12 | Learning Theory

> The Universe is under no obligation to make sense to you. –
> Neil deGrasse Tyson

Learning Objectives:

- Explain why inductive bias is necessary.
- Define the PAC model and explain why both the "P" and "A" are necessary.
- Explain the relationship between complexity measures and regularizers.
- Identify the role of complexity in generalization.
- Formalize the relationship between margins and complexity.

Dependencies:

By now, you are an expert at building learning algorithms. You probably understand how they work, intuitively. And you understand why they should generalize. However, there are several basic questions you might want to know the answer to. Is learning always possible? How many training examples will I need to do a good job learning? Is my test performance going to be much worse than my training performance? The key idea that underlies all these answer is that *simple functions generalize well.*

The amazing thing is that you can actually prove strong results that address the above questions. In this chapter, you will learn some of the most important results in learning theory that attempt to answer these questions. The goal of this chapter is not theory for theory's sake, but rather as a way to better understand why learning models work, and how to use this theory to build better algorithms. As a concrete example, we will see how 2-norm regularization provably leads to better generalization performance, thus justifying our common practice!

12.1 The Role of Theory

In contrast to the quote at the start of this chapter, a practitioner friend once said "I would happily give up a few percent performance for an algorithm that I can understand." Both perspectives are completely valid, and are actually not contradictory. The second statement is presupposing that theory helps you understand, which hopefully you'll find to be the case in this chapter.

Theory can serve two roles. It can justify and help understand why common practice works. This is the "theory after" view. It can also serve to suggest new algorithms and approaches that turn out to work well in practice. This is the "theory before" view. Often, it turns out to be a mix. Practitioners discover something that works surprisingly well. Theorists figure out why it works and prove something about it. And in the process, they make it better or find new algo-

rithms that more directly exploit whatever property it is that made the theory go through.

Theory can also help you understand what's possible and *what's not possible*. One of the first things we'll see is that, in general, machine learning can not work. Of course it *does* work, so this means that we need to think harder about what it means for learning algorithms to work. By understanding what's not possible, you can focus our energy on things that are.

Probably the biggest *practical* success story for theoretical machine learning is the theory of **boosting**, which you won't actually see in this chapter. (You'll have to wait for Chapter 13.) Boosting is a very simple style of algorithm that came out of theoretical machine learning, and has proven to be incredibly successful in practice. So much so that it is one of the de facto algorithms to run when someone gives you a new data set. In fact, in 2004, Yoav Freund and Rob Schapire won the ACM's Paris Kanellakis Award for their boosting algorithm AdaBoost. This award is given for theoretical accomplishments that have had a significant and demonstrable effect on the practice of computing.[1]

[1] In 2008, Corinna Cortes and Vladimir Vapnik won it for support vector machines.

12.2 Induction is Impossible

One nice thing about theory is that it forces you to be *precise* about what you are trying to do. You've already seen a formal definition of binary classification in Chapter 7. But let's take a step back and re-analyze what it means to learn to do binary classification.

From an *algorithmic* perspective, a natural question is whether there is an "ultimate" learning algorithm, $\mathcal{A}^{\text{awesome}}$, that solves the Binary Classification problem above. In other words, have you been wasting your time learning about *K*NN and Perceptron and decision trees, when $\mathcal{A}^{\text{awesome}}$ is out there.

What would such an ultimate learning algorithm do? You would like it to take in a data set D and produce a function f. No matter what D looks like, this function f should get perfect classification on all future examples drawn from the same distribution that produced D.

A little bit of introspection should demonstrate that this is impossible. For instance, there might be label noise in our distribution. As a very simple example, let $\mathcal{X} = \{-1, +1\}$ (i.e., a one-dimensional, binary distribution. Define the data distribution as:

$$\mathcal{D}(\langle +1\rangle, +1) = 0.4 \qquad \mathcal{D}(\langle -1\rangle, -1) = 0.4 \tag{12.1}$$

$$\mathcal{D}(\langle +1\rangle, -1) = 0.1 \qquad \mathcal{D}(\langle -1\rangle, +1) = 0.1 \tag{12.2}$$

In other words, 80% of data points in this distrubtion have $x = y$

and 20% don't. No matter what function your learning algorithm produces, there's no way that it can do better than 20% error on this data.

