

COS 511: Theoretical Machine Learning

Lecturer: Rob Schapire
Scribe: Nader Al-Naji

Lecture #2
February 7, 2013

1 Quick Review

We would like to explore the consistency model in a little more depth but before we do so, let's have a quick review of some salient points.

1.1 Concept Class

To make things as simple as possible, we will often assume that only two labels are possible, call them 0 and 1. We will also make the simplifying assumption that there is a mapping from examples to labels. This mapping is called a *concept*. Thus, a concept is a function of the form $c : X \rightarrow \{0, 1\}$ where X is the space of all possible examples called the *domain* or *instance space*. A collection of concepts is called a *concept class*. We will often assume that the examples have been labeled by an unknown concept from a *known* concept class.

1.2 The Consistency Model

We say that a concept class \mathcal{C} is learnable in the consistency model if there is an algorithm A which, when given any set of labeled examples $(x_1, y_1), \dots, (x_m, y_m)$, where $x_i \in X$ and $y_i \in \{0, 1\}$, finds a concept $c \in \mathcal{C}$ that is consistent with the examples (so that $c(x_i) = y_i$ for all i), or says (correctly) that there is no such concept. Moreover, we are especially interested in finding efficient algorithms in this model.

2 Examples of Learning Under Consistency Model

In this section we cover a host of concrete examples and discuss their learnability under the consistency model. We will find that while the consistency model works well in some cases, it leaves much to be desired in others. This will be our motivation for developing a more sophisticated learning model later on.

2.1 Monotone Conjunctions

This example was discussed before but we include it here for completeness.

Suppose the domain is $X = \{0, 1\}^n$, the set of all n -bit vectors, that is, vectors of the form $\mathbf{x} = (x_1, \dots, x_n)$ where each variable $x_j \in \{0, 1\}$. Let the concept class \mathcal{C} consist of all monotone conjunctions, that is, the AND of a subset of the (unnegated) variables, such as

$c(\mathbf{x}) = x_2 \wedge x_7 \wedge x_{10}$. Our data then looks like a set of bit vectors with labels, for instance:

01101	+
11011	+
11001	+
00101	-
11000	-

Given such data, we can learn this class in the consistency model by taking the bitwise AND of all of the positive examples, then form a conjunction of all variables corresponding to the bits that are still on. For instance, for the data above, the bitwise AND of the positive examples gives 01001, i.e., the conjunction $x_2 \wedge x_5$. This procedure clearly gives a conjunction c that is consistent with the positive examples. Moreover, by construction, any other conjunction c' that is consistent with the positive examples must contain a subset of the variables in c , so if c' is consistent with all the examples (both positive and negative), then c must be as well.

2.2 Conjunctions

Suppose that as before the domain is $X = \{0, 1\}^n$, the set of all n -bit vectors, that is, vectors of the form $\mathbf{x} = (x_1, \dots, x_n)$ where each variable $x_j \in \{0, 1\}$. But now, let the concept class \mathcal{C} consist of all conjunctions (not just monotone conjunctions), that is, the AND of a subset of the (possibly negated) variables, such as $c(\mathbf{x}) = x_2 \wedge \bar{x}_7 \wedge \bar{x}_{10}$ where \bar{x} denotes the negation of a variable x .

Our data, of course, will look just as it did before, but our learning algorithm will need to change. Since we already know how to learn when our concept class consists of *monotone* conjunctions, the best thing to do would seem to be to reduce the problem of learning conjunctions to the problem of learning *monotone* conjunctions and just use our previous algorithm. We can do this in the following way. First, for each variable x_i , introduce a new variable $z_i = \bar{x}_i$ representing its negation. Recall that we've assumed that we're dealing with n -bit vectors where x_i represents the i^{th} bit of a particular vector and z_i represents the *negation* of the i^{th} bit of a particular vector. Then, to get our examples into a form our monotone conjunction learning algorithm can handle, we extend each of our n -bit examples to $2n$ -bit examples by concatenating each example with its *negation*. For instance, if one of our examples were 11001, our new example would be 11001 concatenated with 00110, or simply 1100100110. We then have (x_1, \dots, x_n) represent bits $(1, \dots, n)$ (as before) and have (z_1, \dots, z_n) represent bits $(n + 1, \dots, 2n)$. As an example, the following set of bit vectors:

01101	+
11101	+
11100	+
01111	-
11000	-

would become:

$$\begin{array}{rcl}
 0110110010 & + & \\
 1110100010 & + & \\
 1110000011 & + & \\
 0111110000 & - & \\
 1100000111 & - &
 \end{array}$$

and applying our monotone conjunction learning algorithm would yield $x_2 \wedge x_3 \wedge z_4 = x_2 \wedge x_3 \wedge \bar{x}_4$. Note that no consistent *monotone* conjunction exists for these examples.

