COS 511: Theoretical Machine Learning

Lecturer: Rob Schapire
Scribe: Siddhartha Sen

Lecture #2
February 6, 2008

1 The Consistency Model

In the previous class we introduced our first learning model, called the consistency model. To review, we say that a concept class $\mathcal{C}$ is learnable in the consistency model if there is an algorithm A which, when given any set of labeled examples $\langle (x_1, y_1), \ldots, (x_m, y_m) \rangle$, where $x_i \in X$ and $y_i \in \{0, 1\}$, finds a concept $c \in \mathcal{C}$ that is consistent with the examples, meaning that $c(x_i) = y_i$ for all i, or says (correctly) that there is no such concept. Machine learning tries to find algorithms that learn entire classes of concepts, not just single concepts. Since a concept class is assumed to be very large (i.e., exponential in the size of the example set), an algorithm that simply tests all possible concepts is not considered practical.

In what follows, we will look at several examples of concept classes and see how the consistency model applies to each one. Specifically, we will try to devise an algorithm that finds the right concept for a given set of labeled examples.

1.1 Examples from Boolean Logic

Let's look at some examples of concept classes taken from boolean logic and see if they are learnable in the consistency model. Let $X = \{0, 1\}^n$ be a domain consisting of n-bit strings representing the values of n boolean variables, and let $\mathcal{C}$ be the concept class consisting of monotone conjunctions of these variables. (By *monotone* we mean that negations of variables are not allowed.) Examples of boolean variables could be the following:

z_1 = "is a bird"
z_2 = "is a mammal"
z_3 = "lives on land"

and an example of a concept could be the conjunction $z_2 \wedge z_3$, representing the proposition "is a land mammal". Figure 1 shows an example data set. Each bit in each example assigns a value to the corresponding variable in that position. For instance, the first example assigns a value of false to the first variable, true to the second variable, true to the third variable, and so on. As before, examples are labeled positive ("+") or negative ("−"). Given this set of examples, our goal is to devise an algorithm that finds a concept $c \in \mathcal{C}$ that is consistent with the set. One approach is to try all possible conjunctions of variables that appear in the example set, but this algorithm must test exponentially many conjunctions, making it impractical. Another algorithm suggested in class does the following:

1. Divide the labeled examples into positive and negative examples. (This has already been done in Figure 1.)

2. Look at the positive examples and find all variables that are true (have a value of '1') in every positive example. Let c be the conjunction of these variables.

01101	+
11011	+
11001	+
00101	−
11000	−

Figure 1: A set of labeled examples from the domain $X = \{0, 1\}^n$.

3. Test c for consistency with the negative examples (c should evaluate to false for every negative example). If any negative example is inconsistent with c, return `NULL`; otherwise, return c.

If we run this algorithm on the example set in Figure 1, we get $c = z_2 \wedge z_5$, where z_2 and z_5 represent the second and fifth variables, respectively. A simple check verifies that c is consistent with both the positive and negative examples. Evidently, the algorithm is correct, but let's try to prove this more formally.

Claim 1. *Given a set of labeled examples, the above algorithm returns a non-null concept $c \in \mathcal{C}$ that is consistent with examples if and only if such a concept exists.*

Proof sketch. In the forward direction, we want to show that if the algorithm returns a non-null concept c, then c is consistent with the set of labeled examples. This is straightforward and can be shown by construction. In step 2, the algorithm only includes variables that are true in all positive examples; hence, their conjunction is also true. In step 3, the algorithm checks c against each negative example for consistency; if any example is inconsistent, the algorithm returns `NULL`. Therefore, if the algorithm returns c, then c is consistent with all labeled examples.

In the reverse direction, we want to show that if there is a concept in $\mathcal{C}$ that is consistent with the labeled examples, then the algorithm finds this concept. We consider the positive and negative examples separately. Observe that any conjunction that is consistent with the positive examples must be a sub-expression of c from step 2. To see this, consider what happens when we include a variable z_f that is not in c. The new concept $c' = c \wedge z_f$ must be false for at least one positive example, since z_f is not true for all positive examples (if it were, it would already be in c). Therefore z_f cannot be a part of c. Now, observe that if any $c_n \subseteq c$ is consistent with the negative examples, then c is also consistent with the negative examples. This is true because c_n must evaluate to false on all negative examples (by definition), and therefore $c = c_n \wedge (c \setminus c_n)$ is also false. This shows that c is necessarily consistent with all negative examples as well. Moreover, if c is not consistent with all negative examples, then no sub-expression of c is, and since any consistent concept is a subset of c, it follows that there is no consistent concept, and the algorithm correctly returns `NULL`.