Given this, it seems hopeless to have an algorithm $\mathcal{A}^{\text{awesome}}$ that always achieves an error rate of zero. The best that we can hope is that the error rate is not "too large."

Unfortunately, simply weakening our requirement on the error rate is not enough to make learning possible. The second source of difficulty comes from the fact that the only access we have to the data distribution is through sampling. In particular, when trying to learn about a distribution like that in 12.1, you only get to see data points *drawn* from that distribution. You know that "eventually" you will see enough data points that your sample is representative of the distribution, but it might not happen immediately. For instance, even though a fair coin will come up heads only with probability 1/2, it's completely plausible that in a sequence of four coin flips you never see a tails, or perhaps only see one tails.

So the second thing that we have to give up is the hope that $\mathcal{A}^{\text{awesome}}$ will *always* work. In particular, if we happen to get a lousy sample of data from $\mathcal{D}$, we need to allow $\mathcal{A}^{\text{awesome}}$ to do something completely unreasonable.

Thus, we cannot hope that $\mathcal{A}^{\text{awesome}}$ will do perfectly, every time. We cannot even hope that it will do pretty well, all of the time. Nor can we hope that it will do perfectly, most of the time. The best *best* we can reasonably hope of $\mathcal{A}^{\text{awesome}}$ is that it it will do pretty well, most of the time.

? It's clear that if your algorithm produces a *deterministic* function that it cannot do better than 20% error. What if it produces a stochastic (aka randomized) function?

12.3 Probably Approximately Correct Learning

Probably Approximately Correct (**PAC**) learning is a formalism of inductive learning based on the realization that the best we can hope of an algorithm is that it does a good job (i.e., is approximately correct), most of the time (i.e., it is *probably* appoximately correct).[2]

Consider a hypothetical learning algorithm. You run it on ten different binary classification data sets. For each one, it comes back with functions $f_1, f_2, \ldots, f_{10}$. For some reason, whenever you run f_4 on a test point, it crashes your computer. For the other learned functions, their performance on test data is always at most 5% error. If this situtation is guaranteed to happen, then this hypothetical learning algorithm is a PAC learning algorithm. It satisfies "probably" because it only failed in one out of ten cases, and it's "approximate" because it achieved low, but non-zero, error on the remainder of the cases.

This leads to the formal definition of an (ϵ, δ) PAC-learning algorithm. In this definition, ϵ plays the role of measuring accuracy (in

[2] Leslie Valiant invented the notion of PAC learning in 1984. In 2011, he received the Turing Award, the highest honor in computing for his work in learning theory, computational complexity and parallel systems.

the previous example, $\epsilon = 0.05$) and δ plays the role of measuring failure (in the previous, $\delta = 0.1$).

Definitions 1. *An algorithm $\mathcal{A}$ is an (ϵ, δ)-PAC learning algorithm if,* for all *distributions $\mathcal{D}$: given samples from $\mathcal{D}$, the probability that it returns a "bad function" is at most δ; where a "bad" function is one with test error rate more than ϵ on $\mathcal{D}$.*

There are two notions of *efficiency* that matter in PAC learning. The first is the usual notion of *computational complexity*. You would prefer an algorithm that runs quickly to one that takes forever. The second is the notion of **sample complexity**: the number of examples required for your algorithm to achieve its goals. Note that the goal of both of these measure of complexity is to bound how much of a scarse resource your algorithm uses. In the computational case, the resource is CPU cycles. In the sample case, the resource is labeled examples.

Definition: An algorithm $\mathcal{A}$ is an efficient (ϵ, δ)-PAC learning algorithm if it is an (ϵ, δ)-PAC learning algorithm whose runtime is polynomial in $\frac{1}{\epsilon}$ and $\frac{1}{\delta}$.

