A Formal Proof of PAC Learnability for Decision Stumps

Dedicated to Olivier Danvy on the occasion of his sixtieth birthday.

Joseph Tassarotti
tassarot@bc.edu
Boston College
USA

Koundinya Vajjha
kov5@pitt.edu
University of Pittsburgh
USA

Anindya Banerjee
anindya.banerjee@imdea.org
IMDEA Software Institute
Spain

Jean-Baptiste Tristan*
tristanj@bc.edu
Boston College
USA

Abstract

We present a formal proof in Lean of probably approximately correct (PAC) learnability of the concept class of decision stumps. This classic result in machine learning theory derives a bound on error probabilities for a simple type of classifier. Though such a proof appears simple on paper, analytic and measure-theoretic subtleties arise when carrying it out fully formally. Our proof is structured so as to separate reasoning about deterministic properties of a learning function from proofs of measurability and analysis of probabilities.

CCS Concepts: • **General and reference** → **Verification**; • **Theory of computation** → *Sample complexity and generalization bounds.*

Keywords: interactive theorem proving, probably approximately correct, decision stumps

ACM Reference Format:
Joseph Tassarotti, Koundinya Vajjha, Anindya Banerjee, and Jean-Baptiste Tristan. 2021. A Formal Proof of PAC Learnability for Decision Stumps. In *Proceedings of the 10th ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP '21), January 18–19, 2021, Virtual, Denmark.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3437992.3439917

*Corresponding author

CPP '21, January 18–19, 2021, Virtual, Denmark

ACM ISBN 978-1-4503-8299-1/21/01…$15.00
https://doi.org/10.1145/3437992.3439917

1 Introduction

Machine learning (ML) has achieved remarkable success in a number of problem domains. However, the often opaque nature of ML has led to concerns about its use in important contexts such as medical diagnosis and fraud detection. To address these concerns, researchers have developed a number of algorithms with proved guarantees of robustness, privacy, fairness, and accuracy.

One essential property for an ML algorithm is to *generalize* well to unseen data. That is, after an algorithm has been trained on some data, it should be possible to present it with new data and expect it to classify or analyze it correctly. Valiant introduced the framework of Probably Approximately Correct (PAC) learnability [39], which gives a mathematical characterization of what it means for an algorithm to generalize well. This framework has become an essential part of the study of computational and statistical learning theory, and a large body of theoretical results has been developed for proving that an algorithm generalizes.

However, at present, the vast majority of these and other proofs in ML theory are pencil-and-paper arguments about idealized versions of algorithms. There is considerable room for error when real systems are built based on these algorithms. Such errors can go unnoticed for long periods of time, and are often difficult to diagnose with testing, given the randomized behavior of ML systems. Moreover, the original pencil-and-paper proofs of correctness may have errors. Because machine learning algorithms often involve randomized sampling of continuous data, their formal analysis usually requires measure-theoretic reasoning, which is technically subtle.

Formal verification offers a way to eliminate bugs from analyses of algorithms and close the gap between theory and implementation. However, the mathematical subtleties that complicate rigorous pencil-and-paper reasoning about ML algorithms also pose a serious obstacle to verification. In particular, while there has been great progress in recent

years in formal proofs about randomized programs, this work has often been restricted to *discrete* probability theory. In contrast, machine learning algorithms make heavy use of both discrete and continuous data, a mixture that requires measure-theoretic probability.

Thus, to even begin formally verifying ML algorithms in a theorem prover, results from measure theory must be formalized first. In recent years, some libraries of measure-theoretic results have been developed in various theorem provers [1, 20, 27, 36, 38]. However, it can be challenging to tell which results needed for ML algorithms are missing from these libraries. The difficulty is that, on the one hand, standard textbooks on the theoretical foundations of machine learning [28, 29, 35] omit practically all measure-theoretic details. Meanwhile, research monographs that give completely rigorous accounts [16] present results in maximal generality, well beyond what appears to be needed for many ML applications. This generality comes at the cost of mathematical prerequisites that go beyond even a first or second course in measure theory.

This paper describes the formal verification in Lean [15] of a standard result of theoretical machine learning which illustrates the complexities of measure-theoretic probability. We consider a simple type of classifier called a *decision stump*. A decision stump classifies real-numbered values into two groups with a simple rule: all values $\leq$ some threshold t are labeled 1, and those above t are labeled 0. We show that decision stumps are PAC learnable by proving a *generalization* bound, which bounds the number of training examples needed to obtain a chosen level of classification accuracy with high probability.

We describe more precisely below how decision stumps are trained and the bound that we have proved (Section 2). Although decision stumps appear simple, they are worth considering because they are the 1-dimensional version of a classifying algorithm for axis-aligned rectangles that is used as a motivating example in all of the standard textbooks in this field [28, 29, 35].

In spite of the seeming simplicity of this example, all three of the cited textbooks either give an incorrect proof of this result or omit what we found to be the most technically challenging part of the proof (Section 3). In noting this, we do not wish to exaggerate the importance of these errors. The proofs can be fixed, and in each book, the results are correctly re-proven later as consequences of general theorems. Rather our point is to emphasize that even basic results in this area can touch on subtle issues, and errors can evade notice despite much review and scrutiny by a wide audience. We believe this further motivates the need for machine-checked proofs of such results if they are to be deployed in high-importance settings.

A key component of our work is that we structure our formal proof in a manner that lets us separate the high-level reasoning found in textbook descriptions from the low-level details about measurability. We outline the structure used to achieve this separation of concerns in Section 4. We then describe some preliminaries about measure-theoretic probability in Section 5. The Giry monad [18] allows us to give a precise description of the sources of randomness involved in training and evaluating the performance of classifiers (Section 6). We exploit this to split deterministic reasoning about basic properties of the stump learning algorithm (Section 7) from proofs of measurability (Section 8) and the analysis of bounds on probabilities (Section 9).

The proof is publicly available at https://github.com/jtristan/stump-learnable.

2 Decision Stumps and PAC Learnability

To motivate decision stumps and the result that we have formalized, consider the following scenario. Suppose a scientist has developed a test to measure levels of some protein in blood in order to diagnose a disease. Assume there is some (unknown) threshold $t \in \mathbb{R}$ such that if the protein level is $\leq t$, then the patient has the disease, and otherwise does not. Given a random sample of patients whose disease status is known, the scientist wants to estimate the threshold t so that the test can be used to screen and diagnose future patients whose disease statuses are unknown.

In other words, the scientist wants to find a decision stump to classify whether patients have the disease or not. We can model the blood test as returning a nonnegative real number. There is some distribution μ on the interval $[0, \infty)$ representing levels of the protein in the population. The scientist has samples $x_1, \ldots, x_n \in [0, \infty)$ independently drawn from μ giving the results of the test on a collection of n patients, along with labels $y_1, \ldots, y_n \in \{0, 1\}$ giving each patient's disease status. A label of 1 means a patient has the disease, while 0 means they do not, so that $y_i = 1$ if and only if $x_i \leq t$. The $(x_1, y_1), \ldots, (x_n, y_n)$ are called *training examples*. The scientist is trying to pick some value $\hat{t}$ as an estimate of t to use to classify future patients. In particular, she will use her estimate to define a decision stump classifier. To state this formally, we first define a function Label that assigns a label to a point x according to a threshold d:

$$\textsc{Label}(d, x) = \begin{cases} 1 & \text{if } x \leq d \\ 0 & \text{if } x > d \end{cases} \tag{1}$$

The scientist will pick some threshold $\hat{t}$ and then use the classifier $\lambda x.\, \textsc{Label}(\hat{t}, x)$ to label future patients. We call such a classifier a *hypothesis*.

How should the scientist select $\hat{t}$? One idea is to take $\hat{t}$ to be the maximum of the x_i that have label 1. (If no x_i has label 1, she can take $\hat{t}$ to be 0.) This estimate, at least, would correctly label all of the training examples. This corresponds to the following *learning algorithm* $\mathcal{A}$, which returns a classifier

using this estimate:

$$\begin{aligned}&\mathcal{A}\left([(x_1, y_1), \ldots, (x_n, y_n)]\right)\\ &= \text{let } \hat{t} = \max\{x_i \mid y_i = 1\} \text{ in}\\ &\quad \lambda x.\, \textsc{Label}(\hat{t}, x)\end{aligned} \tag{2}$$

where max of the empty set is defined to be 0.[1] Of course, the estimate $\hat{t}$ used in the classifier returned by this algorithm is not going to be exactly the value of t, especially if the number of training examples, n, is small. But if n is large enough, we might hope that a good estimate can be produced. The key question then becomes, how many training examples should the scientist use?

