CS6781 - Theoretical Foundations of Machine Learning

Lecture 2: The Consistency Model

January 23, 2020

Lecturer: Nika Haghtalab *Readings: Chp 2, UML*

As a motivating example, consider going to the Ithaca Apple Harvest Festival. You want to identify delicious apples from the bad ones. Let's make a simplifying assumption that only three features of an apple affect its deliciousness: color (green/red), firmness (soft/crunchy), and size (small/medium/large). In the past, you have had a several apples from the Apple Fest and you have diligently recorded the features of every apple you ate and their level of tastiness. Your goal is to use this historical record to learn to identify tasty apples from the non-tasty ones. That is, given a new apple that you haven't yet tasted, predict whether this apple is tasty.

For today's lecture, we make a simplifying assumption that there is an unknown mapping from apples to labels, denoted by $c : \{green, red\} \times \{soft, crunchy\} \times \{small, medium, large\} \to \{\text{tasty}, \text{not tasty}\}$, that perfectly determines the deliciousness of an apple. Such a mapping is called a *concept*. A collection of concepts is called a *concept class*. We assume that the concept c belongs to some known concept class $\mathcal{C}$ that is pre-determined. Our goal is to learn c or a close approximation of it, so that we can near-perfectly identify all delicious apples in the Ithaca Apple Harvest Fest.

How do you learn c? This is what we discuss in this lecture.

1 Formal Model

Let us formally define notations that will be used in this lecture and many of the following lectures.

- **Domain (Instance space):** An arbitrary set $\mathcal{X}$ that includes all possible instances, e.g., apples, that the learner may wish to label. An instance is typically described by a vector of values, representing the relevant feature values. For example, and apple can be described by a feature vector $(green, crunchy, medium)$. In this case, the domain can be the set of all possible feature vectors, i.e., $\mathcal{X} := \{green, red\} \times \{soft, crunchy\} \times \{small, medium, large\}$. An $x \in \mathcal{X}$ is called an *instance*.

- **Labels:** A set $\mathcal{Y}$ that includes all possible labels or predictions for a single instance. In the apple example, we have $\mathcal{Y} = \{\text{tasty}, \text{not tasty}\}$. For simplicity, this course works with 2-element label sets, which we usually refer to as $\{0, 1\}$, $\{-1, 1\}$, $\{false, true\}$, etc. For ease of presentation, in the apple example we refer to tasty, 1, $true$ interchangeably.

- **Labeled instance:** An instance-label pair $(x, y) \in \mathcal{X} \times \mathcal{Y}$ is called a labeled instance.

- **Concept:** A *concept* (later on will also be called a classifier or predictor or hypothesis) is a function $c : \mathcal{X} \to \mathcal{Y}$. For example, the concept $(color = red) \vee (size \neq small)$ assigns to any apple that is red or is not small, the label $true$.

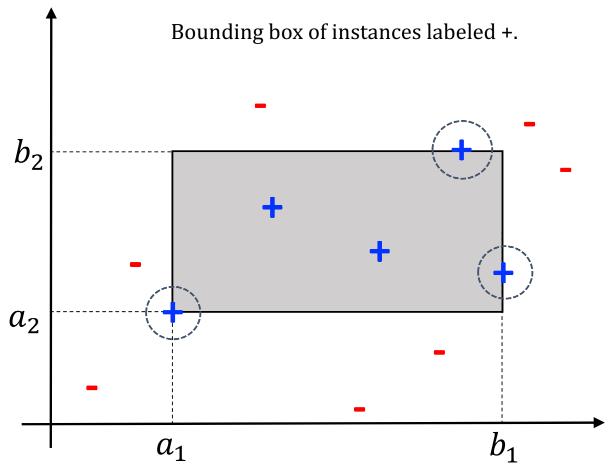

Figure 1: An axis-aligned rectangle in two dimensions.

- **Concept class:** A concept class $\mathcal{C}$ is a pre-determined set of concepts.

2 The Consistency Model

We start our study of *learnability* with the*consistency* model. While this may not be a very realistic model of learning, it's a great place for demonstrating ideas that will come up again later.

We say that a concept $c \in \mathcal{C}$ is *consistent* with a set of samples $\{(x_1, y_1), \ldots, (x_m, y_m)\}$, if for all $i \in [m]$, $c(x_i) = y_i$. We say that a concept class $\mathcal{C}$ is *learnable* in the *consistency model* if there is an algorithm $\mathcal{A}$ such that, for any set of labeled instances $S = \{(x_1, y_1), \ldots, (x_m, y_m)\}$, $\mathcal{A}(S) = c$ for some $c \in \mathcal{C}$ that is consistent with the examples, or $\mathcal{A}(S) =$ "no such concept exists" if no such concept $c \in \mathcal{C}$ exists.