Applying the monotone conjunction algorithm to the extended examples results in a conjunction c that is consistent with the positive (unextended) examples. To see this, simply note that in order for the monotone conjunction algorithm to include z_i , the i^{th} bit of all of the positive examples must be zero, making \bar{x}_i 's inclusion in c not affect c 's consistency. Further, by construction, any other conjunction c' that is consistent with the positive examples must contain a subset of the variables in c , so if c' is consistent with all the examples (both positive and negative), then c must be as well.

2.3 Monotone Disjunctions

Suppose that as before the domain is $X = \{0, 1\}^n$, the set of all n -bit vectors, that is, vectors of the form $\mathbf{x} = (x_1, \dots, x_n)$ where each variable $x_j \in \{0, 1\}$. But now, let the concept class \mathcal{C} consist of all monotone *disjunctions*, that is, the OR of a subset of the (unnegated) variables, such as $c(\mathbf{x}) = x_2 \vee x_7 \vee x_{10}$.

Once again, our data will remain the same but our algorithm will need to change. As before, we will reduce the problem of learning a monotone disjunction to the problem of learning a monotone *conjunction*, which we know how to do. To do this, we need to recall one simple fact from logic:

$$(x_1 \vee \dots \vee x_n) = \overline{(\bar{x}_1 \wedge \dots \wedge \bar{x}_n)} \tag{1}$$

where \bar{x} again denotes the negation of a variable x . This is DeMorgan's Law and we can use it to immediately reduce the monotone disjunction problem to the monotone conjunction problem. Clearly, if we flip all of the bits in our input data and flip all of our labels, and apply our monotone conjunction algorithm, we will find a concept c such that if we negate all of the literals in c and then negate the conjunction itself, we will end up with a new (negated) conjunction c' that is consistent with the original data. Then, by DeMorgan's law, if c' is consistent with the original data, then the *disjunction* of all the *unnegated* literals in c' , call this c'' , will also be consistent with the original data. Thus c'' will be the monotone disjunction we're looking for. Further, it should also be clear that if the monotone conjunction algorithm fails to find a concept c consistent with the flipped data, then no consistent monotone *disjunction* exists for the original data, completing our reduction.

2.4 k -CNF Formulas

Suppose that as before the domain is $X = \{0, 1\}^n$, the set of all n -bit vectors, that is, vectors of the form $\mathbf{x} = (x_1, \dots, x_n)$ where each variable $x_j \in \{0, 1\}$. But now, let the concept class \mathcal{C}

consist of all k -CNF Formulas, that is, conjunctions of disjunctions (called “clauses”) where each disjunction has at most k literals. We usually think of k as a small constant; this means that an $O(n^k)$ algorithm (where n is the input size) is thought of as polynomial rather than exponential. As an example, with $k = 2$, we could have: $c(\mathbf{x}) = (x_2 \vee x_7) \wedge (x_{10}) \wedge (x_4 \vee \bar{x}_5)$.

How do we learn concepts in this class? Once again, we will reduce to monotone conjunction. We can do this by creating a new variable for every possible clause (disjunction) that could appear in our k -CNF formula. For example, if we have data in which each example is two bits long and we are trying to find a consistent 2-CNF, we create the following variables:

$$\begin{aligned} z_1 &= (x_1) \\ z_2 &= (x_2) \\ z_3 &= (x_1 \vee x_2) \\ z_4 &= (x_1 \vee \bar{x}_2) \\ z_5 &= (\bar{x}_1 \vee x_2) \\ z_6 &= (\bar{x}_1 \vee \bar{x}_2) \end{aligned}$$

Using the example above to describe our reduction, we would then convert our two-bit examples into six-bit examples where each bit i is 1 if z_i is true for that example and 0 otherwise. This could then be fed to our monotone conjunction algorithm, which would result in a conjunction c of z_i 's consistent with the data. Converting these z_i 's into their corresponding disjunctions would yield a consistent 2-CNF, c' . It should be clear that the process as described for this two-bit example can be easily extended to handle the task of finding consistent k -CNF formulas with n -bit input data. In the general case, we would simply have more z_i variables each corresponding to some permutation of the n (instead of just 2) variables and their negations. It should also be clear that if the algorithm described fails to find a consistent k -CNF then no consistent k -CNF can exist, thus completing our reduction.