To answer this more precisely, we need to decide how to evaluate the quality of the classifier returned by $\mathcal{A}$. At first, one might think that the goal should be to minimize $|\hat{t} - t|$, that is, to get an estimate that is as close as possible to the true threshold t. While minimizing the distance between $\hat{t}$ and t is useful, our primary concern should be how well we classify future examples. The Error of a classifier h is the probability that h mislabels a test example x randomly sampled from μ:

$$\textsc{Error}(h) = \Pr_{x\sim\mu}\left(h(x) \neq \textsc{Label}(t, x)\right) \tag{3}$$

We write $x \sim \mu$ in the above to indicate that the random variable x has distribution μ. This x is independent of the training examples used by the scientist. While the definition of Error refers to this randomized scenario of drawing a test sample, for a fixed hypothesis h the quantity $\textsc{Error}(h)$ is a real number.

Because the training examples the scientist uses are randomly selected, the Error of the hypothesis she selects using $\mathcal{A}$ is a random variable. In practice, the scientist does not know either the distribution μ or the target t, so she cannot compute the exact Error of the classifier she obtains. Nevertheless, she might want to try to ensure that the Error of the classifier she defines using $\mathcal{A}$ will be below some small ϵ with high probability.

To talk about the Error of the selected hypothesis precisely, let us first introduce a helper function which takes an unlabeled list of examples and returns a list where each example has been paired with its true label:

$$\begin{aligned}&\textsc{LList}([x_1, \ldots, x_n])\\ &\quad = \textsc{Map}\left(\lambda x.\, (x, \textsc{Label}(t, x))\right)\, [x_1, \ldots, x_n]\end{aligned} \tag{4}$$

Then through her choice of n, the scientist can bound the following probability:

$$\Pr_{(x_1,\ldots,x_n)\sim\mu^n}\left(\textsc{Error}(\mathcal{A}(\textsc{LList}([x_1, \ldots, x_n]))) \leq \epsilon\right)$$

where $(x_1, \ldots, x_n) \sim \mu^n$ indicates that the variables x_i are drawn independently from the same distribution μ. The following theorem, which is the central result that we have formalized, tells the scientist how to select n to achieve a desired bound on this probability:

Theorem 2.1. For all ϵ and δ in the open interval $(0, 1)$, if $n \geq \frac{\ln(\delta)}{\ln(1-\epsilon)} - 1$ then

$$\Pr_{(x_1,\ldots,x_n)\sim\mu^n}\left(\textsc{Error}(\mathcal{A}(\textsc{LList}([x_1, \ldots, x_n]))) \leq \epsilon\right) \geq 1 - \delta \tag{5}$$

Before giving an informal sketch of how this theorem is proved, we briefly describe how this result fits into the framework of PAC learnability [39].

PAC Learnability. PAC learning theory gives an abstract way to explain scenarios like the one with the scientist described above. In this general set up, there is some set $\mathcal{X}$ of possible examples and a set C of hypotheses $h : \mathcal{X} \to \{0, 1\}$, which are possible classifiers we might select. The set C is also called a *concept class*. In the case of decision stumps, $\mathcal{X} = \mathbb{R}^{\geq 0}$, and $C = \{\lambda x.\, \textsc{Label}(d, x) \mid d \in \mathbb{R}^{\geq 0}\}$.

As in the stump example, there is assumed to be some unknown distribution μ over $\mathcal{X}$. Additionally, there is some function $f : \mathcal{X} \to \{0, 1\}$ that maps examples to their true labels. The Error of a hypothesis is the probability that it incorrectly labels an example drawn according to μ. The function f is said to be *realizable* if there is some hypothesis $h \in C$ that has error 0. The goal is to select a hypothesis with minimal Error when given a collection of training examples that have been labeled according to f. A concept class is said to be PAC learnable if there is some algorithm to select a hypothesis for which we can compute the number of training examples needed to achieve error bounds with high probability as in the stump example:

Definition 2.2. A concept class C is PAC learnable if there exists an algorithm $\mathcal{A} : \text{List}(\mathcal{X} \times \{0, 1\}) \to C$ and a function $g : (0, 1)^2 \to \mathbb{N}$ such that for all distributions μ on $\mathcal{X}$ and realizable label functions f, when $\mathcal{A}$ is run on a list of at least $g(\epsilon, \delta)$ independently sampled examples from μ with labels computed by f, the returned hypothesis h has error $\leq \epsilon$ with probability $\geq 1 - \delta$.

By "there exists" in the above definition, one typically conveys a constructive sense: one can exhibit the algorithm and compute g.[2] The function g is a bound on the *sample complexity* of the algorithm, telling us how many training examples are needed to achieve a bound on the Error with a given probability. There is a large body of theoretical results for showing that a concept class is PAC learnable and for bounding the sample complexity by analyzing the VC-dimension [41] of the concept class.

[1] We call this function an *algorithm* here, following Shalev-Shwartz and Ben-David [35], although the operations on real numbers involved are non-computable.

[2] Some authors require further that the algorithm $\mathcal{A}$ has polynomial running time, but we will not do so.

Theorem 2.1 states that the concept class of decision stumps is PAC learnable. We next turn to how this theorem is proved.

3 Informal Proof of PAC Learnability

The PAC learnability of decision stumps follows from the general VC-dimension theory alluded to above. However, mechanizing that underlying theory in all its generality is beyond the scope of this paper.

Instead this paper formalizes a more elementary proof, based on a description from three textbooks [28, 29, 35]. The proofs in the textbooks are for a slightly different result—learning a 2-dimensional rectangle instead of a decision stump—but the idea is essentially the same. We first sketch how the proof is presented in two of the textbooks [28, 35]. As we shall see, this argument has a flaw.

Proof sketch of Theorem 2.1. Given μ and ϵ, consider labeled samples $(x_1, y_1), ..., (x_n, y_n)$. Recall that the true threshold is some unknown value t, and the learning algorithm $\mathcal{A}$ here returns a classifier that uses the maximum of the positively labeled examples as the decision threshold $\hat{t}$.

We start by noting some deterministic properties about this classifier. Observe that $\mathcal{A}$ ensures $\hat{t} \leq t$, because all positively labeled training examples must be $\leq t$. Furthermore, a test example x is misclassified only if $\hat{t} < x \leq t$. Thus the only errors that the classifier selected by $\mathcal{A}$ can make is by incorrectly assigning the label 0 to an example that should have label 1.

With this in mind, the proof goes by cases on the probability that a randomly sampled example $x \sim \mu$ will have true label 1. First assume $\Pr_{x\sim\mu}(x \leq t) \leq \epsilon$. That is, this case assumes that examples with true label 1 are rare. Then the classifier returned by $\mathcal{A}$ must have Error that is $\leq \epsilon$. In other words, if the returned classifier can only misclassify examples whose true label is 1, and those are sufficiently rare, i.e., have probability $\leq \epsilon$ by the above assumption, then the classifier has the desired error bound.

Next assume $\Pr_{x\sim\mu}(x \leq t) > \epsilon$. The idea for this case is to find an interval $\mathcal{I}$ such that, so long as at least one of the training examples x_i falls into $\mathcal{I}$, the classifier returned by $\mathcal{A}$ will have error $\leq \epsilon$. Then we find a bound on the probability that *none* of the x_i fall into $\mathcal{I}$.

In particular, set $\mathcal{I} = [\theta, t]$, choosing θ so that

$$\Pr_{x\sim\mu}(x \in \mathcal{I}) = \epsilon$$

That is, we want $\mathcal{I}$ to enclose exactly probability ϵ under μ. Let E be the event that at least one of the training examples falls in $\mathcal{I}$. If E occurs, then the threshold $\hat{t}$ selected by $\mathcal{A}$ is in $\mathcal{I}$. To see this, observe that if for some x_i we have $\theta \leq x_i \leq t$, then we know $y_i = 1$, and hence $\theta \leq x_i \leq \hat{t} \leq t$.

