Machine learning theory

Computational complexity of learning algorithms

Hamid Beigy

Sharif university of technology

April 27, 2020

Table of contents

Introduction

Introduction

1. We have studied the statistical perspective of learning, namely, how many samples are needed for learning.

Sample complexity of learning

How many examples do we need in order to learn from a specific concept class?

2. This focused on the amount of information learning requires.
3. We can't ignore the computational price.

Computational complexity of learning

How much computational effort is needed for PAC learning?

4. Once a sufficient training sample is available to the learner, there is some computation to be done to find a hypothesis.
5. The computational complexity of learning should be viewed in the wider context of the computational complexity of general algorithmic tasks.

Computational complexity

Introduction

1. How can we say that one algorithm performs better than another?
2. Quantify the resources required to execute an algorithm.
   - Time
   - Memory
   - I/O
   - circuits, power, etc
3. Time is not merely CPU clock cycles, we want to study algorithms independent or implementations, platforms, and hardware.
4. We need an objective point of reference.
5. We measure time by the number of operations as a function of an algorithm's input size.
6. Hence, we need
   - a computational model
   - definition of the operations
   - definition of input size
   - definition of cost (uniform vs logarithmic)
   - studying time independent of platforms and hardware
7. The input size is defined as the number of bits required to represent the input. For example

   Sorting : The number of items to be sorted.

   Graphs : The number of vertices and/or edges.

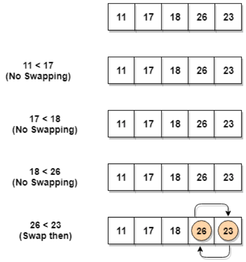
   Numerical : The number of bits needed to represent a number.

Random access machine model

1. Running time of algorithms can be measured in a machine-independent way using the a computational model such as random access machine (RAM) or Turing machine.
2. This model assumes a single processor.
3. Instructions are executed one after the other, with no concurrent operations.
4. This model of computation is an abstraction that allows us to compare algorithms on the basis of performance.
5. The assumptions made in the RAM model to accomplish this are:
    - Each simple operation takes 1 time step.
    - Loops and subroutines are not simple operations.
    - Each memory access takes one time step, and there is no shortage of memory.
6. For any given problem the running time of an algorithms is assumed to be the number of time steps.
7. The space used by an algorithm is assumed to be the number of RAM memory cells.

Types of complexity

1. Four types of complexity could be considered when analyzing algorithm performance.
   - worst-case complexity,
   - best-case complexity,
   - average-case complexity, and
   - amortized complexity.
2. In the worst case analysis, we calculate upper bound on running time of an algorithm.
3. Considering bubble sort



4. In bubble Sort, $(n-1)$ comparisons will be done in the 1st pass, $(n-2)$ in 2nd pass,$(n-3)$ in 3rd pass and so on. So the total number of comparisons $(c(n))$ will be,

$$\begin{aligned} T(n) &= (n-1)+(n-2)+\ldots+3+2+1 \\ &= \frac{n(n-1)}{2}. \end{aligned}$$

O notation

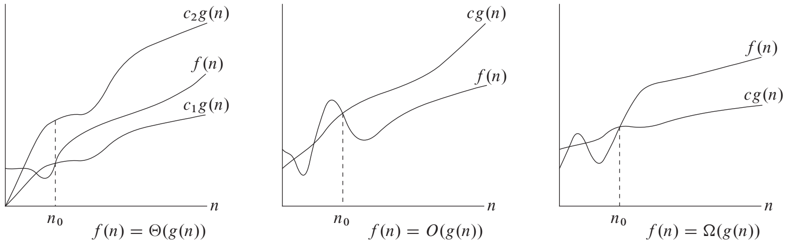
1. The runtime of an algorithm depends on the machine running the algorithm.
2. To avoid such dependency, the runtime is computed in an asymptotic sense.
3. We are typically only interested in how fast $T(n)$ is growing as a function of input size n

Definition (big-O notation)

Let f and g be functions $f, g : \mathbb{N} \mapsto \mathbb{R}^+$. We say that $f(n) = O(g(n))$ if there exists positive integers c and n_0 such that for every integer $n \geq n_0$,

$$f(n) \leq cg(n).$$

When $f(n) = O(g(n))$, we say that $g(n)$ is an upper bound for $f(n)$.