One thing to note is that the number of z_i variables we must create depends on k and on n . In particular, if we are trying to learn a consistent k -CNF for n -bit strings, we must consider $O((2n)^k)$ possible z_i variables. To see this, simply note that the number of k -CNF's one can generate with n variables is $(2n)(2n-1)\dots(2n-k) = O((2n)^k)$ because each position in the k -CNF has $2n$ possible choices of variables, all the of x_i and their negations. The algorithm is thus efficient (polynomial time) if we assume k to be a small constant but not otherwise. In particular, if we allow k to be arbitrarily large the algorithm will be exponential.

2.5 Axis-aligned Rectangles

Suppose now that our domain consists of points in 2-D space. That is, $X = \mathbb{R}^2$, or points of the form $\mathbf{x} = (x_1, x_2)$ where each variable $x_j \in \mathbb{R}$. Now, let the concept class \mathcal{C} consist of all 2-D axis-aligned rectangles, or to be precise all functions mapping points in a particular 2-D axis-aligned rectangle to $\{0, 1\}$. As an example, one concept $c \in \mathcal{C}$ could map all points *inside* the rectangle with lower-left and upper-right corners at $(0, 0)$ and $(1, 1)$ respectively to 1 and all points *outside* this rectangle to 0. An illustration of this is shown below.

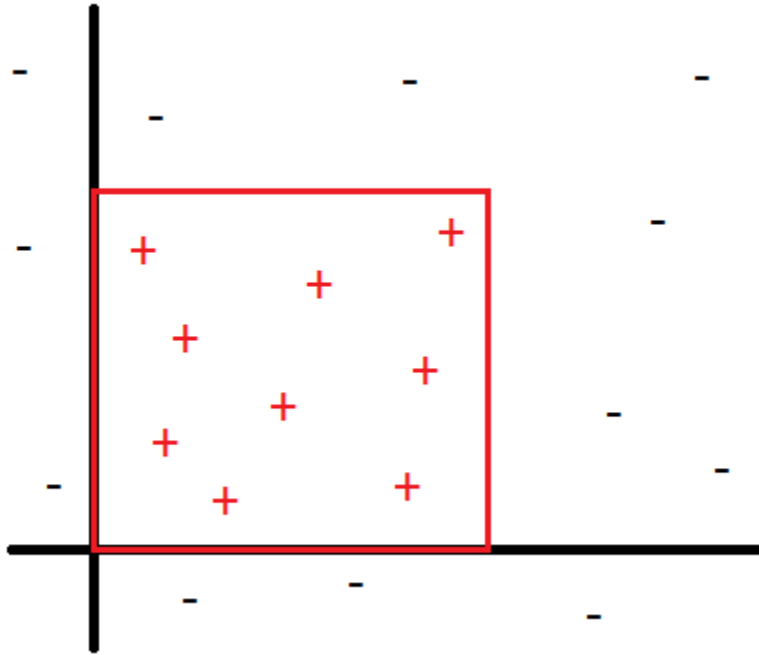


Figure 1: A concept that maps points inside the rectangle defined by $(0,0)$ and $(1,1)$ to $+$ and points outside this rectangle to 0 .

The goal is then to take input examples $\langle (\mathbf{x}_1, y_1), \dots, (\mathbf{x}_m, y_m) \rangle$ consisting of points, \mathbf{x}_i , and labels, y_i , and to then find a rectangle that is consistent with these points. Namely, a rectangle that contains all of the \mathbf{x}_i that have $y_i = 1$ without containing any of the \mathbf{x}_i that have $y_i = 0$.

In order to find such a rectangle, we simply need to compute x_{min} the smallest x coordinate of all of the positive examples, x_{max} the largest x coordinate of all the positive examples, y_{min} the smallest y coordinate of all the positive examples, and y_{max} the largest y coordinate of all the positive examples. Our rectangle, c , will then simply be the rectangle defined by (x_{min}, y_{min}) and (x_{max}, y_{max}) . This will be the *smallest* rectangle that can possibly contain all the positive examples and so if any negative examples are contained inside, we safely conclude that no consistent rectangle exists (because any rectangle that contains all the positive examples must include the smallest rectangle as a subset). Otherwise, if all the negative examples are outside this rectangle, we conclude that this rectangle is consistent.

