CS269: Machine Learning Theory
Lecture 1: A Gentle Introduction to Learning Theory
September 27, 2010

Lecturer: Jennifer Wortman Vaughan
Scribe: Jennifer Wortman Vaughan

1 What is Machine Learning?

Machine learning studies automatic techniques for making accurate predictions or choosing useful actions based on past experience or observations. In machine learning, there is typically an emphasis on *efficient* techniques. Machine learning algorithms must be tractable in terms of time and space, but should also not require unreasonably large quantities of data.

Machine learning has been applied successfully in a wide variety of domains, and impacts us all on a day-to-day basis. For example, websites like Netflix offer us movie recommendations based on our own ratings of movies we've seen and ratings from other movie viewers who are like us. When we search for a phrase like "Los Angeles restaurants" on Google, machine learning techniques are used not only to find the relevant search results, but also to determine which ads we are most likely to click. Other applications of machine learning include autonomous helicopter flight,[1] medical diagnosis, handwritten character recognition, customer segmentation for marketing, document segmentation (e.g., for sorting news articles), spam filtering, weather prediction and climate tracking, gene prediction, and face recognition.

2 Some Typical Learning Settings

Let's look at one of these examples in more detail. Suppose we would like to classify email messages as spam or not spam. We might start with a large set of email messages, each marked as spam or not by hand by the email's recipient. Based on this data, we would like a way to automatically determine whether a new message we see is spam.

The first thing we need to do to turn this problem into something that can be solved algorithmically is to figure out how to represent our data (in this case, the email messages). This is typically done by representing each example (that is, each individual email) as a vector of *features*. We might have one feature that is 1 if the phrase "CS269" appears in the email and 0 otherwise, another feature that is 1 if the sender is in our address book and 0 otherwise, and so on. Along with these feature vectors, we have a binary *label* for each email which tells us whether or not the email is spam.

The next thing we need to do is narrow our search to some reasonable set of prediction rules. For example, we might try to find a rule that predicts labels according to the value of a *disjunction* of some subset of the features (predict spam if and only if the sender is not known or "Viagra" appears in the message) or one that predicts labels according to a *threshold* function (predict spam if and only if "Jenn" in email + "CS269" in email + sender known < 2).

[1] See `http://heli.stanford.edu/` for some cool videos!

Finally, we need to design an algorithm to choose a prediction rule from the specified set based on the particular data we've seen (called the *training data*). Our hope is that the rule that we choose will perform well on data we haven't seen yet too.

Of course, one might argue that in the real world, email messages don't split nicely into "training data" and "test data" sets. New email messages continue to arrive every day. Ideally we would like a learning algorithm that can predict the labels of new messages as they arrive, and learn from its mistakes.

In the *online learning* setting (as opposed to the *batch learning* setting described above), examples arrive one at a time. The learning algorithm must predict a label for each example as it arrives, and only later learns the true label of the example. The goal is to design an algorithm that updates its prediction rule over time while making as few mistakes as possible. We'll talk about both batch learning and online learning problems in this class.

Some other learning settings you might come across include unsupervised learning (in which there are no explicit labels, as in clustering), semi-supervised learning (in which there are labels for some, but not all, examples), active learning (in which the learning algorithm may choose which examples to receive labels for), and reinforcement learning. We will not have time to cover these learning settings in this course, but you are free to explore them in your final project.

3 What is Learning Theory?

The goal of learning *theory* is to develop and analyze formal models that help us understand a variety of important issues that come up in machine learning. Learning theory helps us understand what concepts (or prediction rules) we can hope to learn efficiently and how much data is sufficient to learn them, what types of guarantees we might hope to achieve (e.g., in terms of error bounds or complexity bounds), and why particular algorithms may or may not perform well under various conditions. Understanding these important issues helps generate intuition that is useful for practical algorithm design.

The types of questions we might ask in learning theory include:

- What is the computational complexity of a particular learning problem? What is the complexity in terms of data required?
- What are the intrinsic properties of a learning problem that make it harder or easier to solve?
- How much prior information or domain knowledge do we need to learn effectively?
- Are simpler hypotheses always better? Why?

In this course, we will focus on these types of questions. The course will be broken into four main parts: classification and the *probably approximately correct* (PAC) model of learning, online learning in adversarial settings, a closer look at some of learning theory's practical successes (such as SVMs and boosting), and a brief look at new research directions.