In that case, for a test example x to be misclassified, we must have $\hat{t} < x \leq t$, meaning x must also lie in $\mathcal{I}$. Thus, the event of misclassifying x is a subset of the event that x lies in $\mathcal{I}$. Hence, if E occurs, the probability of misclassifying x is at most the probability that x lies in $\mathcal{I}$. But the probability that x lies in $\mathcal{I}$ is ϵ by the way we defined θ. Therefore if E occurs, the probability that a randomly selected example x will be misclassified is $\leq \epsilon$, meaning the Error of the classifier will be $\leq \epsilon$.

So for the error to be above ϵ means that *none* of our training examples x_i came from $\mathcal{I}$. For each i, we have

$$\Pr_{(x_1,...,x_n)}(x_i \notin \mathcal{I}) = 1 - \epsilon$$

Because each x_i is sampled independently from μ, the probability that none of the x_i lie in $\mathcal{I}$ is $(1-\epsilon)^n$. Thus the probability of E, the event that *at least* one x_i is in $\mathcal{I}$, is $1-(1-\epsilon)^n$. Since we have shown that if E occurs, then the Error is $\leq \epsilon$, this means that the probability that the Error is $\leq \epsilon$ is *at least* the probability of E. The rest of the proof follows by choosing n to ensure $1-(1-\epsilon)^n \geq 1-\delta$. □

The careful reader may notice that there is one subtle step in the above: how do we choose θ to ensure that "$\mathcal{I}$ encloses exactly probability ϵ under μ"? The phrasing "encloses exactly" comes from Kearns and Vazirani [28] (page 4), which does not say how to prove that θ exists, beyond giving some geometric intuition in which we visualize shifting the left edge of $\mathcal{I}$ until the enclosed amount has the specified probability. Shalev-Shwartz and Ben-David [35] similarly instructs us to select θ so that the probability "is exactly" ϵ.[3]

Unfortunately, the argument is not correct, because such a θ may not exist.[4] The following counterexample demonstrates this.

Counterexample 3.1. Take μ to be the Bernoulli distribution which returns 1 with probability .5 and 0 otherwise. Let $t = .5$, and $\epsilon = .25$. Then for all a we have:

$$\Pr_{x\sim\mu}(x \in [a, t]) = \begin{cases} .5 & \text{if } a \leq 0 \\ 0 & \text{otherwise} \end{cases} \tag{6}$$

so this means that no matter how we select θ, we cannot have $\Pr_{x\sim\mu}(x \in [\theta, t]) = \epsilon$, so the desired θ does not exist.

The issue in the proof is that the distribution function μ has been assumed to be *continuous*, whereas in the above counterexample μ is a *discrete* distribution. In particular, if there is some point y such that $\Pr_{x\sim\mu}(x = y) > 0$ then this introduces a *jump discontinuity* in the distribution function.

However, the statement of PAC learnability says that the error bound should be achievable for *any* distribution μ. In order to fix the proof to work for any μ, we need to consider

[3] The cited references address the more general problem of axis-aligned rectangles instead of stumps, so more specifically they describe shifting the edge of a rectangle until the enclosed probability is $\epsilon/4$.

[4] We are not the first to observe this error. The errata for the first printing of Mohri et al. [29] points out the issue in the proof of Kearns and Vazirani [28].

the following revised definition of θ, which will ensure it exists:

$$\theta = \sup \left\{ d \in \mathcal{X} \mid \Pr_{x \sim \mu} (x \in [d, t]) \geq \epsilon \right\} \tag{7}$$

In this definition, the set (say S) over which we are taking the supremum might be infinite. However, recall that we only need to construct the point θ in the sub-case of the proof where we assume that $\Pr_{x \sim \mu}(x \leq t) > \epsilon$. This assumption implies that the supremum exists, because it means that S is nonempty, and furthermore we know that S is bounded above by t. The existence of the supremum then follows from the fact that the real numbers are Dedekind complete.

The idea behind this definition of θ is that, if the distribution function is continuous, then the definition picks a θ that has the property required in the erroneous proof. Instead if there is a discontinuity that causes a jump in the distribution function past the value ϵ, then the definition selects the point at that discontinuity. In particular, we can show that with this definition

$$\Pr_{x \sim \mu} (x \in [\theta, t]) \geq \epsilon \tag{8}$$

and *also* that

$$\Pr_{x \sim \mu} (x \in (\theta, t]) \leq \epsilon \tag{9}$$

Note in Equation 8 we have the closed interval $[\theta, t]$, while Equation 9 is about the half-open interval $(\theta, t]$. This means that if $\Pr_{x \sim \mu}(x = \theta) = 0$, as we would have in a continuous probability distribution, then $\Pr_{x \sim \mu}(x \in [\theta, t]) = \epsilon$. Whereas if $\Pr_{x \sim \mu}(x = \theta) \neq 0$, as can occur in a discrete distribution, the probabilities of lying in $[\theta, t]$ and $(\theta, t]$ will differ. For example, for the discrete distribution in Counterexample 3.1, the definition of θ in Equation 7 would yield $\theta = 0$. Observe that $\Pr_{x \sim \mu}(x = 0) = .5 \neq 0$ while $\Pr_{x \sim \mu}(x = v) = 0$ for any $v \in (0, 1)$.

The original proof of PAC learnability of the class of rectangles [13] did give a correct definition of θ, as does the textbook by Mohri et al. [29], although neither gives a proof for why the point defined this way has the desired properties. Indeed, Mohri et al. say that it is "not hard to see" that these properties hold.

In fact, this turned out to be the most difficult part of the whole proof to formalize. While it only requires some basic results in measure theory and topology, it is nevertheless the most technical step of the argument. There were two other parts of the proof that seemed obvious on paper but turned out to be much more technically challenging than expected, having to do with showing that various functions are measurable. Often, details about measurability are elided in pencil-and-paper proofs. This is understandable because these measurability concerns can be tedious and trivial, and checking that everything is measurable can clutter an otherwise insightful proof. However, many important results in statistical learning theory do not hold without certain measurability assumptions, as discussed by Blumer et al. [13] and Dudley [16, chapter 5].

Now that we have seen some intuition for this result and some of the pitfalls in proving it, we describe the structure of our formal proof and how it addresses these challenges.

4 Structure of Formal Proof

When we examine the informal proof sketched above, we can see that there are several distinct aspects of reasoning. Instead of intermingling these reasoning steps as in the proof sketch, we structure our formal proof to separate these components. As we will see, this decomposition is enabled by features of Lean. We believe that this proof structure applies more generally to other proofs of PAC learnability and related results in ML theory.

Specifically, we identify the following four components. We describe each briefly in the paragraphs below. The remaining sections of the paper then elaborate on each of these parts of the formal proof, after giving some background on measure theory in Section 5.

Specifying sources of randomness: As we saw in Section 2, there are two randomized scenarios under consideration in the statement of Theorem 2.1. First, there is the randomized choice of the training examples that are given as input to $\mathcal{A}$. These are sampled independently and from the same distribution μ. This first source of randomness is explicit since it appears in the statement of the probability appearing in Equation 5. The second kind of randomization is in the definition of Error. Recall that we defined Error in Equation 3 as the probability that an example randomly sampled from μ would be misclassified. This random sampling is entirely separate from the sampling of the training examples, although both samplings utilize the same μ.

Finally, the theorem statement is quantifying over the distribution μ. As we saw in the counterexample to the informal proof, inadvertently considering only certain classes of distributions (such as continuous ones) leads to erroneous arguments.

All of these details must be represented formally in the theorem prover. To handle these issues, we make use of the Giry monad [18] which allows us to represent sampling from distributions as monadic computations. Section 6 explains how this provides a convenient way to model the training and testing of a learning algorithm in order to formally state Theorem 2.1.

Deterministic properties of the algorithm: In the beginning of the proof sketch of Theorem 2.1 we started by noting certain *deterministic* properties of the learning algorithm $\mathcal{A}$, such as the fact that the threshold value $\hat{t}$ in the classifier returned by $\mathcal{A}$ must be $\leq$ the true unknown threshold t. These deterministic properties were the only details

about $\mathcal{A}$ upon which we relied when later establishing the Error bound. This means that an analogue of Theorem 2.1 will hold for any other stump learning algorithm with those properties.

As we will see, the Giry monad enables us to encode $\mathcal{A}$ as a purely functional Lean term that selects the maximum of the positively labeled training examples. This means we can prove these preliminary deterministic properties in the usual way one reasons about pure functions in Lean. The Lean statements of these deterministic properties are described in Section 7.