2.6 Half Hyperspaces

Suppose now that our domain consists of points in n -dimensional space. That is, $X = \mathbb{R}^n$, or points of the form $\mathbf{x} = (x_1, \dots, x_n)$ where each variable $x_j \in \mathbb{R}$. Now, let the concept class \mathcal{C} consist of all *half hyperspaces* or, equivalently, all *linear threshold functions*. As an example, if we are dealing with 2-D points, one concept $c \in \mathcal{C}$ could map all points *above*

some threshold line to 1 and all points *below* this threshold to 0. An illustration of this is shown below. The goal is then to take as input a set of examples $\langle (\mathbf{x}_1, y_1), \dots, (\mathbf{x}_m, y_m) \rangle$

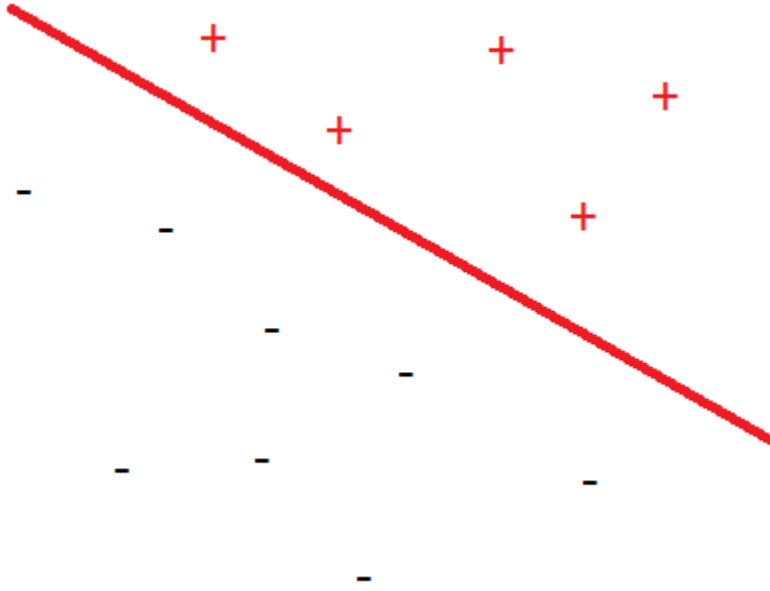


Figure 2: A concept that maps points in 2-D space above some threshold line to 1 and points below this threshold line to 0.

consisting of n -dimensional points, \mathbf{x}_i , and labels, y_i , and then to find a hyperplane separating the examples with $y_i = 1$ from the examples with $y_i = 0$.

In other words, we want a weight vector \mathbf{w} and some threshold b such that $\mathbf{w} \cdot \mathbf{x}_i > b$ if $y_i = 1$ and $\mathbf{w} \cdot \mathbf{x}_i \leq b$ if $y_i = 0$. Since the \mathbf{x}_i are all known, this results in a simple linear program, which we can solve efficiently using any available linear programming package.

2.7 2-Term DNF

Suppose that the domain is $X = \{0, 1\}^n$, the set of all n -bit vectors, that is, vectors of the form $\mathbf{x} = (x_1, \dots, x_n)$ where each variable $x_j \in \{0, 1\}$. Let the concept class \mathcal{C} consist of all *2-term DNF's*, that is, the OR of two arbitrary length conjunctions. As an example, we could have $c(\mathbf{x}) = (x_2 \wedge x_7 \wedge x_{10}) \vee (x_1 \wedge x_5)$.

Now, how do we learn a 2-term DNF given input data? If we can reduce the problem of finding a 2-Term DNF to the problem of finding a k -CNF, we'll be done so let's try that. First, we recall some basic rules of logic. Informally, we recall that disjunction can be treated as "multiplication" and conjunction can be treated as "addition", for example: $(x_1 \wedge x_2) \vee (x_3 \wedge x_4) = (x_1 \vee x_3) \wedge (x_1 \vee x_4) \wedge (x_2 \vee x_3) \wedge (x_2 \vee x_4)$. This implies that we can always convert 2-term DNF's into 2-CNF's, but does this mean the class is learnable? Well, we can always run our 2-CNF learning algorithm and find a consistent 2-CNF, and if we can always convert this 2-CNF into a 2-term DNF then we're good. The problem is that, while we can always convert a 2-term DNF into a 2-CNF, we can't necessarily go the

other direction. That is, even if we find a consistent 2-CNF, this doesn't guarantee that there exists a consistent 2-term DNF. Informally, the issue is that the space of 2-CNF's is a superset of the space of 2-term DNF's, or: (2-term DNF) \subseteq (2-CNF). As it turns out, in spite of the fact that learning 2-CNF's is easy, learning 2-term DNF's is NP-hard.