We are especially interested in algorithms that are *computationally efficient* and can learn in the consistency model. Let's consider a few examples of such algorithms.

2.1 Geometrical Examples

Axis-aligned rectangles. In this example, we consider $\mathcal{X} = \mathbb{R}^2$ and $\mathcal{Y} = \{+, -\}$. An axis-aligned rectangle is a concept that assigns $+$ to instance that are within some rectangle and $-$ to those outside. More formally, each concept $c \in \mathcal{C}$ is defined by four parameters $a_1, b_1, a_2, b_2 \in \mathbb{R}$ and

$$c(\mathbf{x}) = \begin{cases} + & \text{if for } i \in \{1,2\}, a_i \leq x_i \leq b_i \\ - & \text{otherwise} \end{cases}$$

How would you design an algorithms $\mathcal{A}$ that runs efficiently, in the size of the input set S and learns $\mathcal{C}$ in the consistency model? A simple solution is to find the minimum and maximum instances labeled $+$ along each of the axes. Then, consider the axis-aligned rectangle whose boundaries are defined by these examples (See Figure 1). Note that, this is the most *conservative* concept in $\mathcal{C}$ that is consistent with all $(\mathbf{x}_i, +) \in S$. That is, the positive region of any other axis-aligned rectangle $c' \in \mathcal{C}$ that is also consistent with all $(\mathbf{x}_i, +) \in S$ includes the positive region of c. All that is left is to check if c is also consistent with all $(\mathbf{x}_i, -) \in S$. If it is consistent then $\mathcal{A}(S) = c$. Otherwise, no other concept can be consistent with the data in which case $\mathcal{A}(S)$ states that no consistent axis-aligned rectangle exists.

Note that such an algorithm take $O(|S|)$ to find the minimum and maximum instances labeled $+$ along each axis and to form the bonding box. It takes an additional $O(|S|)$ runtime to check that the concept defined by the bounding box is consistent with the rest of the data.

Linear Thresholds In this example, we consider $\mathcal{X} = \mathbb{R}^n$ and $\mathcal{Y} = \{+, -\}$. A homogeneous linear threshold (or a halfspace) is a concept that assigns $+$ to one side of a linear hyperplane that passes through the origin and $-$ to the other side. Formally, each concept corresponds to a vector $\mathbf{w} \in \mathbb{R}^n$ such that

$$c(\mathbf{x}) = \begin{cases} + & \text{if } \mathbf{w} \cdot \mathbf{x} \geq 0 \\ - & \text{otherwise} \end{cases}$$

How would you design an algorithms $\mathcal{A}$ that runs efficiently, in the size of the input set S and n, and learns $\mathcal{C}$ in the consistency model?

Note that a concept c defined by $\mathbf{w}$ is consistent with the data if,

$$\begin{aligned} \forall (\mathbf{x}_i, +) \in S \quad & \mathbf{w} \cdot \mathbf{x}_i \geq 0 \text{ and} \\ \forall (\mathbf{x}_i, -) \in S \quad & \mathbf{w} \cdot \mathbf{x}_i < 0 \end{aligned} \tag{1}$$

We are now very close to having a polynomial time algorithm that can check whether there is a $\mathbf{w}$ that satisfies the above constraints. Recall that Linear Programs (LP) are problems that can be expressed as maximizing a linear function subject to linear (non-strict) inequalities. Any optimization problem that can be expressed as a linear program can be solved efficiently in the number of constraints and the dimension of the space.

Is Equation (1) already in an LP form? No. That is because Equation (1) uses strict inequalities in the constraints. But, we can rewrite this optimization so that the inequalities are in the non-strict form. To make things simple, let's assume that if there is a concept that is consistent with S, then there is a concept $\mathbf{w}^*$ that is not only consistent with respect to S, but also has some wiggle room. That is, no instance is exactly on this hyperplane.[1] Let this wiggle room be $\gamma = \min_i y_i(\mathbf{w}^* \cdot \mathbf{x}_i) > 0$. Then, note that for all i, $(\mathbf{w}^* \cdot \mathbf{x}_i) y_i \geq \gamma$. Therefore, $\forall (\mathbf{x}_i, y_i) \in S, y_i \left(\frac{\mathbf{w}^*}{\gamma}\right) \cdot \mathbf{x}_i \geq 1$. This is exactly the type of non-strict inequality we were hoping for. Now, we can use an LP to see if there is a $\mathbf{w}$ that satisfy the constraints

$$\forall (\mathbf{x}_i, y_i) \in S, \quad y_i(\mathbf{w} \cdot \mathbf{x}_i) \geq 1$$

If the LP returns such a $\mathbf{w}$, then the concept defined by $\mathbf{w}$ is consistent with S. If not, no consistent linear threshold exists.

