given in Figure 2. The leaves of the tree contain function values, either $+1$ or $-1$. To evaluate an input assignment, start at the root of the tree, in this case $x_1$, and move to one of the children along the edge that corresponds to the variable's value. Continue to traverse the tree in this way until you reach a leaf, which holds the resulting function value for the input assignment. Note that a decision list is a special case of a decision tree.

How do we fit a decision tree to a data set $S$? Well, if all examples in $S$ have value $+1$, the decision tree is a $+1$ leaf. If all examples in $S$ have value $-1$, the decision tree is a $-1$ leaf. Otherwise, we can fix an arbitrary $x_i$ and partition $S$ into $S_1$ and $S_2$ where $S_1$ contains examples in $S$ where $x_i = 0$ and $S_2$ contains examples in $S$ where $x_i = 1$. We can arrange these in a tree structure as shown in Figure 3. Then we can recurse on each of $S_1$ and $S_2$ for the remaining variables to form the two subtrees. Eventually once we have checked all variables we will be done.

The depth of the resulting tree is at most $n$. Each example only follows one path to the leaves, so each example is processed $O(n)$ times as it is copied at most $n$ levels down. Thus one can show that the total run time is $O(mn)$. So we have a polynomial-time algorithm.

Note, however, that though our algorithm is efficient, it could produce a tree with $m$ separate leaves, one for each training example. This will eliminate the error on the

[2]We get $n!\,2^n$ as a trivial lower bound for the number of *distinct* functions representable by decision lists by only counting functions where the output values alternative from left to right in the figure.

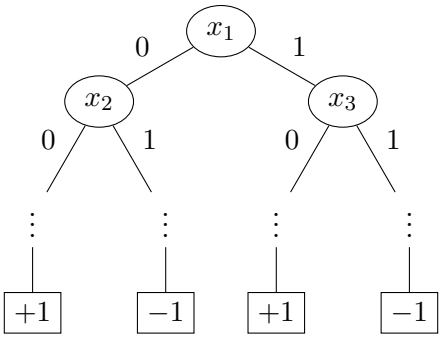


Figure 2: An example decision tree.

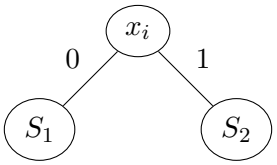


Figure 3: A decision tree in development.

training set, but the complexity of the tree is not well bounded. This is worrisome because it means that our resulting model might be too closely tailored to the training set and not general enough to correctly classify new examples.

Recall that for our Boolean disjunction example, we got $|H| \approx 3^n$, so that we needed roughly $m \geq \frac{n}{\epsilon} \log \frac{1}{\delta}$ to achieve PAC parameters $\delta$ and $\epsilon$. For decision lists, we had $|H| \approx n!\, 4^n$, requiring $m \geq \frac{n \log n}{\epsilon} \log \frac{1}{\delta}$, more or less. These are examples of efficient PAC learning since we can get a small error even while keeping the number of examples small and the run time polynomial.

However, decision trees can represent any Boolean function. Since a function on $n$ Boolean variables has $2^n$ possible inputs and there are two possible outputs for each of these inputs, there are $|H| = 2^{2^n}$ distinct functions in the class of decision tree functions. Thus, the ERM theorem requires roughly $m \geq \frac{2^n}{\epsilon} \log \frac{1}{\delta}$ examples to satisfy our PAC guarantees. This is exponential in $n$ since the class of decision trees is huge. So decision trees are not PAC learnable since the number of examples needed is exponential in $n$. Decision trees are too powerful to be learned easily.

We are generally interested in the efficiently PAC learnable regime.

# 4 Preview

In the next lecture, we will look at how to do PAC learning if $|H|$ is infinite. Most interesting classes fall into this regime. Note that we cannot use our current bound, since that would say we need an infinite number of training examples. We will see what is the right bound and discuss how to do ERM over infinite function classes.