4 The Consistency Model

In order to make precise, mathematical statements about machine learning problems, we need to develop *models* in which to study them. A good learning model should be realistic enough to capture all of the fundamental issues that arise in a particular learning setting, but simultaneously must be simple enough to

analyze. It should answer several important questions: What is it that we are trying to learn? What kind of data is available to the learner? In what way is the data presented to the learner (online, actively, etc.)? What type of feedback does the learner receive, if any? What is the learner's goal?

A good learning model should also be robust to minor variations in its definition. Intuitively, it is unlikely that a model can provide any real fundamental insight if small changes in the definitions result in drastic changes in the results.

To get our feet wet, let's start by looking at a very simple learning model, the *consistency model*.

Definition 1. *We say that algorithm $\mathcal{A}$ learns concept class $\mathcal{C}$ in the consistency model if given any set of labeled examples S, $\mathcal{A}$ produces a concept $c \in \mathcal{C}$ consistent with S if one exists and states that none exists otherwise.*

Definition 2. *We say that a class $\mathcal{C}$ is learnable in the consistency model if there exists an efficient algorithm $\mathcal{A}$ that learns $\mathcal{C}$ in the consistency model.*

Here "efficient" is in terms of the size of the set of examples S and the size of each example in S. Let's look at a couple of examples of concept classes that are learnable in the consistency model.

4.1 Monotone Conjunctions

Consider the following data set, in which each example represents a song and each label (displayed in the final column) tells us whether a particular person liked the song or not.

Guitar	Fast beat	Male singer	Acoustic	New	**Liked**
1	0	0	1	1	**1**
1	1	1	0	0	**0**
0	1	1	0	1	**0**
1	0	1	1	0	**1**
1	0	0	0	1	**0**

There are many different concept classes we could use to learn this data. Consider first the class of monotone conjunctions of the features. A *monotone* conjunction is a conjunction in which negations of the variables are not allowed. (Guitar $\wedge$ Fast Beat $\wedge$ Acoustic) is a valid monotone conjunction, but ($\neg$Guitar $\wedge$ Fast Beat) is not.

We will see that the class of monotone conjunctions is learnable in the consistency model using the following algorithm:

1. Identify the set of variables (or features) that are true (1) in every positive example.
2. Let h be the conjunction of these variables.
3. If h is consistent with every negative example (that is, if h predicts 0 for every negative example), then output h; otherwise, output "none."

Let's first check what would happen if we ran the algorithm on the data set of songs above. There are only two positive examples. Both have three positive features (Guitar, Acoustic, New, and Guitar, Male Singer, Acoustic, respectively). The set of features that are positive in both are Guitar and Acoustic. We therefore let h be the conjunction Guitar $\wedge$ Acoustic. This hypothesis is consistent with the negative examples, so we can output h as our consistent hypothesis.

We can quickly sketch a proof that this algorithm outputs a correct answer. First, consider the case in which the algorithm outputs some function h. This case is relatively straight-forward. We have chosen h to be consistent with the positive examples by design, and have checked h against each negative example, so h must be consistent.

What about the case in which the algorithm states that no consistent hypothesis exists? If this happens, it means that we found one particular h that was consistent with the positive examples, but that there was at least one negative example $\vec{x}$ for which h predicted the wrong label, i.e., for which $h(\vec{x}) = 1$. (We use vector notation for each example $\vec{x}$ here because these examples are vectors of features.) Suppose there was a different hypothesis h' that was consistent with all of the examples. We can first argue that the set of variables in the conjunction h' must be a *subset* of the variables in the conjunction h. This is because h already includes all of the variables that were true in every positive example, so if h' contained a variable not in h, h' would be false on some positive example and therefore not consistent. But this implies that since $h(\vec{x}) = 1$, $h'(\vec{x}) = 1$ too. Therefore h' cannot be consistent with all of the negative examples, and no consistent hypothesis exists.

This leads to the following theorem.

Theorem 1. *The class of monotone conjunctions is learnable in the consistency model.*

4.2 DNFs

Another concept class we might wish to consider is the class of Boolean functions in disjunctive normal form, often referred to as *DNFs*. A DNF is an "or of ands" of variables. That is, it is a disjunction of some set of terms, where each term is a conjunction of some set of feature variables or their negations. Some examples of DNFs that are consistent with the data set above include (Guitar $\wedge$ Acoustic), or alternately (Acoustic $\wedge$ $\neg$ Male Singer) $\vee$ ($\neg$ Fast Beat $\wedge$ $\neg$ New).