□

Note that our algorithm does not necessarily find the smallest concept (as measured by the number of variables) that is consistent with the labeled examples.

There are other examples of concept classes from boolean logic that can be studied in the context of the consistency model. We consider them briefly. In all of these examples, $X = \{0, 1\}^n$.

Let $\mathcal{C}$ be the set of monotone disjunctions. An example of a concept in this class is $c = z_2 \vee z_5 \vee z_7$. Observe that any monotone disjunction can be expressed as a monotone conjunction if we negate each variable and apply De Morgan's law. That is, $c = z_2 \vee z_5 \vee z_7 = \overline{\overline{z_2} \wedge \overline{z_5} \wedge \overline{z_7}}$. If we replace each z_i with $z_i' = \overline{z_i}$ and flip all positive labels in the data set to negative and vice versa, we can use our learning algorithm for monotone conjunctions on the modified class. To obtain a concept from the original class, simply negate each variable in c and replace all conjunctions with disjunctions.

Let $\mathcal{C}$ be the set of conjunctions (not necessarily monotone). As in the previous example, we can use a separate variable $z_i' = \overline{z_i}$ to represent the negation of each variable z_i. This modified class is now monotone and we can again use our learning algorithm for monotone conjunctions. To obtain a concept from the original class, simply replace each z_i' with $\overline{z_i}$ in c.

Let $\mathcal{C} = k\text{-}CNF$, the set of conjunctive normal form formulas with k or less literals per clause. Concepts in this class take the form:

$$\underbrace{(\ldots \vee \ldots \vee \ldots)}_{k \text{ literals}} \wedge \underbrace{(\ldots \vee \ldots \vee \ldots)}_{k \text{ literals}} \ldots$$

If we assign a new variable z_i for every disjunction of k (or less) variables, then we can replace each clause with a single variable to yield a conjunction, which we already know how to learn from the previous example. There are $O(n^k)$ such new variables created, and since k is assumed to be constant relative to n, this is a tractable solution.

Let $\mathcal{C} = k$-term DNF, the set of disjunctive normal form formulas with k terms. Concepts in this class take the form:

$$\underbrace{(\ldots \wedge \ldots \wedge \ldots) \vee (\ldots \wedge \ldots \wedge \ldots) \ldots}_{k \text{ terms}}$$

Since the number of variables in each term is unbounded, this problem turns out to be NP-complete even for $k = 2$. However, if we expand the expression by distributing the disjunctions (i.e. multiplying), this yields a $k\text{-}CNF$ expression, which we already know how to learn efficiently. (In general, k-term $DNF \subseteq k\text{-}CNF$.) The problem is that solutions to the latter class do not easily map to solutions in the former class. We are thus faced with a somewhat unsettling situation where a concept class is learnable in our learning model, but a subclass of this class is not.

Another unsettling situation arises when we consider the general class $\mathcal{C} = DNF$. Given a set of labeled examples, a learning algorithm can trivially construct a DNF expression that enumerates all positive examples, since we are not given a restriction on the number of clauses allowed (unlike in the $k\text{-}DNF$ case). For instance, for the data set in Figure 1, we could construct the following DNF:

$$c = (\overline{z_1} \wedge z_2 \wedge z_3 \wedge \overline{z_4} \wedge z_5) \vee (z_1 \wedge z_2 \wedge \overline{z_3} \wedge z_4 \wedge z_5) \vee (z_1 \wedge z_2 \wedge \overline{z_3} \wedge \overline{z_4} \wedge z_5)$$

This concept is consistent with all labeled examples (we can verify this in polynomial time) and is linear in the total size of the examples. It is easy to prove that the algorithm

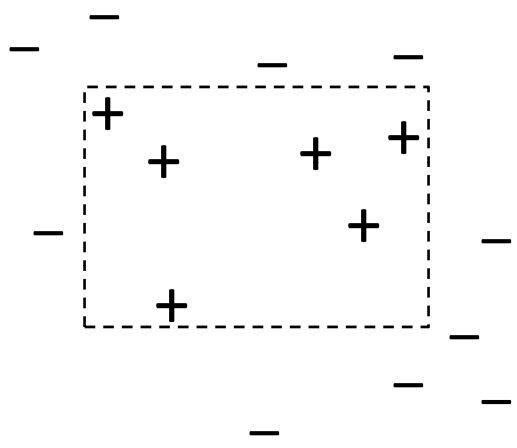

Figure 2: A set of labeled examples from $X = \mathbb{R}^2$. Here, $\mathcal{C}$ is the set of axis-parallel rectangles.

finds a consistent concept if and only if one exists. Therefore, the class $\mathcal{C} = DNF$ is learnable in the consistency model via the most trivial solution, which seems problematic. Clearly, there is something missing in our learning model if we consider such solutions to be satisfactory.

