8803 Machine Learning Theory

Maria-Florina Balcan Lecture 2: January 14, 2010

Plan: Review the PAC model and talk about simple PAC algorihms for learning boolean classes; talk about the Perceptron algorithm for learning linear separators.

1 The PAC Model

The basic idea of the PAC model is to assume that examples are being provided from a fixed (but perhaps unknown) distribution over the instance space. The assumption of a fixed distribution gives us hope that what we learn based on some training data will carry over to new test data we haven't seen yet.

Definition 1 *Given a example distribution $\mathcal{D}$, the* error *of a hypothesis h with respect to a target concept c is $\mathbf{Prob}_{x\in\mathcal{D}}[h(x) \neq c(x)]$. ($\mathbf{Prob}_{x\in\mathcal{D}}(A)$ means the probability of event A given that x is selected according to distribution $\mathcal{D}$.)*

In the following definition, "n" denotes the size of an example.

Definition 2 *An algorithm $\mathcal{A}$ PAC-learns concept class C by hypothesis class H if for any $c^* \in C$, any distribution D over the instance space, any $\epsilon, \delta > 0$, and for some polynomial p, the following is true. Algorithm $\mathcal{A}$, with access to labeled examples of c^* drawn from distribution $\mathcal{D}$ produces with probability at least $1-\delta$ a hypothesis $h \in H$ with error at most ϵ. In addition,*

1. *$\mathcal{A}$ runs in time polynomial in n and the size of the sample.*
2. *The sample has size $p(1/\epsilon, 1/\delta, n, size(c^*))$.*

The quantity ϵ is usually called the accuracy parameter and δ is called the confidence parameter. A hypothesis with error at most ϵ is often called "ϵ-good."

Note: This definition allows us to make statements such as: "the class of k-term DNF formulas is learnable by the hypothesis class of k-CNF formulas."

Definition 3 *We say that algorithm $\mathcal{A}$ learns class C in the consistency model if given any set of labeled examples S, the algorithm produces a concept $c \in C$ consistent with S if one exists, and outputs "there is no consistent concept" otherwise. The algorithm runs in polynomial time (in the size of S and the size n of the examples).*

We can relate the consistency model with the PAC model as follows.

Theorem 1 *Let $\mathcal{A}$ be an algorithm that learns class C in the consistency model (i.e., it finds a consistent $h \in C$ whenever one exists). Then $\mathcal{A}$ needs only*

$$\frac{1}{\epsilon}\left(\ln|C| \ + \ \ln\frac{1}{\delta}\right)$$

examples to output a hypothesis of error at most ϵ with probability at least $1-\delta$. Therefore, $\mathcal{A}$ is a PAC-learning algorithm for learning C (by C) in the PAC model so long as this quantity is polynomial in $1/\epsilon, 1/\delta, size(c)$, and n.

Note: If we learn C by H, we just need to replace $\ln|C|$ with $\ln|H|$ in the bound. For example, if $\ln|H|$ is polynomial in n (the description length of an example) and if we can find a consistent $h \in H$ in polynomial time, then we have a PAC-learning algorithm for learning the class C.

Note also that this theorem requires C (or H) to be finite. We will discuss in a couple of lecture the case where C (or H) is not finite.

1.1 Examples

Assume each example x is given by n boolean features (variables).

AND functions (monotone conjunctions). We can learn this class in the consistency model by the following method:

1. Throw out any feature that is set to 0 in any positive example. Notice that these cannot possibly be in the target function. Take the AND of all that are left.
2. If the resulting conjunction is also consistent with the negative examples, produce it as output. Otherwise halt with failure.

Since we only threw out features when absolutely necessary, if the conjunction after step 1 is not consistent with the negatives, then no conjunction will be. There are at most 2^n monotone conjunctions, so by Theorem 1 the class of monotone conjunctions is learnable in the PAC model.