4. For bubble sort, it is easy to show that $T(n) = O(n^2)$.

Computational complexity of learning

Introduction

1. Recall from previous sessions

Learning algorithm

A learning algorithm has access to a domain of examples, $\mathcal{Z}$, a hypothesis class, H, a loss function, ℓ, and a training set, S, of examples from $\mathcal{Z}$ that are sampled i.i.d. according to an unknown distribution $\mathcal{D}$. Given parameters ϵ and δ, the algorithm should output a hypothesis h such that with probability of at least $1-\delta$,

$$\mathbf{R}(h) \leq \min_{h' \in H} \mathbf{R}(h') + \epsilon$$

2. The actual runtime of an algorithm in seconds depends on the specific machine.
3. To allow machine independent analysis, we use the standard approach in computational complexity theory.
    - First, we rely on a notion of an abstract machine, such as a Turing machine or RAM.
    - Second, we analyze the runtime in an asymptotic sense, while ignoring constant factors.
4. Usually, the asymptote is with respect to the size of the input to the algorithm. For example, the number of elements of array given to the Bubble-sort algorithm.
5. In the context of learning algorithms, there is no clear notion of input size.

What is the input size?

1. In the context of learning algorithms, there is no clear notion of input size.
2. We can define the input size as the size of the training set, m, the algorithm receives.

Problem

If we give the algorithm a very large number of examples, much larger than the sample complexity of the learning problem, the algorithm can simply ignore the extra examples.

3. Therefore, a larger m does not make the problem more difficult, and, the runtime of learning algorithm should not increase as we increase m.
4. We can still analyze the runtime as a ϵ, δ, n, or some measures of the complexity of H.

Example (Learning axis aligned rectangles)

- This problem is derived by specifying ϵ, δ, and n.
- We can define a sequence of rectangles learning problems by fixing ϵ, δ, and varying $n = 2, 3, \ldots$.
- We can also define another sequence of rectangles learning problems by fixing d, δ and varying $\epsilon = \frac{1}{2}, \frac{1}{3}, \ldots$.
- One can of course choose other sequences of such problems.
- When a sequence of the problems is fixed, one can analyze the asymptotic runtime.

Learning algorithm

1. A learning algorithm receives a training set and outputs a hypothesis, which is a program.
2. A learning algorithm can cheat, by transferring the computational burden to the output hypothesis.
3. Considering the following learning algorithm.
   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 x it calculates the ERM hypothesis on the training set and applies it on x.
4. This algorithm has a fixed output and can run in constant time.
5. The hardness is now in implementing the output classifier to obtain a label prediction.
6. To prevent this cheating,
   We shall require that the output of a learning algorithm must be applied to predict the label of a new example in time that does not exceed the runtime of training.

Computational complexity of a learning algorithm

Definition (Computational complexity of a learning algorithm)

We define the complexity of learning in two steps. First we consider the computational complexity of a fixed learning problem $(\mathcal{Z}, H, \ell)$. Then, in the second step we consider the rate of change of that complexity along a sequence of such tasks.

