Advanced Course in Machine Learning Spring 2010

The computational complexity of learning

Handouts are jointly prepared by Shie Mannor and Shai Shalev-Shwartz

In this lecture we discuss several issues regarding the computational complexity of learning algorithms.

1 What is the "input size" of a learning algorithm?

Usually, the computational complexity of an algorithm is analyzed as a function of its input size. When dealing with a learning algorithm, the input is a training set of examples. So, one can think that the computational complexity of a learning algorithm should be analyzed as a function of the training set size. However, this is clearly a wrong choice —having more examples should never increase the computational complexity since we can always sub-sample a smaller training set. In other words, having more examples is an asset rather than a burden.

Following the PAC learning framework, a more reasonable choice is to analyze the computational complexity of learning as a function of the sample complexity of learning. That is, we think on the sample complexity as the effective "input size" of the learning algorithm. Recall that the sample complexity usually depends on the required accuracy, ϵ, the required confidence, δ, and the complexity of the hypothesis class (e.g. the VC dimension of $\mathcal{H}$). Consequently, we shall say that a hypothesis class $\mathcal{H}$ is learnable in polynomial time if there exists a learning algorithm that (ϵ, δ)-learns $\mathcal{H}$ and its computational complexity is polynomial in its sample complexity.

The above assumes that the "size" of each example should be treated as a constant. It is sometimes convenient to explicitly analyze the dependence on the size or dimension of each example. For example, if each example is in $\{0,1\}^d$ we will define the input size as the sample complexity times d. Sometimes, each example is in $\mathbb{R}^d$. In that case, we shall assume that a representation of a real number using a constant number of bits suffices for our needs and therefore we will again refer to the input size as the sample complexity times $\Theta(d)$.

There is one more subtlety we swept under the rug. Based on the above definition, a learning algorithm can "cheat", by transferring the computational burden on the output hypothesis. For example, the algorithm can simply define the output hypothesis to be the function that stores the training set in its memory and whenever it gets a test example $\mathbf{x}$ it calculates the ERM hypothesis on the training set and applies it on $\mathbf{x}$. To prevent this "cheating", we shall require that the computational complexity of applying the output hypothesis should also be polynomial in the sample complexity.[1] The computational complexity of learning is therefore formally defined as follows:

Definition 1 (Computational complexity of learning) *Let $\mathcal{H}$ be a hypothesis class of functions defined over an instance space of size d, and let ϵ, δ be accuracy and confidence parameters. Let $m(\mathcal{H}, \epsilon, \delta)$ be the sample complexity of learning $\mathcal{H}$ with accuracy ϵ and confidence $1-\delta$. Then, the computational complexity of (ϵ,δ)-learning $\mathcal{H}$ is said to be bounded by $T(d\,m(\mathcal{H},\epsilon,\delta))$ if there exists a learning algorithm that (ϵ,δ)-learns $\mathcal{H}$ and whose runtime is bounded by $T(d\,m(\mathcal{H},\epsilon,\delta))$ and such that the runtime of the hypothesis it outputs on any new instance $\mathbf{x}$ is also bounded by $T(d\,m(\mathcal{H},\epsilon,\delta))$.*

In the above definition, we set all the relevant parameters, namely, $\mathcal{H}$,d,ϵ, and δ. Therefore, the effective "input size", $d\,m(|\mathcal{H}|,\epsilon,\delta)$, is also a predefined constant. Next, we would like to define learnability in polynomial time. To do so, we should speak about a family of learning problems of increasing effective input

[1]One should be more careful when referring to sample complexity in this context because we can mean the worst case sample complexity over all distributions or the sample complexity for the given distribution. Here, we left this definition in its vague mode.

size. Since the input size depends on $d, \mathcal{H}, \epsilon$, and δ, we should be precise about which of the parameters is regarded as a constant and which is regarded as a variable.

To illustrate this point, consider the problem of learning a finite hypothesis class. It is possible to solve the ERM problem in time $O(|\mathcal{H}|\, d\, m(\mathcal{H}, \epsilon, \delta))$ by performing an exhaustive search over $\mathcal{H}$ with a training set of size $m(\mathcal{H}, \epsilon, \delta)$. If we think on $|\mathcal{H}|$ as a constant while the rest of the parameters can change, then this algorithm runs in polynomial time. However, if we think on the rest of the parameters as constant and analyze the complexity as a function of $|\mathcal{H}|$ then this algorithm is exponential because it depends on $|\mathcal{H}|$ while $m(|\mathcal{H}|, \epsilon, \delta)$ grows only logarithmically with $|\mathcal{H}|$.