Non-monotone conjunctions, disjunctions, k-CNF, k-DNF. What about functions like $x_1\bar{x}_4x_7$? Instead of thinking about this from scratch, we can just perform a reduction to the monotone case. If we define $y_i = \bar{x}_i$ then we can think of the target function as a monotone conjunction over this space of $2n$ variables and use our previous algorithm. k-CNF is the class of Conjunctive Normal Form formulas in which each clause has size at most k. E.g., $x_4 \wedge (x_1 \vee x_2) \wedge (x_2 \vee \bar{x}_3)$ is a 2-CNF. So, the 3-CNF learning problem is like the inverse of the 3-SAT problem: instead of being given a formula and

being asked to come up with a satisfying assignment, we are given assignments (some satisfying and some not) and are asked to come up with a formula. k-DNF is the class of Disjunctive Normal Form formulas in which each term has size at most k. We can learn these too by reduction: e.g., we can think of k-CNFs as conjunctions over a space of $O(n^k)$ variables, one for each possible clause. There are at most $2^{O(n^k)}$ k-CNFs, so for constant k, by Theorem 1 the class of k-CNFs is learnable in the PAC model.

Decision lists. A Decision List is a list of if-then rules: "if ℓ_1 then b_1, else if ℓ_2 then b_2, else if ℓ_3 then b_3, ..., else b_m", where each ℓ_i is a literal (either a variable or its negation) and each $b_i \in \{0, 1\}$. The meaning is: given an example, we look at the first left-hand-side satisfied and output the corresponding right-hand-side. For instance, one possible decision list is the rule: "if $\overline{x}_1$ then positive, else if x_5 then negative, else positive."

If you like, you can think of this as a decision tree with just one long path.

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, $\ln |C| = O(n \log n)$.

An algorithm for learning DLs in the consistency model.

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 c^*, and suppose that some examples still remain in our data set. Let R be the highest rule in c^* 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 c^* to fire, which contradicts our definition of R. Therefore, our algorithm has at least one legal choice (namely, rule R) in step 1.

Since $\ln |C| = O(n \log n)$, by Theorem 1, the class of decision lists is learnable in the PAC model.

2 Learning Linear Separators

Here we can think of examples as being from $\{0,1\}^n$ or from R^n. In the consistency model, the goal is to find a hyperplane $w \cdot x = w_0$ such that all positive examples are on one side and all negative examples are on other. I.e., $w \cdot x > w_0$ for positive x's and $w \cdot x < w_0$ for negative x's. We can solve this using linear programming. The sample complexity results for classes of finite VC-dimension that we will cover later in the course together with known results about linear programming imply that the class of linear separators is learnable in the PAC model. Today we talk about the Perceptron algorithm, an online algorithm for learning linear separators, one of the oldest algorithms used in machine learning (from early 60s).

2.1 The Perceptron Algorithm

For simplicity, we'll use a threshold of 0, so we're looking at learning functions like:

$$w_1x_1 + w_2x_2 + ... + w_nx_n > 0.$$

We can simulate a nonzero threshold with a "dummy" input x_0 that is always 1, so this can be done without loss of generality. The guarantee we'll show for the Perceptron Algorithm is the following:

Theorem 2 *Let $\mathcal{S}$ be a sequence of labeled examples consistent with a linear threshold function $\mathbf{w}^* \cdot \mathbf{x} > 0$, where $\mathbf{w}^*$ is a unit-length vector. Then the number of mistakes M on $\mathcal{S}$ made by the online Perceptron algorithm is at most $(1/\gamma)^2$, where*

$$\gamma = \min_{\mathbf{x} \in \mathcal{S}} \frac{|\mathbf{w}^* \cdot \mathbf{x}|}{||\mathbf{x}||}.$$

(I.e., if we scale examples to have Euclidean length 1, then γ is the minimum distance of any example to the plane $\mathbf{w}^* \cdot \mathbf{x} = 0$.)