1. Given function $f : (0,1)^2 \to \mathbb{N}$, a problem $(\mathcal{Z}, H, \ell)$, and an algorithm, A. Algorithm A solves the problem in time $O(f)$, if there exists some constant c, such that for every distribution $\mathcal{D}$ over $\mathcal{Z}$, and input $\epsilon, \delta \in (0,1)$, when A has access to samples $S \sim \mathcal{D}$,
   - Algorithm$A$ terminates after performing at most $cf(\epsilon, \delta)$ operations.
   - The output of A, denoted h_A, can be applied to predict the label of a new example while performing at most $cf(\epsilon, \delta)$ operations.
   - The output of A is probably approximately correct; i.e.with probability of at least $1 - \delta$ $\mathbf{R}(h) \leq \min_{h' \in H} \mathbf{R}(h') + \epsilon$.
2. Consider a sequence of problems, $(\mathcal{Z}_n, H_n, \ell_n)_{n=1}^{\infty}$, where problem n is defined by $(\mathcal{Z}, H, \ell)$. Let A be an algorithm designed for solving these problems. Given a function $g : (0,1)^2 \to \mathbb{N}$, we say that the runtime of A with respect to $(\mathcal{Z}_n, H_n, \ell_n)$ is $O(g)$, if for all n, A solves the problem $(\mathcal{Z}_n, H_n, \ell_n)$ in time $O(f_n)$, where $f_n : (0,1)^2 \to \mathbb{N}$ is defined by $f_n(\epsilon, \delta) = f_n(n, \epsilon, \delta)$.

Computational complexity of a learning algorithm

1. Algorithm A is efficient with respect to a sequence $(\mathcal{Z}_n, H_n, \ell_n)$ if its runtime is $O(p(n, \frac{1}{\epsilon}, \frac{1}{\delta}))$, for some polynomial p.
2. This definition implies that the question whether a general learning problem can be solved efficiently depends on **how it can be broken into a sequence of specific learning problems**.

Example (Learning a finite hypothesis class)

- It was shown that the ERM rule over H is guaranteed to (ϵ, δ)-learn H if the number of training examples is order of $m_H(\epsilon, \delta) = \dfrac{\log(|H|/\delta)}{\epsilon^2}$.
- Let the evaluation of a hypothesis on an example takes a constant time, it is possible to implement the ERM rule in time $O(|H| m_H(\epsilon, \delta))$ by performing an exhaustive search over H with a training set of size $m_H(\epsilon, \delta)$.
- For any fixed finite H, the exhaustive search algorithm runs in polynomial time.
- If we define a sequence of problems in which $|H_n| = n$, then the exhaustive search is still considered to be efficient.
- However, if we define a sequence of problems for which $|H_n| = 2^n$, then **the sample complexity is still polynomial in n** but the**computational complexity of the exhaustive search algorithm grows exponentially with n** (thus, rendered inefficient).

Implementing ERM rule

1. For problem, $(H, \mathcal{Z}_n, \ell)$, the corresponding ERM rule can be defined as follows:

Definition (ERM rule)

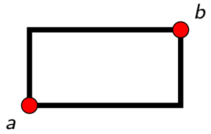
For a finite sample $S \in \mathcal{Z}^m$ output $h \in H$ that minimizes $\hat{\mathbf{R}}(h) = \frac{1}{|S|} \sum_{z \in S} \ell(h, z)$.

Example (Finite hypothesis classes)

- The sample complexity of learning a finite class is upper bounded by $m_H(\epsilon, \delta) = c \log\left(c|H|/\delta\right)/\epsilon^c$, where $c = 1$ in realizable case and $c = 2$ in nonrealizable case.
- A simple implementation of ERM rule for finite hypothesis class is exhaustive search.
- Assuming that the evaluation of $\ell(h, z)$ on a single example takes a constant amount of time, k, the runtime of this exhaustive search becomes $k|H|m$,where m is the size of the training set.
- Then then the runtime becomes $k|H|c \log\left(c|H|/\delta\right)/\epsilon^c$.
- The linear dependency on H makes this approach inefficient for large classes.
- Formally, if we define a sequence of problems $(\mathcal{Z}_n, H_n, \ell_n)_{n=1}^{\infty}$ such that $\log(|H_n|) = n$, then the **exhaustive search approach yields an exponential runtime**.
- Inefficiency of one implementation doesn't imply that no efficient ERM implementation exists.