Measurability of maps and events: One detail missing from the informal proof was any consideration of *measurability* of functions and events. In measure-theoretic probability, probability spaces are equipped with a collection of *measurable sets*. We can only speak of the probability of an event if we show that the set corresponding to the event is *measurable*, meaning that it belongs to this collection. Similarly, random variables, such as the learning algorithm $\mathcal{A}$ itself, must be *measurable functions*.

While these facts are necessary for a rigorous proof, they risk cluttering a formal proof and obscuring all of the intuition that the informal proof gave. However, with Lean's typeclass mechanism and other proof automation, we can mostly separate the parts of the proof concerning measurability from the rest of the argument, as we describe in Section 8.

Quantitative reasoning about probabilities: The last step of the proof involves constructing the point θ described above and showing bounds on the probability that a sampled example lies in the interval $[\theta, t]$. Other than correcting the issue involved in the definition of θ, this stage of reasoning is similar to the proof style found in informal accounts of this result. Our goal is that this portion of the proof should resemble the kind of probabilistic reasoning that is familiar to experts in ML theory. This portion of the argument, and the final proof of Theorem 2.1 are described in Section 9.

5 Preliminaries

In this section we describe some basic background on measure-theoretic probability and how measure theory has been formalized in Lean as part of the `mathlib` library [38].

Measure theory. The starting point for probability theory is a set Ω called a *sample space*. Elements of Ω are called outcomes, and represent possible results of some randomized situation. For example, if the randomized situation is the roll of a six-sided die, we would have $\Omega = \{1, 2, 3, 4, 5, 6\}$. In naive probability theory, subsets of Ω are called events, and a probability function P on Ω is a function mapping events to real numbers in the interval $[0, 1]$, satisfying some axioms. While this naive approach works so long as Ω is a finite or countable set, attempting to assign probabilities to *all* subsets of Ω runs into technical difficulties when Ω is an uncountable set such as $\mathbb{R}$.

Measure-theoretic probability theory resolves this issue by only assigning probabilities to a collection $\mathcal{F}$ of subsets of Ω. The elements of $\mathcal{F}$ are called *measurable sets*. This collection $\mathcal{F}$ must be a *sigma-algebra*, which means that it must be closed under certain operations (e.g. taking countable unions). We call the pair $(\Omega, \mathcal{F})$ a *measurable space*. A probability measure μ is then a function of type $\mathcal{F} \to [0, 1]$ satisfying the following axioms:

- $\mu(\emptyset) = 0$
- $\mu(\Omega) = 1$
- If $A_1, A_2, \ldots$ is a countable collection of measurable sets such that $A_i \cap A_j = \emptyset$ for $i \neq j$, then

$$\mu\left(\bigcup_{i=1}^{\infty} A_i\right) = \sum_{i=1}^{\infty} \mu(A_i)$$

For the reader familiar with topology, the notion of a measurable space is analogous to the situation in topology, where a topological space is a pair $(X, \mathcal{V})$ where X is a set and $\mathcal{V}$ is a collection of subsets of X called the open sets, and $\mathcal{V}$ must be closed under various set operations. Indeed, for every topological space there is a minimal sigma-algebra containing all open sets, which is called the *Borel sigma-algebra*. We use the Borel sigma-algebra on the real numbers throughout the following.

A function $f : (\Omega_1, \mathcal{F}_1) \to (\Omega_2, \mathcal{F}_2)$ between two measurable spaces is said to be a *measurable function* if, for all $A \in \mathcal{F}_2$, we have $f^{-1}(A) \in \mathcal{F}_1$. Again, there is an analogy to topology, where a function between topological spaces is continuous if inverse images of open sets are open. In fact, if f is a continuous function between two topological spaces, then f is measurable when those spaces are equipped with their Borel sigma-algebras. Continuity implies that many standard arithmetic operations on the reals are measurable. Other examples which are measurable but not continuous include functions for testing whether a real number is $=$, $\leq$ or $\geq$ to some value.

The general measure theory in `mathlib` allows the measures of events to be greater than 1. To obtain just probability measures, we restrict these general definitions to require that if μ is a probability measure on X, then $\mu(X) = 1$. In Section 2 we subscripted the Pr notation to indicate the distributions that we were considering. In the context of a theorem prover, which obliges us to be precise in this manner, we forgo the Pr notation altogether. Instead, for the probability of an event E with respect to some measure μ, we simply write $\mu(E)$.

Above we have used the traditional mathematical notation of writing $f(x)$ for the application of a function f to an argument x. However, to more closely match notation in Lean, the sequel uses $f\ x$ when referring to definitions from our Lean development.

Typeclasses. In mathematical writing, we often associate a particular mathematical structure, such as a topology or sigma-algebra with a given set, with the convention that the structure should be used throughout. For example, when talking about continuous functions from $\mathbb{R} \to \mathbb{R}$, we do not constantly clarify that we mean continuous functions with respect to the topology generated by the Euclidean metric on $\mathbb{R}$.

This style of mathematical writing can be mimicked with Lean's typeclasses. After defining a typeclass, the user can declare instances of that typeclass, which associate a default structure with a given type. This mechanism is used throughout `mathlib` to supply default topologies, ring structures, and so on with particular types.

For example, the commands below first introduce the notation $\mathbb{H}$ to refer to the type `nnreal` of nonnegative real numbers from `mathlib`. We will use this notation when referring to the type of training examples and thresholds used for classification. After declaring this notation, an instance of `measurable_space` is defined on this type:

```
notation `ℍ` := nnreal
instance meas_ℍ: measurable_space ℍ := ...
```

where we have omitted the definition after the `:=` sign. After this instance is declared, any time we refer to $\mathbb{H}$ in a context where we need a sigma-algebra, this instance will be used.

The `mathlib` library comes with lemmas to automatically derive instances of `measurable_space` from other instances. For example, if a type has been associated with a topology, we can automatically derive the Borel sigma-algebra as an instance of `measurable_space` for that type. We use this Borel sigma-algebra on $\mathbb{H}$ above. Similarly, we can derive a product sigma-algebra on the product `A × B` of two types from existing instances for `A` and `B`, as in the following example:

```
instance meas_lbl: measurable_space (ℍ × bool)
```

Measurable spaces for stump training. At this point, considerations about the sigma-algebras with which a type is equipped introduce the first discrepancy between the informal set-up in Section 2 and our formalization. As we described there, it is common to treat the learning algorithm as if it returned a *function* of type $\mathbb{H} \to \{0, 1\}$, mapping examples to labels. Since one wants to speak about probabilities involving these classifiers, this means the type of classifiers must be equipped with a sigma-algebra. What sigma-algebra should be chosen? While there are canonical choices for types that have a topology (the Borel sigma-algebra) and for various operations on spaces such as products, there is no such standard choice for function types. In particular, the category of measurable spaces is not Cartesian closed [4]. Hence, there is no generic sigma-algebra on function types that would also make evaluation measurable. The textbook by Shalev-Shwartz and Ben-David points out that the PAC learnability framework requires the existence of a sigma-algebra on the class of hypotheses that makes classification measurable [35, Remark 3.1], but the *construction* of this sigma-algebra is not typically explained in examples.

Fortunately, the subset of decision stump classifiers has a simpler structure than the type of *all* functions from $\mathbb{H} \to \{0, 1\}$. In particular, the behavior of a decision stump classifier is entirely determined by the threshold used as a cut-off when assigning labels. These thresholds have type $\mathbb{H}$, which is equipped with the Borel sigma-algebra. In particular, given a threshold t, the function $\lambda x.\, \textsc{Label}(t, x)$ is measurable, and this is the evaluation function for a decision stump classifier. Thus, as we will see in the next section, we formalize the learning algorithm $\mathcal{A}$ such that it directly returns a threshold instead of a classifier. Similarly we adjust the definitions of Error (and associated functions) to take a threshold as input instead of a classifier.

A similar concern arises with how we represent the collection of training examples passed to the learning algorithm $\mathcal{A}$. In the earlier informal presentation, $\mathcal{A}$ takes a list of labeled examples as input. However, the construction of a sigma-algebra on variable-length lists is not commonly discussed in measure theory texts. We therefore work with dependently typed vectors of a specified length. Given a type `A` and natural `n`, the type `vec A n` represents vectors of size `n+1` of values of type `A`. When `A` is a topological space, `vec A n` can be given the $(n + 1)$-ary product topology, and we can then make it into a measurable space by equipping it with the Borel sigma-algebra.