Here we arrive at a fundamental problem with our consistency model. It is possible for a class \mathcal{C} to be learnable but to have a subclass of \mathcal{C} be unlearnable.

2.8 DNF

Suppose that the domain is $X = \{0, 1\}^n$, the set of all n -bit vectors, that is, vectors of the form $\mathbf{x} = (x_1, \dots, x_n)$ where each variable $x_j \in \{0, 1\}$. Let the concept class \mathcal{C} consist of all DNF's, that is, the OR of an arbitrary number of arbitrary-length conjunctions. As an example, we could have $c(\mathbf{x}) = (x_2 \wedge x_7 \wedge x_{10}) \vee (x_1 \wedge x_5) \vee (x_6 \wedge x_3 \wedge x_4 \wedge x_8)$.

Given input data, finding a consistent DNF is fairly simple. Indeed, all one needs to do is construct a clause for each positive example with a literal corresponding to the truth value of each bit. As an example, consider the following examples as input:

$$\begin{array}{ll} 01101 & + (\bar{x}_1 \wedge x_2 \wedge x_3 \wedge \bar{x}_4 \wedge x_5) \vee \\ 11101 & + (x_1 \wedge x_2 \wedge x_3 \wedge \bar{x}_4 \wedge x_5) \vee \\ 11100 & + (x_1 \wedge x_2 \wedge x_3 \wedge \bar{x}_4 \wedge \bar{x}_5) \\ 01111 & - \\ 11000 & - \end{array}$$

For the positive examples, the DNF has x_i as true if the i^{th} bit is 1 and x_i as false otherwise. It should be clear that constructing a DNF in this way works in general.

Clearly this method is effective and efficient at learning a DNF that is consistent with the input. But what is this DNF even useful for? The DNF will always perfectly fit the data it is trained on; however, in spite of this it will always label examples it *hasn't* seen before as negative, suggesting that the algorithm will perform very poorly when presented with new test examples. Not only that, but the size of the DNF is exactly the size of the positive input data; it is as if the DNF is more a representation of the input data than a representation of some learned structure in the data. This example highlights a distinction between *memorization* (training that involves closely or exactly recording a small number of examples) and *generalization* (training with the intention of minimizing error on future unseen examples) that will be discussed later on.

3 Problems With the Consistency Model

From our examples, we've already seen a few shortcomings of the consistency model. In our 2-term DNF example, we saw that a class \mathcal{C} can be learnable while a subclass of \mathcal{C} can be unlearnable. Not only that, but from our DNF example, we saw that the consistency model yields a concept that tells us nothing about the *accuracy* of the model on *new data*. That is, because the only criterion the consistency model considers is how well a concept fits the training data, it misses a fundamental aspect of learning: predicting labels on new data.

Indeed, there doesn't appear to be *any* direct connection to learning in the case of DNF's. Finally, the consistency model has a practical problem in that training data that contains noise is not handled in a robust way. That is, if presented with noisy data, a concept that could be learned under a more robust model will be overlooked by the consistency model. We endeavor to fix these problems in what follows.

4 Probability Review

In order to revise our learning model, we first need an understanding of basic probability. We don't need much, just a refresher of a few definitions.