Axis aligned rectangles (Realizable case)

1. Let $H_n = \left\{ h_{(a_1,\ldots,a_n,b_1,\ldots,b_n)} \;\middle|\; \forall i, a_i \leq b_i \right\} \in \mathbb{R}^n$, where $h_{(a_1,\ldots,a_n,b_1,\ldots,b_n)}(x) = 1$ when $\forall i, x_i \in [a_i, b_i]$.
2. This problem is efficiently learnable in the realizable case.
   - Consider implementing the ERM rule in the realizable case.
   - We need only to specify n corners of this rectangle.



   - For each $i \in \{1, 2, \ldots, n\}$, set

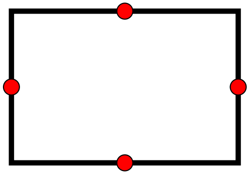
$$a_i = \min\{x_i \mid (x, 1) \in S\}$$
$$b_i = \max\{x_i \mid (x, 1) \in S\}$$

   - The resulting rectangle has zero training error and the total runtime is $O(nm)$, where $m \geq m_{H_n}(\epsilon, \delta) = \frac{2n + \ln(2/\delta)}{\epsilon}$.

Axis aligned rectangles (Unrealizable case) i

This problem is not efficiently learnable in the agnostic case.

1. We can specify each rectangle with at most $2n$ points.



2. There are $\binom{m}{2n}$ different subsets with size $2n$ points, which contain non-repetitive elements of the training set.
3. We have also $\binom{m}{2n} = O(m^{2n})$ and $\binom{m}{2n} \leq m^{2n}$.
4. If you allow the repetitive elements in each subset, then we have exactly m^{2n} subsets of size $2n$. In learning, this case is allowed.
5. Let these subsets be $S_1, S_2, \ldots, S_{m^{2n}}$ and we use the following algorithm.
   a. Build a rectangle for each S_i using $2n$ points in this set and then calculate the empirical risk of this rectangle using the training set S. Let this rectangle be denoted by h_i, which contains the set S_i.
   b. Return the rectangle with the minimum empirical risk, i.e. return $\min_{1 \leq i \leq m^{2n}} h_i$.

Axis aligned rectangles (Unrealizable case) ii

6. We must prove the correctness of the given algorithm.

Lemma (Correctness of algorithm for finding the smallest empirical risk hypothesis)

For any $h \in H_n$ and for every S, there exist a h_i such that $\hat{\mathbf{R}}(h_i) \leq \hat{\mathbf{R}}(h)$.

7. The running time for this algorithm is $(m_{H_d}(\epsilon, \delta))^{2n+1}$.
8. If n is fixed, then running time is polynomial and there exist efficient learning algorithms for this class.
9. if If n is not fixed, then running time is exponential and there is no efficient learning algorithms for this class.
10. Solving this problem by using ERM in the agnostic setting is **NP-hard** unless $\mathbf{P} = \mathbf{NP}$.
11. There are successful agnostic PAC learners that run in time polynomial in $\frac{1}{\epsilon}$ and $\frac{1}{\delta}$ but their dependence on the dimension n is not polynomial.
12. This does not contradict the hardness result given before.