In other words, suppose that you want your algorithm to achieve 4% error rate rather than 5%. The runtime required to do so should no go up by an exponential factor.

12.4 PAC Learning of Conjunctions

To get a better sense of PAC learning, we will start with a completely irrelevant and uninteresting example. The purpose of this example is *only* to help understand how PAC learning works.

The setting is learning conjunctions. Your data points are binary vectors, for instance $\boldsymbol{x} = \langle 0, 1, 1, 0, 1 \rangle$. Someone guarantees for you that there is some boolean conjunction that defines the true labeling of this data. For instance, $x_1 \wedge \neg x_2 \wedge x_5$ ("or" is not allowed). In formal terms, we often call the true underlying classification function the **concept**. So this is saying that the concept you are trying to learn is a conjunction. In this case, the boolean function would assign a negative label to the example above.

Since you know that the concept you are trying to learn is a conjunction, it makes sense that you would represent your function as a conjunction as well. For historical reasons, the function that you learn is often called a **hypothesis** and is often denoted h. However, in keeping with the other notation in this book, we will continue to denote it f.

Formally, the set up is as follows. There is some distribution $\mathcal{D}^X$ over binary data points (vectors) $\boldsymbol{x} = \langle x_1, x_2, \ldots, x_D \rangle$. There is a fixed

concept conjunction c that we are trying to learn. There is no noise, so for any example $\boldsymbol{x}$, its true label is simply $y = c(\boldsymbol{x})$.

What is a reasonable algorithm in this case? Suppose that you observe the example in Table 12.1. From the first example, we know that the true formula cannot include the term x_1. If it did, this example would have to be negative, which it is not. By the same reasoning, it cannot include x_2. By analogous reasoning, it also can neither include the term $\neg x_3$ nor the term $\neg x_4$.

y	x_1	x_2	x_3	x_4
+1	0	0	1	1
+1	0	1	1	1
-1	1	1	0	1

Table 12.1: Data set for learning conjunctions.

This suggests the algorithm in Algorithm 12.4, colloquially the "Throw Out Bad Terms" algorithm. In this algorithm, you begin with a function that includes all possible $2D$ terms. Note that this function will initially classify everything as negative. You then process each example in sequence. On a negative example, you do nothing. On a positive example, you throw out terms from f that contradict the given positive example.

? Verify that Algorithm 12.4 maintains an *invariant* that it always errs on the side of classifying examples negative and never errs the other way.

If you run this algorithm on the data in Table 12.1, the sequence of fs that you cycle through are:

$$f^0(\boldsymbol{x}) = x_1 \wedge \neg x_1 \wedge x_2 \wedge \neg x_2 \wedge x_3 \wedge \neg x_3 \wedge x_4 \wedge \neg x_4 \tag{12.3}$$
$$f^1(\boldsymbol{x}) = \neg x_1 \wedge \neg x_2 \wedge x_3 \wedge x_4 \tag{12.4}$$
$$f^2(\boldsymbol{x}) = \neg x_1 \wedge x_3 \wedge x_4 \tag{12.5}$$
$$f^3(\boldsymbol{x}) = \neg x_1 \wedge x_3 \wedge x_4 \tag{12.6}$$

The first thing to notice about this algorithm is that after processing an example, it is guaranteed to classify that example correctly. This observation *requires* that there is no noise in the data.

The second thing to notice is that it's very computationally efficient. Given a data set of N examples in D dimensions, it takes $\mathcal{O}(ND)$ time to process the data. This is linear in the size of the data set.

However, in order to be an efficient (ϵ, δ)-PAC learning algorithm, you need to be able to get a bound on the **sample complexity** of this algorithm. Sure, you know that its run time is linear in the number of example N. But *how many* examples N do you need to see in order to guarantee that it achieves an error rate of at most ϵ (in all but δ-many cases)? Perhaps N has to be gigantic (like $2^{2^{D/\epsilon}}$) to (probably) guarantee a small error.