6 Specifying Randomized Processes with the Giry Monad

Now that we have described some of the preliminaries of measure-theoretic probability, we turn to the question of how to formally represent the learning algorithm in the theorem prover.

In traditional probability theory, it is common to fix some sample space Ω and then work with a collection of *random variables* on this sample space. If V is a measurable space, a V-valued random variable is a measurable function of type $\Omega \to V$. One can think of the elements of the sample space Ω as some underlying source of randomness, and then the random variables encode how that randomness is transformed into an observable value. For example, Ω could be a sequence of random coin flips, and a random variable f might be a randomized algorithm that uses those coin flips.

In fact, at a certain point most treatments of probability theory start to leave the sample space Ω completely abstract. One simply postulates the existence of some Ω on which a collection of random variables with various distributions are said to exist. To ensure that the resulting theory is not vacuous, a theorem is proven to show that there exists an Ω

and a measure μ on Ω for which a suitably rich collection of random variables can be constructed.

While this pencil-and-paper approach could be used in formalization, it is inconvenient in several ways. First, while random variables are formally functions on the sample space, in practice we often treat them as if they were elements of their codomain. For example if X and Y are two real-valued random variables, then one writes $X+Y$ to mean the random variable $\lambda\omega.\, X(\omega) + Y(\omega)$. Similarly, if $f : \mathbb{R} \to \mathbb{R}$, we write $f(X)$ for the random variable $(\lambda\omega.\, f(X(\omega)))$. While this kind of convention is well understood on paper, trying to overload notations in a theorem prover to support it seems difficult.

The Giry monad [18] solves this problem by providing a syntactic sugar to describe stochastic procedures concisely.

6.1 Definition of the Giry Monad

The Giry monad is a triple (`Meas(.)`, `bind`, `ret`). For any measurable space X, `Meas`(X) is the space of probability measures over X, that is, functions from measurable subsets of X to $[0, 1]$ that satisfy the additional axioms of probability measures. The function `bind` is of type `Meas`$(X) \to (X \to$ `Meas`$(Y)) \to$ `Meas`(Y). That is, it takes a probability measure on X, a function that transforms values from X into probability measures over Y, and returns a probability measure on Y. The return function `ret` is of type $X \to$ `Meas`(X). It takes a value from X and returns a probability measure on X. The `mathlib` library defines this monad for general measures, which we then restrict to probability measures.

Functions `bind` and `ret` construct probability measures, so their definitions say what probability they assign to an event. Letting A be an event we define:

$$\texttt{bind}\ \mu\ f\ A = \int_{x \in X} f(x)(A) d\mu \tag{10}$$

$$\texttt{ret}\ x\ A = \begin{cases} 1 & \text{if } x \in A \\ 0 & \text{otherwise} \end{cases} \tag{11}$$

While we give the definitions here using standard mathematical notation, the formalization in `mathlib` uses the `Lean` definition of the integral. Here, `ret`(x) is the distribution that always returns x with probability 1. The definition of `bind`$\,\mu\, f$ corresponds to first sampling from μ to obtain some value x, and then continuing with the probability measure $f(x)$. Then, `bind` and `ret` satisfy the usual monad laws

$$\texttt{bind}\ (\texttt{ret}\ x)\ f = \texttt{ret}(f\ x) \tag{12}$$

$$\texttt{bind}\ \mu\ (\lambda x.\, \texttt{ret}\ x) = \mu \tag{13}$$

$$\texttt{bind}\ (\texttt{bind}\ \mu\ f)\ g = \texttt{bind}\ \mu\ (\lambda x.\texttt{bind}\ (f\ x)\ g) \tag{14}$$

However, these laws only hold when f and g are measurable functions. We will use the usual do-notation, writing do $x \leftarrow \mu\,;\ g(x)$ for `bind`$\,\mu\, g$.

As with any monad, we can define a function `map` which lifts a function $f : A \to B$, to a function of type `Meas`$(A) \to$ `Meas`(B). Given μ : `Meas`(A), we interpret `map`$\ f\ \mu$ as the probability distribution that first samples from μ to obtain a value of type A, and then applies f to it. Concretely, this is defined in terms of `bind` and `ret` as:

$$\texttt{map}\ f\ \mu = \text{do}\ x \leftarrow \mu\,;\ \texttt{ret}\ (f\ x) \tag{15}$$

As an example, we show how to construct a distribution that samples n independent times from a distribution μ and returns the result as a tuple. That is, we formally define the μ^n distribution that we used in Section 2. Let $(X, \mathcal{F})$ be a measurable space and $(X^n, \mathcal{F}^n)$ be the measurable space where X^n is the cartesian product of X (n times) and $\mathcal{F}^n$ is the product of $\mathcal{F}$ (n times). Let μ be a probability measure on $(X, \mathcal{F})$. Then, we define the measure μ^n recursively on n as

$$\mu^1 = \mu \tag{16}$$

$$\mu^n = \text{do}\ v \leftarrow \mu^{n-1}\,;\ \text{do}\ s \leftarrow \mu\,;\ \texttt{ret}(s, v) \tag{17}$$

Instead of nested tuples, as in the above, or lists of training examples, as we did in Section 2, we will use dependently typed vectors in `Lean`. The following term (definition omitted) gives the probability measure corresponding to taking `n+1` independent samples from a distribution and assembling them in a vector:

```
def vec.prob_measure
  (n : ℕ) (μ : probability_measure A)
  : probability_measure (vec A n) := ...
```

6.2 Modeling Stump Training and Testing

Let us now see how the decision stump training and testing procedures can be described using the Giry monad. `Lean` has a sectioning mechanism and a way to declare local variables. In our formalization, the lines below declare probability measures over the class of examples, and an arbitrary target threshold value for labeling samples:

```
variables (μ: probability_measure ℍ) (target: ℍ)
```

When we write definitions that use these variables, `Lean` will interpret the definition to treat these variables as if they were additional parameters on which the function depends.[5]

The following pure function labels a sample according to the target.

```
def label (target: ℍ): ℍ → ℍ × bool :=
    λ x: ℍ, (x,rle x target)
```

where `rle x target` returns true if `x` $\leq$ `target` and false otherwise.

Finally, we can define the event that an example is misclassified, and the Error function:

```
def error_set (h: ℍ) :=
  {x: ℍ | label h x ≠ label target x}

def error := λ h, μ (error_set h target)
```

[5] This is different from the behavior in Coq, where a definition that depends on section variables is only generalized to take additional arguments *outside* the section where the variables are declared.

The learning function $\mathcal{A}$ will take a vector of labeled examples as input and output a hypothesis. The function vec_map takes a function as an argument and applies it pointwise to the elements of the vector. We use this to label the inputs to the learning function:

```
def label_sample := vec_map (label target)
```

Our learning algorithm starts by transforming any negative example to 0 and then stripping off the labels, with the following function:

```
def filter :=
  vec_map (λ p, if p.snd then p.fst else 0)
```

This is safe since, if there were no positive examples, the learning algorithm should return 0 as we described in Section 2. Finally, we can define the learning function $\mathcal{A}$ that we will use. We call this function choose in the formalization. It selects the largest example after the above filtering process:

```
def choose (n: ℕ):vec (ℍ × bool) n → ℍ :=
  λ data: (vec (ℍ × bool) n), max n (filter n data)
```

We then use the Giry monad to describe the measure on classifiers that results from running the algorithm:

```
def denot: probability_measure ℍ :=
    let η := vec.prob_measure n μ  in
    let ν := map (label_sample target n) η in
    let γ := map (choose n) ν in
    γ
```

Note that map here is the monadic map operation defined in Equation 15, not vec_map. Thus, η is a distribution on training examples which is then transformed to ν, a distribution on labeled training examples. Then ν is transformed into a distribution γ on thresholds by lifting choose.

Finally, we have the following formal version of Theorem 2.1:

```
theorem choose_PAC:
    ∀ ϵ: nnreal, ∀ δ: nnreal, ∀ n: ℕ,
    ϵ > 0 → ϵ < 1 → δ > 0 → δ < 1 →
    n > (complexity ϵ δ) →
    (denot μ target n) {h: ℍ | error μ target h ≤ ϵ}
       ≥ 1 - δ
```

where complexity is the following function:

```
def complexity (ϵ: ℝ) (δ: ℝ) : ℝ :=
  (log(δ) / log(1 - ϵ)) - (1: nat)
```

The following sections outline how we prove this theorem in Lean.