Boolean conjunctions

1. A Boolean conjunction is in the form of $x_{i_1} \wedge \ldots \wedge x_{i_k} \wedge \neg x_{j_1} \wedge \ldots \wedge \neg x_{j_r}$ for some indices $i_1, \ldots, i_k, j_1, \ldots, j_r \in \{1, \ldots, n\}$.
2. This proposition defines function $h(x) = 1$ if $i_1 = \ldots = i_k = 1$ and $j_1 = \ldots = j_r = 0$.
3. What is VC dimension of this class? We can calculate the upper bound of the VC as $VC(H) \leq \log|H|$.
4. Let H_{C_n} be the class of all Boolean conjunctions over $\{0,1\}^n$, where $|H_{C_n}| = \Theta(3^n)$ and hence $VC(H_{C_n}) \leq \log|H_{C_n}| = n \log 3$.
5. Hence, the sample complexity of learning H_{C_n} using the ERM rule is at most $\dfrac{n \log 3 + \log(1/\delta)}{\epsilon}$.
6. This problem is efficiently learnable in the realizable case.
   - Let $h_0(x) = (x_1 \wedge \neg x_1) \wedge (x_2 \wedge \neg x_2) \wedge \ldots \wedge (x_n \wedge \neg x_n)$.
   - Note that $\forall x$, we have $h_0(x) = 0$.
   - Then we build a sequence of hypothesis $h_1, h_2, \ldots$ by testing only positive samples and removing inconsistent literals.
   - The resulting conjunction has zero training error and the total runtime is $O(nm)$.
7. This problem is not efficiently learnable in the agnostic case.
   - There is no algorithm whose running time is polynomial in m and n that guaranteed to find an ERM hypothesis for the class of Boolean conjunctions in the unrealizable case unless $\mathrm{P} = \mathrm{NP}$.

Learning 3-Term DNF

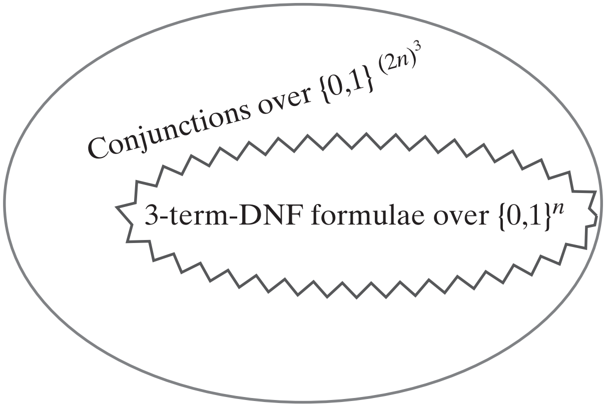
1. Each hypothesis is represented by a Boolean formula of the form $h(x) = A_1(x) \vee A_2(x) \vee A_3(x)$, where each $A_i(x)$ is a Boolean conjunction.
2. $h(x) = 1$ if either $A_1(x)$ or $A_2(x)$ or $A_3(x)$ output the label 1.
3. Let $H^n_{3DNF_n}$ be the hypothesis class of all such 3-term DNF formula. We have $|H_{3DNF_n}| = 3^{3n}$ and $VC(H_{3DNF_n}) \leq \log|H_{3DNF_n}| = 3n$.
4. The sample complexity of learning H_{3DNF_n} is at most $\dfrac{3n + \log(1/\delta)}{\epsilon^2}$.
5. How hard it is to compute ERM over H_{3DNF_n} using sample of size alert $\dfrac{3n + \log(1/\delta)}{\epsilon^2}$?
6. There is no polynomial time algorithm that properly learns a sequence of 3DNF learning problems unless RP = NP even in realizable case.
7. By properly, we mean that the algorithm should output a hypothesis that is a 3DNF formula.

Learning 3-Term DNF

1. We will show that it is possible to learn 3DNF efficiently, but using ERM with respect to a larger class by allowing representation independent learning.
2. In this case, we allow the learning algorithm to output a hypothesis that is not a 3DNF formula.
3. The basic idea is to replace the original hypothesis class of 3DNF formula with a larger hypothesis class so that the new class is easily learnable.
4. The learning algorithm might return a hypothesis that does not belong to the original hypothesis class; hence the name representation independent learning.
5. In most situations, we are interested in returning a hypothesis with good predictive ability.
6. By distributing $\vee$ over $\wedge$, each 3DNF formula can be written as
$$A_1 \vee A_2 \vee A_3 = \bigwedge_{u \in A_1, v \in A_2, w \in A_3} (u \vee v \vee w).$$
7. Let us define $\psi : \{0,1\}^n \mapsto \{0,1\}^{(2n)^3}$ such that for each triplet of literals u, v, w there is a variable in the range of ψ indicating if $(u \vee v \vee w)$ is true or false.
8. For each 3DNF over $\{0,1\}^n$ there is a conjunction over $\{0,1\}^{(2n)^3}$, with the same truth table.
9. We can solve the ERM problem with respect to class of conjunctions over $\{0,1\}^{(2n)^3}$ with sample complexity $\dfrac{n^3 + \log(1/\delta)}{\epsilon^2}$ and runtime is polynomial in n.