We are beginning to see some problems with the consistency model, which we discuss in the next section.

1.2 Examples from Geometry

Let's look at some examples of concept classes taken from geometry and see if they are learnable in the consistency model.

Let $X = \mathbb{R}^2$ be the set of points in 2-dimensional space. Let $\mathcal{C}$ be the set of axis-parallel rectangles, so that points that fall within the bounds of these rectangles are positive examples, and points that fall outside are negative. Figure 2 shows an example data set and a corresponding concept that is consistent with the data. What is an appropriate learning algorithm for this class? A simple approach is to find the minimum and maximum positive example along each axis, which defines a bounding box. Then, we can apply an argument similar to the one for our algorithm for monotone conjunctions to show that any rectangle consistent with the dataset must be a subspace of this bounding box. As before, we must test any rectangle we propose against the negative examples to make sure it is consistent with them.

Let $X = \mathbb{R}^n$ be the set of points in n-dimensional space. Let $\mathcal{C}$ be the set of half-spaces, so that points that lie above some hyperplane are positive examples, and points that fall below are negative. Figure 3 shows a sample data set for $n = 2$, and also shows a concept (half-plane) that is consistent with the data. What is an appropriate learning algorithm for this class? One approach suggested in class is to draw lines between positive and negative examples until one of these lines partitions the data set consistently. While this is efficient for $n = 2$, it is in general not efficient for higher dimensions, since all possible sets of n examples may need to be tested before a consistent one is found (which is exponential in n). Another approach is to find the convex hull of the positive examples and the convex hull of the negative examples, and then find a half-space that separates these hulls. Finding

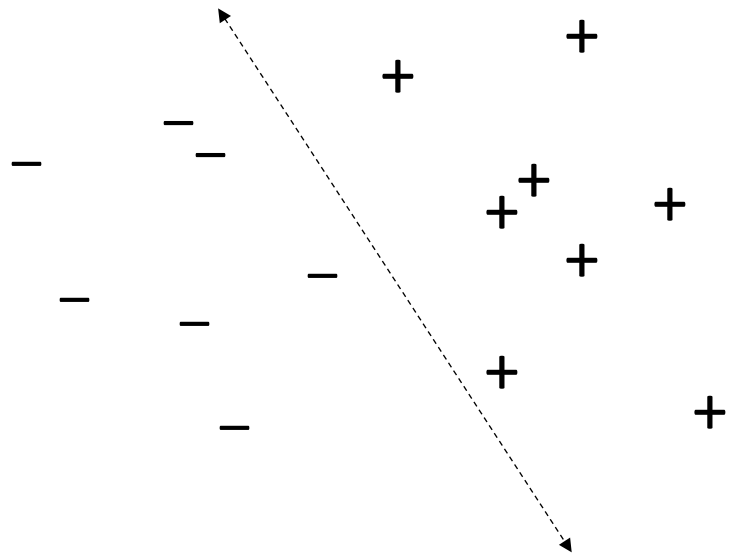

Figure 3: A set of labeled examples from $X = \mathbb{R}^2$. Here, $\mathcal{C}$ is the set of half-spaces.

a convex hull is difficult to do, however, so it is not clear that this solution will work either.

A more promising approach is to express the half-space as an n-dimensional normalized (unit) vector $\mathbf{w}$, which we can then compare to each labeled example. Specifically, for a given example $\mathbf{x}_i$, the value of $\mathbf{w} \cdot \mathbf{x}_i$ will be greater than some threshold b for all positive examples, and less than this threshold for all negative examples (assuming a consistent concept exists). Thus, we can formulate our learning problem as follows: find a vector $\mathbf{w}$ and a constant b such that:

$$\mathbf{w} \cdot \mathbf{x}_i > b \text{ if } y_i = 1$$
$$\mathbf{w} \cdot \mathbf{x}_i < b \text{ if } y_i = 0$$

where y_i is the label corresponding to example x_i in the data set ('1' means it's a '+' and '0' means its a '-'). This problem can be solved using linear programming, for which there are known polynomial time solutions.

2 Problems with the Consistency Model

In the previous section, we looked at different classes of learning problems and tried to devise efficient algorithms for learning them in the consistency model. While doing this, we ran into a few problems and deficiencies in the consistency model, which we summarize here.

First, the consistency model does not say anything about *generalization*, or how well a learning algorithm performs on data sets it has not seen. That is, even if we find an algorithm that is consistent with a training data set, it is important to evaluate the algorithm's performance on outside test sets. The consistency model does not provide a framework for measuring the algorithm's performance on these test sets.