7 Deterministic Reasoning

The overall proof builds on two simple assumptions that must be satisfied by the learning algorithm. First, the algorithm must return an estimate that is ≤ target. Formally,

```
lemma choose_property_1:
    ∀ n: ℕ, ∀ S: vec ℍ n,
    choose n (label_sample target n S) ≤ target
```

Our implementation satisfies this property since after the filter step in choose, all of the examples will have been mapped to a value ≤ target, and we then select the maximum value from the vector.

Second, the algorithm must return an estimate that is greater or equal to any positive example. This is because the proof uses the assumption that no examples lie in the region between the estimate and the target. Formally,

```
lemma choose_property_2:
    ∀ n: ℕ, ∀ S: vec ℍ n,
    ∀ i,
    ∀ p = kth_projn (label_sample target n S) i,
    p.snd = tt →
    p.fst ≤ choose n (label_sample target n S)
```

where kth_projn l i is an expression giving component i of the vector l. Our implementation satisfies this property because choose calls filter which leaves positive examples unchanged, and then we select the maximum from the filtered vector.

These proofs are trivial and account for only a very small fraction of the overall proof. Yet, these are the two specific properties of the algorithm that we need.

8 Measurability Considerations

About a quarter of the formalization consists in proving that various sets and functions are measurable. The predicate is_measurable S states that the set S is a measurable set, while measurable f states that the function f is measurable. These proofs can be long but are generally routine, with a few notable exceptions.

We will need to divide up the sample space into various intervals. If a and b are two reals, then we write Ioo a b, Ioc a b, Ico a b, Icc a b in Lean to refer to the intervals (a, b), $(a, b]$, $[a, b)$, and $[a, b]$, respectively. To start, we observe that the error of a hypothesis is the measure of the interval between it and the target.

```
lemma error_interval_1:
    ∀ h, h ≤ target →
    error μ target h = μ (Ioc h target)
lemma error_interval_2:
    ∀ h, h > target →
    error μ target h = μ (Ioc target h)
```

We next use these lemmas to prove that the function that computes the Error of a hypothesis is measurable:

```
lemma error_measurable:
    measurable (error μ target)
```

Proof. First, note that if A and B are measurable subsets such that $A \subseteq B$, then $\mu(B \setminus A) = \mu B - \mu A$. If $h \leq$ target, then

$$\begin{aligned} \text{error } \mu \text{ target h} &= \mu\,(h, \text{target}] \\ &= \mu\,[0, \text{target}] - \mu\,[0, h] \end{aligned}$$

Likewise, if $h >$ `target` then `error` μ `target h` $= \mu\,[0, h] - \mu\,[0,$ `target`$]$. Subtraction is measurable and testing whether a value is $\leq$ `target` is measurable. Therefore, since measurability is closed under composition, it suffices to show that the function $\lambda x.\mu\,[0, x]$ is Borel measurable. Because this function is monotone, its measurability is a standard result, though this fact was missing from `mathlib`. □

Next, one must show that the learning algorithm `choose` is measurable, after fixing the number of input examples:

```
lemma choose_measurable: measurable (choose n)
```

Proof. To prove that `choose` is a measurable function, we must prove that `max` is a measurable function. Because `max` is continuous, it is Borel measurable. □

Although the previous proof is straightforward, it hinges on the fact that the sigma-algebra structure we associate with `vec` $\mathbb{H}$ `n` is the Borel sigma-algebra. But, because we define a vector as an iterated product, another possible sigma-algebra structure for `vec` $\mathbb{H}$ `n` is the `n+1`-ary product sigma-algebra.

Recall from the previous section that our development uses Lean's typeclass mechanism to automatically associate product sigma-algebras with product spaces, and Borel sigma-algebras with topological spaces. As the preceding paragraph explains, for `vec` $\mathbb{H}$ `n` there are two possible choices. Which choice should be used? In programming languages with typeclasses, the problem of having to select between two potentially different instances of a typeclass is called a *coherence* problem [31]. Because of this ambiguity, `mathlib` is careful to only enable certain instances by default. Of course, this same potential ambiguity arises in normal mathematical writing, when we omit mentioning the associated sigma-algebra.

Fortunately, in the case of `vec` $\mathbb{H}$ `n`, these two sigma-algebras happen to be the same. In general, if X and Y are topological spaces with a *countable basis*, then the Borel sigma-algebra on $X \times Y$ is equal to the product of the Borel sigma-algebras on X and Y. The standard topology on the nonnegative reals has a countable basis, so the equivalence holds. Thus, although the proof of measurability for max can be simple, it uses a subtle fact that resolves the ambiguity involved in referring to sets without constantly mentioning their sigma-algebras.

9 Probabilistic Reasoning

The remainder of the proof involves the construction of the point θ and explicitly bounding the probability of various events.

Recall from the informal sketch in Section 3 that we first case split on whether $\Pr_{x\sim\mu}(x \leq$ `target`$) \leq \epsilon$ or not. In the language of measures, this is equivalent to a case split on whether $\mu\,[0,$ `target`$] \leq \epsilon$. In the formalization, it simplifies slightly the application of certain lemmas if we instead split on whether $\mu\,(0,$ `target`$] \leq \epsilon$. The following lemma is the key property in the case where the weight between 0 and `target` is $\leq \epsilon$. In that case, the learning algorithm always chooses a hypothesis with error at most ϵ.

```
lemma always_succeed:
    ∀ ϵ: nnreal, ϵ > 0 → ∀ n: ℕ,
    μ (Ioc 0 target) ≤ ϵ →
    ∀ S: vec ℍ n,
    error μ target
            (choose n (label_sample target n S))
    ≤ ϵ
```

Proof. By `error_interval_1`, we know that the error is going to be equal to the measure of the interval

```
(Ioc (choose n (label_sample target n S)) target)
```

Because we know `choose` must return a threshold between 0 and `target`, this interval is a subset of `(Ioc 0 target)`. Since measures are monotone, this means the measure of that interval must be $\leq \mu$ `(Ioc 0 target)`, which is $\leq \epsilon$ by assumption. □

For the case where $\mu\,(0,$ `target`$] > \epsilon$, the informal sketch selected a point θ such that $\mu\,[\theta,$ `target`$] = \epsilon$. However, as we saw in Counterexample 3.1, such a θ may not exist when μ is not continuous. Instead, we construct θ so that $\mu\,[\theta,$ `target`$] \geq \epsilon$, and $\mu\,(\theta,$ `target`$] \leq \epsilon$. The following theorem states the existence of such a point:

```
theorem extend_to_epsilon_1:
∀ ϵ: nnreal, ϵ > 0 →
μ (Ioc 0 target) > ϵ →
∃ θ: nnreal, μ (Icc θ target) ≥ ϵ ∧
             μ (Ioc θ target) ≤ ϵ
```

Proof. We take θ to be $\sup\{x \in \mathcal{X} \mid \mu\,[x,$ `target`$] \geq \epsilon\}$. The supremum exists because the set in question is bounded above by `target`, and the set is nonempty because it must contain 0 by our assumption that $\mu\,(0,$ `target`$] > \epsilon$. To see that $\mu\,[\theta,$ `target`$] \geq \epsilon$, we can construct an increasing sequence of points $x_n \leq \theta$ such that $\lim_{n\to\infty} x_n = \theta$, where for each n, $\mu\,[x_n,$ `target`$] \geq \epsilon$. We have then that:

$$\bigcap_i [x_i, \texttt{target}] = [\theta, \texttt{target}]$$

We use this in conjunction with the fact that measures are continuous from above, meaning that if $A_1, A_2, \ldots$ is a sequence of measurable sets such that $A_{i+1} \subseteq A_i$ for all i, then

$$\mu\left(\bigcap_{i=1}^{\infty} A_i\right) = \lim_{i\to\infty} \mu\, A_i$$

Hence we have

$$\begin{aligned}\mu\,[\theta, \texttt{target}] &= \mu\left(\bigcap_{i=1}^{\infty} [x_i, \texttt{target}]\right)\\ &= \lim_{n\to\infty} \mu\,[x_n, \texttt{target}]\\ &\geq \epsilon\end{aligned}$$

The proof that $\mu\,(\theta, \texttt{target}] \leq \epsilon$ is the dual argument, using continuity from below. □

The conclusion of this theorem states two inequalities involving θ. On the one hand, we need θ to be small enough that we can ensure at least one *training example* will lie between θ and `target`. On the other hand, we want θ to be large enough that if we *only* misclassify *test examples* that lie between θ and `target`, the error will nevertheless be at most ϵ.