4.1 Definitions

- **Event:** An outcome to which a probability (real number denoting the likelihood of the outcome occurring) is assigned. An event is something that either happens or not.
- **Random Variable:** A variable that takes on values in a probabilistic way. Usually denoted by a capital letter.
- **Distribution:** Specifies the likelihood that a random variable takes on particular values. Usually use $Pr[X = x]$ to denote the probability the random variable X takes on the value x . Must specify $Pr[X = x]$ for all x and it must be the case that $\sum_x Pr[X = x] = 1$. $X \sim D$ means “ X is chosen randomly from distribution D .”
- **Expectation:** Expected value of a random variable is denoted by $E[X] = \sum_x x \cdot Pr[X = x]$. Can also take the expected value of a function of a random variable $E[f(X)] = \sum_x f(x) \cdot Pr[X = x]$.
- **Linearity of Expectation:** Most important property of expectation. $E[X + Y] = E[X] + E[Y]$ where X and Y are random variables.
- **Conditional Probability:** The probability of an event a occurring when you know that an event b has occurred is denoted by $Pr[a|b]$ or the probability of “a *given* b”. $Pr[a \wedge b] = Pr[a|b] \cdot Pr[b]$.
- **Independence:** Two events are independent if and only if $Pr[a \wedge b] = Pr[a] \cdot Pr[b]$.
- **Union Bound:** This is a useful bound that we will use fairly often. $Pr[a_1 \vee \dots \vee a_n] \leq Pr[a_1] + \dots + Pr[a_n]$.

5 Probably Approximately Correct Learning

Recall the function of a learning algorithm. Given a set of labelled training examples, its goal is to generate a prediction rule or “hypothesis”, h , that takes in test examples and outputs predictive labels. This is summarized in the diagram below:

What we now need to add to this model is the *distribution* from which the training examples and the test examples are generated. In particular, we assume that both the training examples and the test examples are generated from some unknown distribution \mathcal{D} , the *target distribution*. That is, both sets of examples come from the *same* distribution. Further,

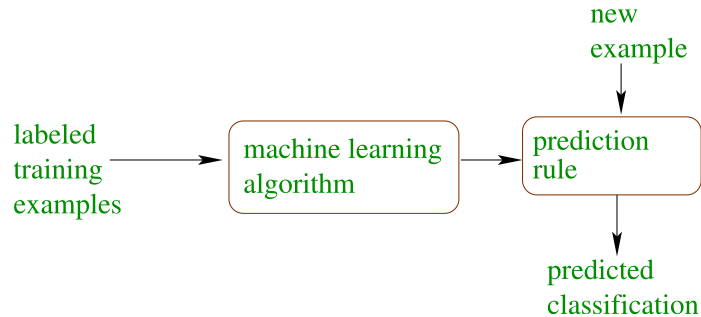


Figure 3: The learning algorithm takes in a set of labelled training examples and computes a prediction rule or “hypothesis”, h . h can then take in new test examples and output predictive labels.

we assume that each example is generated *independently*. So all examples are independent and identically distributed, or iid for short. In general we would like our results to be *distribution-free* or such that they hold for *any* target distribution.

But what about the labels? We want to focus on the simple case where there exists some function c , the *target concept*, and each example is labelled according to c . Further, we assume that $c \in \mathcal{C}$ where \mathcal{C} is our *target concept class* (or space).

Now that we have our definitions down, we can make claims about the accuracy of our algorithm. Our algorithm takes in training examples and outputs a hypothesis h . h makes a *mistake* if $h(x) \neq c(x)$. We can now measure the accuracy of our algorithm as simply:

$$err_{\mathcal{D}}(h) = Pr_{x \sim \mathcal{D}}[h(x) \neq c(x)] \quad (2)$$

We call this the algorithm’s *generalization error*. What we’re generally aiming for is to have $err_{\mathcal{D}}(h)$ be small. When this is true we say that the hypothesis h is *approximately correct*.

But we can’t always guarantee that $err_{\mathcal{D}}(h)$ is small because, depending on which training examples the algorithm gets, it could be the case that the training data is very unrepresentative of the collection of data as a whole. For example, if we’re trying to use a learning algorithm to generate a hypothesis that recognizes hand-written digits and we choose training examples from a bin of letters, it could be the case (although it’s extremely unlikely) that none of the letters we choose have the digit 2 on them. In this case, the hypothesis will almost surely have a large generalization error, or large $err_{\mathcal{D}}(h)$, since it will mislabel all the two’s it sees. But what’s the probability that we, by pure chance, get a set of training examples that don’t contain any two’s? It should be very low indeed, so low as to make it silly to even consider the possibility of it happening. And so we revise our definition of accuracy. Instead of seeking algorithms that generate hypotheses that are approximately correct, we instead seek algorithms that generate hypotheses that are approximately correct *with high probability*. That is, we look for hypotheses that are *probably approximately correct* (“PAC”). Here *approximately correct* refers to a small $err_{\mathcal{D}}(h)$ and *probably* refers to the probability that the $err_{\mathcal{D}}(h)$ is small after considering the distribution of training examples and any randomness inherent in the algorithm.

5.1 PAC Learnable Definition

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