The parameter "γ" is often called the "margin" of $\mathbf{w}^*$ (or more formally, the L_2 margin because we are scaling by the L_2 lengths of the target and examples). Another way to view the quantity $\mathbf{w}^* \cdot \mathbf{x}/||\mathbf{x}||$ is that it is the cosine of the angle between $\mathbf{x}$ and $\mathbf{w}^*$, so we will also use $\cos(\mathbf{w}^*, \mathbf{x})$ for it.

The Perceptron Algorithm: Let's automatically scale all examples $\mathbf{x}$ to have Euclidean length 1, since this doesn't affect which side of the plane they are on.

1. Start with the all-zeroes weight vector $\mathbf{w}_1 = \mathbf{0}$, and initialize t to 1.
2. Given example $\mathbf{x}$, predict positive iff $\mathbf{w}_t \cdot \mathbf{x} > 0$.

3. On a mistake, update as follows:

   - Mistake on positive: $\mathbf{w}_{t+1} \leftarrow \mathbf{w}_t + \mathbf{x}$.
   - Mistake on negative: $\mathbf{w}_{t+1} \leftarrow \mathbf{w}_t - \mathbf{x}$.

   $t \leftarrow t + 1$.

So, this seems reasonable. If we make a mistake on a positive $\mathbf{x}$ we get $\mathbf{w}_{t+1} \cdot \mathbf{x} = (\mathbf{w}_t + \mathbf{x}) \cdot \mathbf{x} = \mathbf{w}_t \cdot \mathbf{x} + 1$, and similarly if we make a mistake on a negative $\mathbf{x}$ we have $\mathbf{w}_{t+1} \cdot \mathbf{x} = (\mathbf{w}_t - \mathbf{x}) \cdot \mathbf{x} = \mathbf{w}_t \cdot \mathbf{x} - 1$. So, in both cases we move closer (by 1) to the value we wanted.

Proof of Theorem 2. We are going to look at the following two quantities $\mathbf{w}_t \cdot \mathbf{w}^*$ and $||\mathbf{w}_t||$.

Claim 1: $\mathbf{w}_{t+1} \cdot \mathbf{w}^* \geq \mathbf{w}_t \cdot \mathbf{w}^* + \gamma$. That is, every time we make a mistake, the dot-product of our weight vector with the target increases by at least γ.

> Proof: if $\mathbf{x}$ was a positive example, then we get $\mathbf{w}_{t+1} \cdot \mathbf{w}^* = (\mathbf{w}_t + \mathbf{x}) \cdot \mathbf{w}^* = \mathbf{w}_t \cdot \mathbf{w}^* + \mathbf{x} \cdot \mathbf{w}^* \geq \mathbf{w}_t \cdot \mathbf{w}^* + \gamma$ (by definition of γ). Similarly, if $\mathbf{x}$ was a negative example, we get $(\mathbf{w}_t - \mathbf{x}) \cdot \mathbf{w}^* = \mathbf{w}_t \cdot \mathbf{w}^* - \mathbf{x} \cdot \mathbf{w}^* \geq \mathbf{w}_t \cdot \mathbf{w}^* + \gamma$.

Claim 2: $||\mathbf{w}_{t+1}||^2 \leq ||\mathbf{w}_t||^2 + 1$. That is, every time we make a mistake, the length squared of our weight vector increases by at most 1.

> Proof: if $\mathbf{x}$ was a positive example, we get $||\mathbf{w}_t + \mathbf{x}||^2 = ||\mathbf{w}_t||^2 + 2\mathbf{w}_t \cdot \mathbf{x} + ||\mathbf{x}||^2$. This is less than $||\mathbf{w}_t||^2 + 1$ because $\mathbf{w}_t \cdot \mathbf{x}$ is negative (remember, we made a mistake on $\mathbf{x}$). Same thing (flipping signs) if $\mathbf{x}$ was negative but we predicted positive.