The goal is to prove that the number of samples N required to (probably) achieve a small error is not-too-big. The general proof technique for this has essentially the same flavor as almost every PAC learning proof around. First, you define a "bad thing." In this case, a "bad thing" is that there is some term (say $\neg x_8$) that should have been thrown out, but wasn't. Then you say: well, bad things happen. Then you notice that if this bad thing happened, you must not have

Algorithm 31 BINARYCONJUNCTIONTRAIN(**D**)

```
f ← x_1 ∧ ¬x_1 ∧ x_2 ∧ ¬x_2 ∧ ··· ∧ x_D ∧ ¬x_D          // initialize function
for all positive examples (x,+1) in D do
  for d = 1 ... D do
    if x_d = 0 then
      f ← f without term "x_d"
    else
      f ← f without term "¬x_d"
    end if
  end for
end for
return f
```

seen any positive training examples with $x_8 = 0$. So example with $x_8 = 0$ must have low probability (otherwise you would have seen them). So bad things must not be that common.

Theorem 14. *With probability at least $(1 - \delta)$: Algorithm 12.4 requires at most $N = \ldots$ examples to achieve an error rate $\leq \epsilon$.*

Proof of Theorem 14. Let c be the concept you are trying to learn and let $\mathcal{D}$ be the distribution that generates the data.

A learned function f can make a mistake if it contains *any* term t that is not in c. There are initially $2D$ many terms in f, and any (or all!) of them might not be in c. We want to ensure that the probability that f makes an error is at most ϵ. It is sufficient to ensure that

For a term t (e.g., $\neg x_5$), we say that t "negates" an example $\boldsymbol{x}$ if $t(\boldsymbol{x}) = 0$. Call a term t "bad" if (a) it does not appear in c and (b) has probability at least $\epsilon/2D$ of appearing (with respect to the unknown distribution $\mathcal{D}$ over data points).

First, we show that if we have no bad terms left in f, then f has an error rate at most ϵ.

We know that f contains *at most* $2D$ terms, since is begins with $2D$ terms and throws them out.

The algorithm begins with $2D$ terms (one for each variable and one for each negated variable). Note that f will only make one type of error: it can call positive examples negative, but can never call a negative example positive. Let c be the true concept (true boolean formula) and call a term "bad" if it does not appear in c. A specific bad term (e.g., $\neg x_5$) will cause f to err only on positive examples that contain a corresponding bad value (e.g., $x_5 = 1$). TODO... finish this □

What we've shown in this theorem is that: *if* the true underlying concept is a boolean conjunction, *and* there is no noise, *then* the "Throw Out Bad Terms" algorithm needs $N \leq \ldots$ examples in order

to learn a boolean conjunction that is $(1-\delta)$-likely to achieve an error of at most ϵ. That is to say, that the **sample complexity** of "Throw Out Bad Terms" is .... Moreover, since the algorithm's runtime is linear in N, it is an efficient PAC learning algorithm.

12.5 *Occam's Razor: Simple Solutions Generalize*

The previous example of boolean conjunctions is mostly just a warm-up exercise to understand PAC-style proofs in a concrete setting. In this section, you get to generalize the above argument to a much larger range of learning problems. We will still assume that there is no noise, because it makes the analysis much simpler. (Don't worry: noise will be added eventually.)

William of Occam (c. 1288 – c. 1348) was an English friar and philosopher is is most famous for what later became known as Occam's razor and popularized by Bertrand Russell. The principle basically states that you should only assume as much as you need. Or, more verbosely, "if one can explain a phenomenon without assuming this or that hypothetical entity, then there is no ground for assuming it i.e. that one should always opt for an explanation in terms of the fewest possible number of causes, factors, or variables." What Occam actually wrote is the quote that began this chapter.

In a machine learning context, a reasonable paraphrase is "simple solutions generalize well." In other words, you have $10,000$ features you could be looking at. If you're able to explain your predictions using just 5 of them, or using all $10,000$ of them, then you should just use the 5.