It turns out that there is a trivial way to learn DNFs in the consistency model. Essentially, we just memorize the positive examples. That is, we create a DNF with one term corresponding to every positive example such that the term corresponding to a particular positive example completely specifies the values of each variable in that example. If this function is consistent with the negative examples, we output it; otherwise we output "none."

4.3 What's Wrong with This Model?

There is something a little unsatisfying about the DNF learning algorithm that we just described. The algorithm essentially just memorizes the training data. This is bad for a couple of reasons. First, the DNF that we output could potentially be unnecessarily large, since we need one term for each positive example. Second, and perhaps more importantly, there is no reason that we should expect the hypothesis that we output to perform well on new data instances we have not seen. In fact, it is guaranteed to predict negative on any example it hasn't already memorized.

This illustrates a major flaw in the Consistency Model: The model says nothing about *generalizability*. A good learning model should say something about the ability of the hypothesis that we output to make predictions on new data.

Another potential problem with the Consistency Model is that it does not easily extend to situations in which the data might be noisy. If there is a single example in which the label is "wrong", the algorithms will give up and say there's no consistent hypothesis. A good learning model should be more robust.

5 Generalizability and the PAC Model

We would like a model that deals with generalization better than the consistency model. To achieve this, it is necessary to make some assumptions. Most of these assumptions can be relaxed, and in fact, we will discuss ways to relax them in upcoming lectures.

For now, we assume that each training or test example $\vec{x}$ is generated at random from a fixed but unknown distribution $\mathcal{D}$ and that the data is *independently and identically distributed* (i.i.d.). We next assume that each label is generated by a fixed but unknown *target concept* $c \in \mathcal{C}$. In other words, we assume that the label of $\vec{x}$ is $c(\vec{x})$. (Note that we are again assuming there is no noise in the data. However, it turns out that this assumption will be easy to relax, as we'll see in upcoming lectures.) We then define the *error* of hypothesis h with respect to the target c as

$$\mathrm{err}(h) = \mathrm{Pr}_{\vec{x} \sim \mathcal{D}} \left[h(\vec{x}) \neq c(\vec{x}) \right].$$

Our goal will be to find a hypothesis h such that the probability that $\mathrm{err}(h)$ is large is small. In other words, we would like to claim that h is *probably approximately correct*.

The "approximately" can be quantified through an *accuracy parameter* ϵ. In particular, since we will generally not have enough data to learn the target c perfectly, we require only that $\mathrm{err}(h) \leq \epsilon$.

The "probably" can be quantified through a *confidence parameter* δ. We can never rule out the unlucky event that we draw an unrepresentative training set and are unable to learn a good approximation of c with the data we have. We instead require that we are able to learn a good approximation with high probability. In particular, we require that $\mathrm{err}(g) \leq \epsilon$ with probability at least $1 - \delta$.

This leads to the following definition. (It may seem a little daunting at first, but it's actually quite natural!)

Definition 3. *An algorithm $\mathcal{A}$ **PAC-learns** a concept class $\mathcal{C}$ by a hypothesis class $\mathcal{H}$ if for any $c \in \mathcal{C}$, for any distribution $\mathcal{D}$ over the input space, for any $\epsilon > 0$ and $\delta > 0$, given access to a polynomial number of examples drawn i.i.d. from $\mathcal{D}$ and labeled by c, $\mathcal{A}$ outputs a function $h \in \mathcal{H}$ such that with probability at least $1 - \delta$, $\mathrm{err}(h) \leq \epsilon$.*

The number of examples may be polynomial in $1/\epsilon$, $1/\delta$, and the "size" of the examples (for example, the number of features).

Note that h is chosen from a class $\mathcal{H}$ which may or may not be the same as $\mathcal{C}$. This can occasionally be useful. For example, for computational reasons, it might be more efficient to learn using a hypothesis class $\mathcal{H}$ such that $\mathcal{C} \subset \mathcal{H}$ rather than restrict our attention to only functions in $\mathcal{C}$.

We can also define what it means for a concept class to be efficiently learnable in the PAC model.

Definition 4. *A concept class $\mathcal{C}$ is **efficiently PAC-learnable** by $\mathcal{H}$ if there exists an efficient algorithm $\mathcal{A}$ that PAC-learns $\mathcal{C}$ by $\mathcal{H}$.*

In the next class we will discuss some general results for the PAC model.