Claim 1 implies that after M mistakes, $\mathbf{w}_{M+1} \cdot \mathbf{w}^* \geq \gamma M$. On the other hand, Claim 2 implies that after M mistakes, $||\mathbf{w}_{M+1}|| \leq \sqrt{M}$. Now, all we need to do is use the fact that $\mathbf{w}_t \cdot \mathbf{w}^* \leq ||\mathbf{w}_t||$, since $\mathbf{w}^*$ is a unit vector. So, this means we must have $\gamma M \leq \sqrt{M}$, and thus $M \leq 1/\gamma^2$. ■

Discussion: In order to use the Perceptron algorithm to find a consistent linear separator given a set S of labeled examples that is linearly separable by margin γ, we do the following. We repeatedly feed the whole set S of labeled examples into the Perceptron algortihm up to $(1/\gamma)^2 + 1$ rounds, until we get to a point where the current hypothesis is consistent with the whole set S. Note that by theorem 2, we are guaranteed to reach such a point. The runnning time is then poly in $|S|$ and $1/\gamma^2$.

In the worst case, γ can be exponentially small in n. On the other hand, if we're lucky and the data is well-separated, γ might even be large compared to $1/n$. This is called the "large margin" case. (In fact, the latter is the more modern spin on things: namely, that in many natural cases, we would hope that there exists a large-margin separator.) In fact, one

nice thing here is that the mistake-bound depends on just a purely geometric quantity: the amount of "wiggle-room" available for a solution and doesn't depend in any direct way on the number of features in the space.

So, if data is separable by a large margin, then the Perceptron is a good algorithm to use.

Guarantee in a distributional setting: In order to get a distributional guarantee we can do the following.[1] Let $M = 1/\gamma^2$. For any ϵ, δ, we draw a sample of size $(M/\epsilon) \cdot \log(M/\delta)$. We then run Perceptron on the data set and look at the sequence of hypotheses produced: h_1, h_2, ... For each i, if h_i is consistent with following $1/\epsilon \cdot \log(M/\delta)$ examples, then we stop and output h_i. We can argue that with probability at least $1 - \delta$, the hypothesis we output has error at most ϵ. This can be shown as follows. If h_i was a bad hypothsis with true error greater than ϵ, then the chance we stopped and output h_i was at most δ/M. So, by union bound, there's at most a δ chance we are fooled by *any* of the hypotheses.

Note that this implies that if the margin over the whole distribution is $1/poly(n)$, the Perceptron algorithm can be used to PAC learn the class of linear separators.

What if there is no perfect separator? What if only *most* of the data is separable by a large margin, or what if $\mathbf{w}^*$ is not perfect? We can see that the thing we need to look at is Claim 1. Claim 1 said that we make "γ amount of progress" on every mistake. Now it's possible there will be mistakes where we make very little progress, or even negative progress. One thing we can do is bound the total number of mistakes we make in terms of the total distance we would have to move the points to make them actually separable by margin γ. Let's call that TD_γ. Then, we get that after M mistakes, $\mathbf{w}_{M+1} \cdot \mathbf{w}^* \geq \gamma M - \text{TD}_\gamma$. So, combining with Claim 2, we get that $\sqrt{M} \geq \gamma M - \text{TD}_\gamma$. We could solve the quadratic, but this implies, for instance, that $M \leq 1/\gamma^2 + (2/\gamma)\text{TD}_\gamma$. The quantity $\frac{1}{\gamma}\text{TD}_\gamma$ is called the total *hinge-loss* of w^*.

So, this is not too bad: we can't necessarily say that we're making only a small multiple of the number of mistakes that $\mathbf{w}^*$ is (in fact, the problem of finding an approximately-optimal separator is NP-hard), but we can say we're doing well in terms of the "total distance" parameter.

[1] This is not the most sample efficient online to PAC reduction, but it is the simplest to think about. We will see a more interesting one later in the course.