The Occam's razor theorem states that this is a good idea, theoretically. It essentially states that if you are learning some unknown concept, and if you are able to fit your training data perfectly, but you don't need to resort to a huge class of possible functions to do so, then your learned function will generalize well. It's an amazing theorem, due partly to the simplicity of its proof. In some ways, the proof is actually *easier* than the proof of the boolean conjunctions, though it follows the same basic argument.

In order to state the theorem explicitly, you need to be able to think about a **hypothesis class**. This is the set of possible hypotheses that your algorithm searches through to find the "best" one. In the case of the boolean conjunctions example, the hypothesis class, $\mathcal{H}$, is the set of all boolean formulae over D-many variables. In the case of a perceptron, your hypothesis class is the set of all possible linear classifiers. The hypothesis class for boolean conjunctions is *finite*; the hypothesis class for linear classifiers is *infinite*. For Occam's razor, we can only work with finite hypothesis classes.

Theorem 15 (Occam's Bound)**.** *Suppose $\mathcal{A}$ is an algorithm that learns a function f from some* finite *hypothesis class $\mathcal{H}$. Suppose the learned function always gets zero error on the training data. Then, the sample complexity of f is at most* $\log |\mathcal{H}|$.

TODO COMMENTS

Proof of Theorem 15. TODO □

This theorem applies directly to the "Throw Out Bad Terms" algorithm, since (a) the hypothesis class is finite and (b) the learned function always achieves zero error on the training data. To apply Occam's Bound, you need only compute the size of the hypothesis class $\mathcal{H}$ of boolean conjunctions. You can compute this by noticing that there are a total of $2D$ possible terms in any formula in $\mathcal{H}$. Moreover, each term may or may not be in a formula. So there are $2^{2D} = 4^D$ possible formulae; thus, $|\mathcal{H}| = 4^D$. Applying Occam's Bound, we see that the sample complexity of this algorithm is $N \leq \ldots$.

Of course, Occam's Bound is general enough to capture other learning algorithms as well. In particular, it can capture decision trees! In the no-noise setting, a decision tree will always fit the training data perfectly. The only remaining difficulty is to compute the size of the hypothesis class of a decision tree learner.

For simplicity's sake, suppose that our decision tree algorithm always learns complete trees: i.e., every branch from root to leaf is length D. So the number of split points in the tree (i.e., places where a feature is queried) is 2^{D-1}. (See Figure 12.1.) Each split point needs to be assigned a feature: there D-many choices here. This gives $D2^{D-1}$ trees. The last thing is that there are 2^D leaves of the tree, each of which can take two possible values, depending on whether this leaf is classified as $+1$ or -1: this is $2{\times}2^D = 2^{D+1}$ possibilities. Putting this all togeter gives a total number of trees $|\mathcal{H}| = D2^{D-1}2^{D+1} = D2^{2D} = D4^D$. Applying Occam's Bound, we see that *TODO* examples is enough to learn a decision tree!

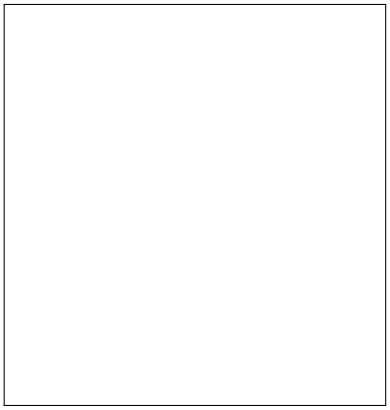

Figure 12.1: `thy:dt`: picture of full decision tree

12.6 Complexity of Infinite Hypothesis Spaces

Occam's Bound is a fantastic result for learning over *finite* hypothesis spaces. Unfortunately, it is completely useless when $|\mathcal{H}| = \infty$. This is because the proof works by using each of the N training examples to "throw out" bad hypotheses until only a small number are left. But if $|\mathcal{H}| = \infty$, and you're throwing out a finite number at each step, there will always be an infinite number remaining.