The next two lemmas formalize these properties. Recall that `choose` maps all negative training examples to 0, leaves positive examples unchanged, and then takes the maximum of the resulting vector. The next lemma says that given a point θ such that $\mu\,[\theta, \texttt{target}] \geq \epsilon$, the measure of the event that an example gets mapped to something less than θ is at most $1 - \epsilon$.

```
lemma miss_prob:
    ∀ ϵ, ∀ θ: nnreal, θ > 0 →
    μ (Icc θ target) ≥ ϵ →
    μ {x : ℍ | ∀ a b,
         (a,b) = label target x →
         (if b then a else 0) < θ} ≤ 1 - ϵ
```

The next lemma shows why the property $\mu\,(\theta, \texttt{target}] \leq \epsilon$ is useful. In particular, it says that for such a θ, in order to have an error $> \epsilon$ on the hypothesis selected by `choose`, all training examples must get mapped to something less than θ. Formally, we say that the set of training samples which would lead to an error greater than ϵ, is a subset of those in which all the examples get mapped to a value less than θ.

```
lemma all_missed:
 ∀ ϵ: nnreal,
 ∀ θ: nnreal,
 μ (Ioc θ target) ≤ ϵ →
 {S | error μ target
              (choose n (label_sample target n S))
       > ϵ} ⊆
 {S |  ∀ i,
       ∀ p = label target (kth_projn S i),
       (if p.snd then p.fst else 0) < θ}
```

Finally, we prove a bound related to the `complexity` function, which computes the number of training examples needed:

```
lemma complexity_enough:
    ∀ ϵ: nnreal, ∀ δ: nnreal, ∀ n: ℕ,
    ϵ > (0: nnreal) → ϵ < (1: nnreal) →
    δ > (0: nnreal) → δ < (1: nnreal) →
   (n: ℝ) > (complexity ϵ δ) → ((1 - ϵ)^(n+1)) ≤ δ
```

Combining these lemmas together, we can finish the proof:

Proof of `choose_PAC`. We have seen that `always_succeed` handles the case $\mu\,(0, \texttt{target}] \leq \epsilon$. For the other case, where $\mu\,(0, \texttt{target}] > \epsilon$, we can apply `extend_to_epsilon_1` to get a θ with the specified properties. By `all_missed` we know that the event that the hypothesis selected has error $> \epsilon$ is a subset of the event where all the training examples get mapped to $< \theta$. Then, by `miss_prob` we know the probability that a given example gets mapped to $< \theta$ is $\leq 1 - \epsilon$. Because the training examples are selected independently, the probability that all $n + 1$ examples get mapped to a value $< \theta$ is at most $(1 - \epsilon)^{n+1}$. Applying `complexity_enough`, we have that $(1 - \epsilon)^{n+1} \leq \delta$, so we are done. □

10 Related Work

Classic results about the average case behavior of quicksort and binary search trees have been formalized by a number of authors using different proof assistants [17, 37, 40]. In each case, the authors write down the algorithm to be analyzed using a variant of the monadic style we discuss in Section 6. Gopinathan and Sergey [19] verify the error rate of Bloom Filters and variants. Affeldt et al. [2] formalize results from information theory about lossy encoding. For the most part, these formalizations only use discrete probability theory, with the exception of Eberl et al.'s analysis of treaps [17], which requires general measure-theoretic probability. They report that dealing with measurability issues adds some overhead compared to pencil-and-paper reasoning, though they are able to automate many of these proofs.

Several projects have formalized results from cryptography, which also involves probabilistic reasoning [8, 9, 12, 30]. A challenge in formalizing such proofs lies in the need to establish a relation between the behavior of two different randomized algorithms, as part of the game-playing approach to cryptographic security proofs. Because cryptographic proofs generally only use discrete probability theory, these libraries do not formalize measure-theoretic results. There are many connections between cryptography and learning theory [32], which would be interesting to formalize.

There have been formalizations of measure-theoretic probability theory in a few proof assistants. Hurd [23] formalized basic measure theory in the HOL proof assistant, including a proof of Caratheodory's extension theorem. Hölzl and Heller [21] developed a more substantial library in the Isabelle theorem prover, which has since been extended further. Avigad et al. [5] used this library to formalize a proof of the Central Limit Theorem. Several measure theory libraries have also been developed in Coq [1, 27, 36]. The ALEA library [3] instead uses a synthetic approach to discrete probability in Coq, a technique that has subsequently been extended to continuous probabilities by Bidlingmaier et al. [11].

More recent work has formalized theoretical machine learning results. Selsam et al. [34] use Lean to prove the correctness of an optimization procedure for stochastic computation graphs. They prove that the random gradients used in their stochastic backpropagation implementation are unbiased. In their proof, they add axioms to the system for various basic mathematical facts. They argue that even if

there are errors in these axioms that could potentially lead to inconsistency, the process of constructing formal proofs for the rest of the algorithm still helps eliminate mistakes.

Bagnall and Stewart [6] use Coq to give machine-checked proofs of bounds on generalization errors. They use Hoeffding's inequality to obtain bounds on error when the hypothesis space is finite or there is a separate test-set on which to evaluate a classifier after training. They apply this result to bound the generalization error of ReLU neural networks with quantized weights. Their proof is restricted to discrete distributions and adds some results from probability theory as axioms (Pinsker's inequality and Gibbs' inequality).

Bentkamp et al. [10] use Isabelle/HOL to formalize a result by Cohen et al. [14], which shows that deep convolutional arithmetic circuits are more expressive than shallow ones, in the sense that shallow networks must be exponentially larger in order to express the same function. Although convolutional arithmetic circuits are not widely used in practice compared to other artificial neural networks, this result is part of an effort to understand theoretically the success of deep learning. Bentkamp et al. report that they proved a stronger version of the original result, and doing so allowed them to structure the formal proof in a more modular way. The formalization was completed only 14 months after the original arXiv posting by Cohen et al., suggesting that once the right libraries are available for a theorem prover, it is feasible to mechanize state of the art results in some areas of theoretical machine learning in a relatively brief period of time.

After the development described by our paper was publicly released, Zinkevich [42] published a `Lean` library for probability theory and theoretical machine learning. Among other results, this library contains theorems about PAC learnability when the class of hypotheses is finite. Because the decision stump hypothesis class is the set of all nonnegative real numbers, our result is not covered by these theorems.

A related but distinct line of work applies machine learning techniques to automatically construct formal proofs of theorems. Traditional approaches to automated theorem proving rely on a mixture of heuristics and specialized algorithms for decidable sub-problems. By using a pre-existing corpus of formal proofs, supervised learning algorithms can be trained to select hypotheses and construct proofs in a formal system [7, 22, 24–26, 33].

11 Conclusion

We have presented a machine-checked, formal proof of PAC learnability of the concept class of decision stumps. The proof is formalized using the `Lean` theorem prover. We used the Giry monad to keep the formalization simple and close to a pencil-and-paper proof. To formalize this proof, we specialized the measure theory formalization of the `mathlib` library to the necessary basic probability theory. As expected, the formalization is at times subtle when we must consider topological or measurability results, mostly to prove that the learning algorithm and Error are measurable functions. The most technical part of the proof has to do with proving the existence of an interval with the appropriate measure, a detail that standard textbook proofs either omit or get wrong.

Our work shows that the `Lean` prover and the `mathlib` library are mature enough to tackle a simple but classic result in statistical learning theory. A next step would be to formally prove more general results from VC-dimension theory. In addition, there exist a number of generalizations of PAC learnability, such as *agnostic* PAC learnability, which removes the assumption that some hypothesis in the class perfectly classifies the examples. Other generalizations allow for more than two classification labels and different kinds of Error functions. It would be interesting to formalize these various extensions and some related applications.