Second, it seems too easy to learn the class DNF in the consistency model. The most trivial algorithm we could have come up with met the criteria for learnability in this model.

Since this algorithm only enumerates the positive examples in the training set, one would expect it to perform poorly on any test set, or to require a large number of training examples in order to do reasonably well on a given test set. Neither of these properties are adequately captured by the requirements of the consistency model.

Third, we came across an example of a class that is learnable in the consistency model but whose subclass is not. This was the case for $\mathcal{C} = k\text{-}CNF$, which is learnable, and k-term $DNF \subseteq k\text{-}CNF$, which is not. It seems contradictory to be able to learn a class but not a proper subset of this class.

Finally, the consistency model does not allow for any noise in the data. That is, if our learning algorithm produces a concept that is inconsistent with even one labeled example, this concept is discarded. In practice, there may be incorrect examples in the data set or labels with margins of error (e.g., if the labels represent weather predictions). A good learning model should accommodate this kind of noise, otherwise it cannot be used to solve real-world problems.

3 PAC: A More Realistic Learning Model

Having seen the problems inherent to the consistency model, let us work towards a more realistic learning model. Recall that a learning algorithm is supposed to produce a prediction rule (or concept) that can be used to classify examples outside the training set (what we call test sets). One of the problems with the consistency model is that it does not provide a means for quantifying the generalizability of such a concept; we address this limitation here. First, we make the following assumptions about examples from the training and test sets:

- the training and test examples come from the same target distribution D;
- all labels are generated by an unknown target concept c from a known class $\mathcal{C}$.

These assumptions give us a basis for evaluating the generalizability of a concept. The intuition is that given a sufficient number of labeled training examples, a good learning algorithm should produce a concept h (the hypothesis) that does reasonably well on future examples drawn from the same distribution. Specifically, we measure the error of h relative to the target concept c as follows:

$$\text{err}_D(h) = \Pr_{x \sim D}[h(x) \neq c(x)]$$

Our goal is to find an h such that the probability that $\text{err}_D(h)$ is small is very high. If we can do this, then we can say that h is *probably approximately correct*.

3.1 PAC Model Definition

PAC stands for "probably approximately correct", capturing the notion above. We state what it means for a class to be learnable in the PAC model; in the next class, we will look at specific examples.

We say that a concept class $\mathcal{C}$ is PAC learnable by $\mathcal{H}$ if there is an algorithm A such that for all $c \in \mathcal{C}$ and for all target distributions D, and for all $\epsilon > 0$ and $\delta > 0$, A takes $m = poly(1/\epsilon, 1/\delta, \ldots)$ examples of the form $S = \langle (x_1, c(x_1)), \ldots, (x_m, c(x_m)) \rangle$ where each $x_i \sim D$, and produces a hypothesis $h \in \mathcal{H}$ such that $\Pr[\text{err}_D(h) \leq \epsilon] \geq 1 - \delta$. We say that

h is "ϵ-good" if this holds. Here, ϵ and δ are tunable parameters that represent the desired error and confidence thresholds, respectively. The number of examples m is polynomial in $1/\epsilon$, $1/\delta$, and some other terms we will see later on.

We will continue our discussion of the PAC model, and analyze the learnability of concept classes in this model, in the next class.

4 Probability Review

This course assumes some basic knowledge of probability theory. Below, we list some of the concepts you should be familiar with but do not provide details or explanations. If you are unfamiliar with any of these concepts or need to review them, the course website has some pointers to appropriate review materials (for example, see Appendix C.2 and C.3 in Introduction to Algorithms by Cormen, Leiserson, Rivest, and Stein).

- event
- random variable (e.g., X)
- distribution ($\Pr[X = x]$)
- joint distribution
- expectation ($\mathrm{E}[X]$; $\mathrm{E}[c] = c$ for any constant c; $\mathrm{E}[X + Y] = \mathrm{E}[X] + \mathrm{E}[Y]$, linearity of expectation)
- conditional probability ($\Pr[A|B]$, where A and B are events)
- independence ($\Pr[A \wedge B] = \Pr[A] \cdot \Pr[B]$ if A and B are independent)
- union bound ($\Pr[A \vee B] \leq \Pr[A] + \Pr[B]$, regardless of whether A and B are independent; this can be replaced with equality if A and B are disjoint)
- marginalization ($\Pr[A] = \mathrm{E}_X[\Pr[A|X]] = \sum_x \Pr[X = x] \cdot \Pr[A|(X = x)]$)