This means that, if you want to establish sample complexity results for infinite hypothesis spaces, you need some new way of measuring

their "size" or "complexity." A prototypical way of doing this is to measure the complexity of a hypothesis class as the *number of different things it can do*.

As a silly example, consider boolean conjunctions again. Your input is a vector of binary features. However, instead of representing your hypothesis as a boolean conjunction, you choose to represent it as a conjunction of *inequalities*. That is, instead of writing $x_1 \wedge \neg x_2 \wedge x_5$, you write $[x_1 > 0.2] \wedge [x_2 < 0.77] \wedge [x_5 < \pi/4]$. In this representation, for each feature, you need to choose an inequality ($<$ or $>$) and a threshold. Since the thresholds can be arbitrary real values, there are now infinitely many possibilities: $|\mathcal{H}| = 2^D \times \infty = \infty$. However, you can immediately recognize that on binary features, there really is no difference between $[x_2 < 0.77]$ and $[x_2 < 0.12]$ and any other number of infinitely many possibilities. In other words, *even though there are infinitely many hypotheses, there are only finitely many behaviors.*

The **Vapnik-Chernovenkis dimension** (or **VC dimension**) is a classic measure of complexity of infinite hypothesis classes based on this intuition[3]. The VC dimension is a very classification-oriented notion of complexity. The idea is to look at a finite set of unlabeled examples, such as those in Figure 12.2. The question is: no matter how these points were labeled, would we be able to find a hypothesis that correctly classifies them. The idea is that as you add more points, being able to represent an arbitrary labeling becomes harder and harder. For instance, regardless of how the three points are labeled, you can find a linear classifier that agrees with that classification. However, for the four points, there exists a labeling for which you cannot find a perfect classifier. The VC dimension is the *maximum* number of points for which you can always find such a classifier.

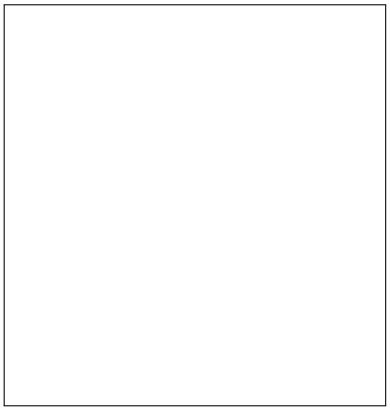

Figure 12.2: thy:vcex: figure with three and four examples

[3] Yes, this is the same Vapnik who is credited with the creation of the support vector machine.

? What is that labeling? What is it's name?

You can think of VC dimension as a game between you and an adversary. To play this game, *you* choose K unlabeled points however you want. Then your adversary looks at those K points and assigns binary labels to them them however they want. You must then find a hypothesis (classifier) that agrees with their labeling. You win if you can find such a hypothesis; they win if you cannot. The VC dimension of your hypothesis class is the *maximum* number of points K so that you can always win this game. This leads to the following formal definition, where you can interpret there exists as your move and for all as adversary's move.

Definitions 2. *For data drawn from some space $\mathcal{X}$, the VC dimension of a hypothesis space $\mathcal{H}$ over $\mathcal{X}$ is the* maximal *K such that: there exists a set $X \subseteq \mathcal{X}$ of size $|X| = K$, such that for all binary labelings of X, there exists a function $f \in \mathcal{H}$ that matches this labeling.*

In general, it is much easier to show that the VC dimension is at least some value; it is much harder to show that it is at most some value. For example, following on the example from Figure 12.2, the image of three points (plus a little argumentation) is enough to show that the VC dimension of linear classifiers in two dimension is *at least three*.

To show that the VC dimension is *exactly three* it suffices to show that you *cannot* find a set of four points such that you win this game against the adversary. This is much more difficult. In the proof that the VC dimension is at least three, you simply need to provide an example of three points, and then work through the small number of possible labelings of that data. To show that it is at most three, you need to argue that no matter what set of four point you pick, you cannot win the game.

12.7 Further Reading

TODO