Acknowledgments

We thank Gordon Stewart for comments on a previous draft of the paper. We thank the anonymous reviewers from the CPP'21 PC for their feedback. Some of the work described in this paper was performed while Koundinya Vajjha was an intern at Oracle Labs. Vajjha was additionally supported by the Alfred P. Sloan Foundation under grant number G-2018-10067. Banerjee's research was based on work supported by the US National Science Foundation (NSF), while working at the Foundation. Any opinions, findings, and conclusions or recommendations expressed in the material are those of the authors and do not necessarily reflect the views of the NSF.

References

[1] Reynald Affeldt, Cyril Cohen, Marie Kerjean, Assia Mahboubi, Damien Rouhling, Kazuhiko Sakaguchi, and Pierre-Yves Strub. 2020. mathcomp Analysis Library. https://github.com/math-comp/analysis.

[2] Reynald Affeldt, Manabu Hagiwara, and Jonas Sénizergues. 2014. Formalization of Shannon's Theorems. *J. Autom. Reason.* 53, 1 (2014), 63–103.

[3] Philippe Audebaud and Christine Paulin-Mohring. 2009. Proofs of randomized algorithms in Coq. *Sci. Comput. Program.* 74, 8 (2009), 568–589.

[4] Robert J. Aumann. 1961. Borel structures for function spaces. *Illinois J. Math.* 5, 4 (12 1961), 614–630.

[5] Jeremy Avigad, Johannes Hölzl, and Luke Serafin. 2017. A Formally Verified Proof of the Central Limit Theorem. *J. Autom. Reason.* 59, 4 (2017), 389–423.

[6] Alexander Bagnall and Gordon Stewart. 2019. Certifying the True Error: Machine Learning in Coq with Verified Generalization Guarantees. In *AAAI'19: The Thirty-Third AAAI Conference on Artificial Intelligence*. 2662–2669.

[7] Kshitij Bansal, Sarah M. Loos, Markus N. Rabe, Christian Szegedy, and Stewart Wilcox. 2019. HOList: An Environment for Machine Learning of Higher Order Logic Theorem Proving. In *Thirty-sixth International Conference on Machine Learning (ICML)*. 454–463.

[8] Gilles Barthe, François Dupressoir, Benjamin Grégoire, César Kunz, Benedikt Schmidt, and Pierre-Yves Strub. 2013. EasyCrypt: A Tutorial. In *Foundations of Security Analysis and Design VII - FOSAD 2012/2013 Tutorial Lectures*. 146–166.

[9] Gilles Barthe, Benjamin Grégoire, and Santiago Zanella Béguelin. 2009. Formal certification of code-based cryptographic proofs. In *POPL*. 90–101.

[10] Alexander Bentkamp, Jasmin Christian Blanchette, and Dietrich Klakow. 2019. A formal proof of the expressiveness of deep learning. *Journal of Automated Reasoning* 63, 2 (2019), 347–368.

[11] Martin E. Bidlingmaier, Florian Faissole, and Bas Spitters. 2019. Synthetic topology in Homotopy Type Theory for probabilistic programming. *CoRR* abs/1912.07339 (2019). arXiv:1912.07339 http://arxiv.org/abs/1912.07339

[12] Bruno Blanchet. 2006. A Computationally Sound Mechanized Prover for Security Protocols. In *2006 IEEE Symposium on Security and Privacy*. 140–154.

[13] Anselm Blumer, Andrzej Ehrenfeucht, David Haussler, and Manfred K Warmuth. 1989. Learnability and the Vapnik-Chervonenkis dimension. *Journal of the ACM (JACM)* 36, 4 (1989), 929–965.

[14] Nadav Cohen, Or Sharir, and Amnon Shashua. 2016. On the Expressive Power of Deep Learning: A Tensor Analysis. In *Proceedings of the 29th Conference on Learning Theory, COLT 2016*. 698–728.

[15] Leonardo Mendonça de Moura, Soonho Kong, Jeremy Avigad, Floris van Doorn, and Jakob von Raumer. 2015. The Lean Theorem Prover (System Description). In *CADE-25 - 25th International Conference on Automated Deduction*. 378–388.

[16] R. M. Dudley. 2014. *Uniform Central Limit Theorems* (2nd ed.). Cambridge University Press.

[17] Manuel Eberl, Max W. Haslbeck, and Tobias Nipkow. 2018. Verified Analysis of Random Trees. In *ITP*. 196–214.

[18] Michèle Giry. 1982. A Categorical Approach to Probability Theory. In *Categorical Aspects of Topology and Analysis (Lecture Notes in Mathematics, Vol. 915)*, B. Banaschewski (Ed.). 68–85.

[19] Kiran Gopinathan and Ilya Sergey. 2020. Certifying Certainty and Uncertainty in Approximate Membership Query Structures. In *CAV*, Shuvendu K. Lahiri and Chao Wang (Eds.). 279–303.

[20] Johannes Hölzl. 2013. *Construction and stochastic applications of measure spaces in higher-order logic*. Ph.D. Dissertation. Technical University Munich.

[21] Johannes Hölzl and Armin Heller. 2011. Three Chapters of Measure Theory in Isabelle/HOL. In *ITP*. 135–151.

[22] Daniel Huang, Prafulla Dhariwal, Dawn Song, and Ilya Sutskever. 2019. GamePad: A Learning Environment for Theorem Proving. In *7th International Conference on Learning Representations, ICLR 2019*.

[23] Joe Hurd. 2003. *Formal Verification of Probabilistic Algorithms*. Ph.D. Dissertation. Cambridge University.

[24] Jan Jakubuv and Josef Urban. 2019. Hammering Mizar by Learning Clause Guidance (Short Paper). In *ITP*. 34:1–34:8.

[25] Cezary Kaliszyk, François Chollet, and Christian Szegedy. 2017. HolStep: A Machine Learning Dataset for Higher-order Logic Theorem Proving. In *5th International Conference on Learning Representations, ICLR 2017*.

[26] Cezary Kaliszyk, Josef Urban, Henryk Michalewski, and Miroslav Olšák. 2018. Reinforcement Learning of Theorem Proving. In *NeurIPS*. 8836–8847.

[27] Robert Kam. 2008. coq-markov Library. https://github.com/coq-contribs/markov.

[28] Michael J Kearns and Umesh Virkumar Vazirani. 1994. *An Introduction to Computational Learning Theory*. MIT press.

[29] Mehryar Mohri, Afshin Rostamizadeh, and Ameet Talwalkar. 2018. *Foundations of Machine Learning*. MIT press.

[30] Adam Petcher and Greg Morrisett. 2015. The Foundational Cryptography Framework. In *POST*. 53–72.

[31] Simon Peyton Jones, Mark Jones, and Erik Meijer. 1997. Type classes: an exploration of the design space. In *Haskell Workshop*.

[32] Ronald L. Rivest. 1991. Cryptography and Machine Learning. In *Advances in Cryptology - ASIACRYPT '91*. 427–439.

[33] Daniel Selsam and Nikolaj Bjørner. 2019. Guiding High-Performance SAT Solvers with Unsat-Core Predictions. In *Theory and Applications of Satisfiability Testing - SAT 2019*. 336–353.

[34] Daniel Selsam, Percy Liang, and David Dill. 2017. Developing Bug-Free Machine Learning Systems With Formal Mathematics. In *International Conference on Machine Learning (ICML)*. 3047–3056.

[35] Shai Shalev-Shwartz and Shai Ben-David. 2014. *Understanding Machine Learning: From Theory to Algorithms*. Cambridge University Press.

[36] Joseph Tassarotti. 2020. coq-proba Probability Library. https://github.com/jtassarotti/coq-proba.

[37] Joseph Tassarotti and Robert Harper. 2018. Verified Tail Bounds for Randomized Programs. In *ITP*. 560–578.

[38] The mathlib Community. 2020. The Lean Mathematical Library. In *CPP*. 367–381.

[39] Leslie G. Valiant. 1984. A Theory of the Learnable. *Commun. ACM* 27, 11 (1984), 1134–1142.

[40] Eelis van der Weegen and James McKinna. 2008. A Machine-Checked Proof of the Average-Case Complexity of Quicksort in Coq. In *TYPES*. 256–271.

[41] Vladimir Naumovich Vapnik. 2000. *The Nature of Statistical Learning Theory, Second Edition*. Springer.

[42] Martin Zinkevich. 2020. https://github.com/google/formal-ml