[1]Think whether this assumption is without loss of generality.

2.2 Examples from Boolean Logic

In this section, we work with boolean logic, where 0 is equivalent to $False$ and 1 equivalent to $True$. We let $\mathcal{X} = \{0,1\}^n$ and for each $\mathbf{x} \in \mathcal{X}$, we interpret bit x_i as the boolean value of the ith variable.

Monotone Conjunctions Here, $\mathcal{C}$ is a class of all boolean functions that are an AND of some of the variables. Monotone means that the variables will not appear negated in the concept. For example, a concept $c(\mathbf{x}) = x_1 \wedge x_3$ is a monotone conjunction, but $c(\mathbf{x}) = x_1 \wedge \bar{x}_3$ is not monotone.

First, can you see whether the following set of labeled instances, appearing as a table, admits a consistent monotone conjunction?

instances	*labels*
1 0 1 0 1	+
0 1 1 0 0	-
1 1 1 0 0	+
1 0 1 1 0	+
0 1 1 1 1	-
1 0 0 0 0	-

It is not hard to see that $c(\mathbf{x}) = x_1 \wedge x_3$ is consistent with these labeled instances. How about a general algorithm that can learn the class of monotone conjunctions in the consistency model? To answer this question, we take a similar approach as we did for the axis-aligned examples. Let's consider only the positive instances. What is the most *conservative* monotone conjunction. That is, what is $P \subseteq [n]$ such that if $c^*(\mathbf{x}) = \bigwedge_{i \in P} x_i$ is consistent with S, then any other monotone conjunction that is consistent with the positive instances corresponds to $c(\mathbf{x}) = \bigwedge_{i \in P'} x_i$ for $P' \subseteq P$?

We can throw out any variable that is set to 0 in any of the positive instances. This is due to the fact that if any of these variables were used in the conjunctions, the conjunction would have resulted in a $-$. Letting P be the set of all remaining variables results in a concept that is consistent with all positive examples. If this concept is also consistent with the negative examples we return this concept. Otherwise, we say that no consistent concept exists.

Why is this algorithm correct? Of course, if the algorithm returns a concept, we know that it was consistent with both positive and negative instances. But, how do we know that if the algorithm says that there is no consistent concept, then there really isn't one?

The reasoning here is very similar to our reasoning for the axis-aligned rectangle example. Note that any conjunction $c(\mathbf{x}) = \bigwedge_{i \in P'} x_i$ that is also consistent with all positive instances must have that $P' \subseteq P$, this is because all other features in $[n] \setminus P$ were disqualified since they would have led to at least one positive instance being mislabeled. So, the concept the algorithm tries to build is the most conservative concept in terms of how it labels the positive points. Therefore, if this concept is not consistent with negative instances, then there is no concept that is consistent with all labeled instance.

Note that this algorithm takes $O(n|S|)$ time to form the most conservative consistent concept on positive instances and another $O(n|S|)$ time to check for the consistency with respect to negative instances.

2.3 Monotone Disjunctions

$\mathcal{C}$ is a class of all boolean functions that are an OR of some of the variables. Monotone here means that the variables will not appear negated in the concept. For example, a concept $c(\mathbf{x}) = x_1 \vee x_3$ is a monotone disjunction, but $c(\mathbf{x}) = x_1 \vee \bar{x}_3$ is not monotone.

How do we learn the class of monotone disjunction in the consistency model? Note that any monotone disjunction is equivalent to the negation of a monotone conjunction with negated variables. That is, using De Morgan's law, $x_1 \vee x_3 = \overline{\overline{x_1} \wedge \overline{x_3}}$. Therefore, we can set variables $z_i = \overline{x_i}$ and flip the labels of all given instances. Then, we can use the same algorithm use for monotone conjunctions to learn a the concept class of monotone disjunctions in the consistency model.

3 Consistency Model and Generalization

At a high level, learning in the consistency model is really about optimization on observed labeled instances. But it is not necessary clear whether the concept that is learned in the consistency model is a good predictor for instances that the algorithm has not encountered yet. As a thought exercise and while ignoring the need for computationally efficient algorithms, consider the setting where $\mathcal{C}$ includes all boolean functions on n bit. Then one can learn (inefficiently though) in the consistency model, by having $\mathcal{A}(S)$ say no consistent concept exists if S includes an $(\mathbf{x}, y)$ and $(\mathbf{x}, \bar{y})$, and otherwise memorize the instances and their respective labels through a disjunctive normal form. While this algorithm does learn in the consistency model, the learning seems ineffective in a way. For example, any instance that hasn't appeared in S will be labeled as negative. This makes the concept class especially brittle on unseen instances.

We see next time how we can change the consistency model to address the above problem.