A possible definition is therefore[2] as follows:

Definition 2 (Efficient learning) *A sequence of learning problems $(d_n, \mathcal{H}_n, \epsilon_n, \delta_n)_{n=1}^{\infty}$ is efficiently learnable if there exists a polynomial p such that for each n, there exists a learning algorithm that (ϵ_n, δ_n)-learns $\mathcal{H}_n$ in time $p(d_n m(\mathcal{H}_n, \epsilon_n, \delta_n))$.*

Getting back to the problem of learning finite hypothesis classes, let us consider two sequence of learning problems. In the first, d_n, $\mathcal{H}_n$, and δ_n are constants, which we denote by $d, \mathcal{H}, \delta$, while $\epsilon_n = 1/n$. In this case, we can define the polynomial to be $p(x) = |\mathcal{H}|x$ and then it is easy to see that the runtime of the trivial ERM solver is $p(d_n m(\mathcal{H}_n, \epsilon_n, \delta_n))$. Therefore, this sequence of learning problems is efficiently learnable.

On the other hand, if we now set $\epsilon_n, \delta_n, d_n$ to be constants while letting $\mathcal{H}_n$ to be s.t. $|\mathcal{H}_n| = n$, then there is no constant polynomial such that the runtime of the trivial ERM solver will be $p(d_n m(\mathcal{H}_n, \epsilon_n, \delta_n))$. Hence, the trivial ERM solver does not solve the sequence of learning problems in polynomial time.

2 Improper learning

In some situations it is not possible to solve the ERM problem, or even to properly learn, in polynomial time. Nevertheless, the problem is efficiently learnable using a technique called "improper learning". The basic idea is to replace the original hypothesis class with a larger class so that the new class is easily learnable. The learning algorithm might return a hypothesis that does not belong to the original hypothesis class, hence the name "improper" learning.

To illustrate this idea consider the class of 3-term disjunctive normal form formulae (3-DNF); The instance space is $\{0,1\}^d$ and each hypothesis is represented by the boolean formula of the form $A_1(\mathbf{x}) \vee A_2(\mathbf{x}) \vee A_3(\mathbf{x})$, where each $A_i(\mathbf{x})$ is a conjunction of literals over the boolean variables $x_1, \ldots, x_d$. The number of such 3-DNF formulae is at most 3^{3d} hence $\log(|\mathcal{H}|) = \Theta(d)$ which gives that $m(\mathcal{H}, \epsilon, \delta) = \text{poly}(\frac{d}{\epsilon} \log \frac{1}{\delta})$.

Throughout this section we assume that the data is realizable, namely, one hypothesis achieves a zero generalization error (and therefore a zero training error with probability 1). It is possible to show that unless RP=NP, there is no polynomial time algorithm that *properly* learns a sequence of 3-term DNF learning problems in which $d_n = n$. By "properly" we mean that the algorithm should output a hypothesis which is a 3-term DNF formula. The proof uses a reduction of the graph 3-coloring problem to the problem of PAC learning 3-term DNF. The interested reader can find a detailed proof in Kearns&Vazirani, Section 1.4.

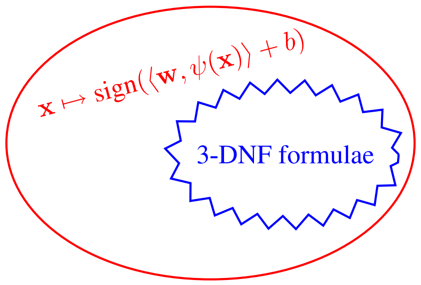
However, we now show that it is possible to efficiently learn 3-term DNF formulae. There is no contradiction to the above hardness result as we now allow improper learning. That is, we allow the learning algorithm to output a hypothesis which is not a 3-term DNF formula.

In particular, we will first make the observation that each 3-term DNF formula is equivalent to a 3-CNF formula (because $\vee$ distributes over $\wedge$). That is, $A_1 \vee A_2 \vee A_3 = \wedge_{u \in A_1, v \in A_2, w \in A_3} (u \vee v \vee w)$. Next, let us define: $\psi : \{0,1\}^d \to \{0,1\}^{2(2d)^3}$ s.t. for each triplet of literals u, v, w there are two variables indicating if $u \vee v \vee w$ is true or false. Observe that for each 3-DNF formula there exist a Halfspace, $\text{sign}(\langle \mathbf{w}, \psi(\mathbf{x}) \rangle + b)$, with the same truth table. Since we assume that the data is realizable, we can solve the ERM problem w.r.t. the class of Halfspaces using linear programming. Furthermore, the VC dimension of Halfspaces is the

[2]This definition is not standard as it does not require that $m(\mathcal{H}_n, \epsilon_n, \delta_n)$ will be polynomial in $1/\epsilon_n, 1/\delta_n$, and the VC of $\mathcal{H}_n$. We note, however, that in classification problems this is always the case

dimension, hence the sample complexity of learning the class of Halfspaces is order of $\frac{d^3}{\epsilon}\log\frac{1}{\delta}$. Thus, the overall runtime of this approach is $\mathrm{poly}\left(\frac{d^3}{\epsilon}\log\frac{1}{\delta}\right) = \mathrm{poly}(d\,m(\mathcal{H},\epsilon,\delta))$.

Intuitively, the idea is as follows. We started with a hypothesis class for which learning is hard. We switched to another representation and defined a larger hypothesis class which is larger than the original class but has more structure. In the new representation, solving the ERM problem is easy.


$\mathbf{x} \mapsto \mathrm{sign}(\langle \mathbf{w}, \psi(\mathbf{x})\rangle + b)$
3-DNF formulae

3 Hardness of learning

The previous section tells us that if we would like to establish hardness of learning it is not enough to show that solving the ERM problem (or even properly learning) is hard. So, how can we prove that a learning problem is hard? How can we show that there is no representation of the problem under which the problem becomes tractable ?

One approach is to rely on cryptographic assumptions. We now briefly describe the basic idea. Many cryptographic systems relies on the assumption that there exists a one way function. Roughly speaking, a one way function is a function $f : \{0,1\}^d \to \{0,1\}^d$ which is easy to compute but is believed to be hard to invert. More formally, f can be computed in time $\mathrm{poly}(d)$ but for every randomized polynomial time algorithm A, and for every polynomial $p(\cdot)$,

$$\mathbb{P}[f(A(f(x))) = f(x)] < \tfrac{1}{p(d)} ,$$

where the probability is taken over a random choice of x according to the uniform distribution over $\{0,1\}^d$ and the randomness of A.

A one-way function is called trapdoor one-way function if it is hard to invert it as long as we don't know a secret key of length $\mathrm{poly}(d)$. Such functions are parameterized by the secret key.

A hard bit function (a.k.a. hard-core predicate), $b : \{0,1\}^d \to \{0,1\}$, associated with a way function, f, is a boolean function that can be computed in polynomial time but such that for every polynomial algorithm A and a polynomial p we have

$$\mathbb{P}[A(f(x)) = b(x)] < \tfrac{1}{2} + \tfrac{1}{p(d)} ,$$

where the probability is taken over a random choice of x according to the uniform distribution over $\{0,1\}^d$ and the randomness of A. This implies that when x is uniform over $\{0,1\}^d$ then $b(x)$ is almost an unbiased coin.

Now, let f be a trapdoor one way function, let b be a hard bit, and consider the following distribution over examples. First, z is chosen from $\{0,1\}^d$ according to the uniform distribution, then the instance, x, is set to be $x = f(z)$, and the label is set to be $y = b(z)$.

Consider the hypothesis class to be the set of trapdoor one way functions parameterized by the secret key. Since the length of the secret key is $\mathrm{poly}(d)$, it means that the sample complexity of PAC learning this class is order of $\mathrm{poly}(d/\epsilon, \log(1/\delta))$. But, learning (even improperly) this class in polynomial time implies the existence of an algorithm for which $\mathbb{P}[A(f(z)) = b(z)] > 1 - \epsilon$, which contradicts the fact that b is a hard bit for f.

A more detailed treatment, as well as a concrete example, can be found in Kearns&Vazirani, Chapter 6. Using reductions, they also show that the class of functions that can be calculated by small Boolean circuits is not efficiently learnable, even in the realizable case.

4 Learning DNF Formulas

Previously, we saw that the class of 3-DNF formulas is learnable in polynomial time. Using the same technique, it is easy to show that learning the class of k-DNF formulas in time $\mathrm{poly}(d^k \frac{1}{\epsilon} \log \frac{1}{\delta})$ is possible.

In this lecture we consider the problem of learning DNF formulas in which k can be as large as $\mathrm{poly}(d)$. We will refer to this class simply as the class of DNF formulas. This class received a lot of attention because:

- Natural form of knowledge representation for people
- Historical reasons: considered by Valiant, who called the problem "tantalizing" and "apparently [simple]" yet has proved a great challenge over the last 25 years
- Useful for machine learning - e.g., learning decision trees or decision lists.

The time complexity of improperly learning DNFs is unknown. The fastest known algorithm, due to Klivans and Servedio, runs in time $\exp(d^{1/3} \log^2 d)$. The idea is to show that for any DNF formula, there is a polynomial in $x_1, \ldots, x_d$ of degree at most $d^{1/3} \log d$ which is positive whenever the DNF is true and negative whenever the DNF is false. Linear programming can be used to find a hypothesis consistent with every example in time $\exp(d^{1/3} \log^2 d)$.

Because the problem seems to be so hard, many researchers attempt to solve it under distributional assumptions, in particular, assuming that the underlying distribution over instances is uniform over $\{0, 1\}^d$.

In 1990, Verbeugt [Ver90] observed that, under the uniform distribution, any term in the target DNF which is longer than $\log(d/\epsilon)$ is essentially always false, and thus irrelevant. This fairly easily leads to an algorithm for learning DNF under uniform distribution in quasipolynomial time: roughly $d^{\log d}$.

The question whether DNF can be learned in polynomial time under the uniform distribution is still open.

In 1992, Kushilevitz and Mansour presented a very elegant algorithm for learning decision trees (a subset of DNFs) under the uniform distribution and while allowing the algorithm to ask membership queries (that is, for any $x \in \{0, 1\}^d$ the algorithm can ask what is the label of x).

In 1994, Jackson extended Kushilevitz and Mansour results and presented the Harmonic Sieve—an algorithm that can learn DNFs in polynomial time under the uniform distribution and with membership queries.