Learning 3-Term DNF

1. Intuitively, the idea is as follows.
   - We started with a hypothesis class for which learning is hard.
   - We switched to another representation where the hypothesis class is larger than the original class but has more structure, which allows for a more efficient ERM search.
   - In the new representation, solving the ERM problem is easy.
   - Then, we may transform back the learned hypothesis to the original hypothesis class


Conjunctions over $\{0,1\}^{(2n)^3}$
3-term-DNF formulae over $\{0,1\}^n$

Hardness of learning

Hardness of learning

1. We have shown that the computational hardness of implementing ERM doesn't imply that such a class H is not learnable.
2. How can we prove that a learning problem is computationally hard?
3. One approach is to rely on cryptographic assumptions.
4. In some sense, cryptography is the opposite of learning.
5. In learning we try to uncover some rule underlying the examples we see.
6. In cryptography, the goal is to make sure that nobody will be able to discover some secret.
7. On that high level intuitive sense, results about the cryptographic security of some system translate into results about the unlearnability of some corresponding task.
8. The common approach for proving that cryptographic protocols are secure is to start with some cryptographic assumptions.

Hardness of learning

1. The basic idea of how to deduce hardness of learnability from cryptographic assumptions.
2. Many cryptographic systems rely on the assumption that there exists a one way function $f : \{0,1\}^n \mapsto \{0,1\}^n$ that is easy to compute but is hard to invert.
3. Formally, f can be computed in time $poly(n)$ but for any randomized polynomial time algorithm A, and for every polynomial $p(.)$,

$$\mathbb{P}\left[f(A(f(x))) = f(x)\right] < \frac{1}{p(n)}$$

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

4. To solve this problem, in cryptography trapdoor one way function are used.

Definition (Trapdoor one way function)

A one way function, f, is called trapdoor one way function if, for some polynomial function p, for every n there exists a bit-string s_n (called a secret key) of length $\leq p(n)$, such that there is a polynomial time algorithm that, for every n and every $x \in \{0,1\}^n$, on input $(f(x), s_n)$ outputs x.

5. Although f is hard to invert, once one has access to its secret key, inverting f becomes feasible.

Hardness of learning

1. let F_n be a family of trapdoor functions over $\{0, 1\}^n$ that can be calculated by some polynomial time algorithm.
2. That is, we fix an algorithm that given a secret key (representing one function in F_n) and an input vector, it calculates the value of the function corresponding to the secret key on the input vector in polynomial time.
3. Consider the task of learning the class of the corresponding inverses, $H_F^n = \left\{f^{-1} \;\middle|\; f \in F_n\right\}$.
4. Since each function in this class can be inverted by some secret key s_n of size polynomial in n, the class H_F^n can be parameterized by these keys and its size is at most $2^{p(n)}$.
5. Its sample complexity is therefore polynomial in n.
6. We claim that there can be no efficient learner for this class.
   - Assume that there is a learner L.
   - Learner L first samples uniformly at random a polynomial number of strings in $\{0, 1\}^n$.
   - Then computes f over them, we could generate a labeled training sample of pairs $(f(x), x)$.
   - This should suffice for our learner to figure out an (ϵ, δ) approximation of f^{-1}.
   - This violates the one way property of f.
7. What is $VC(F_n)$?

Summary

Summary

1. We derived efficient algorithms for solving the ERM problem for some classes under the realizability assumption.
2. However, implementing ERM for some of these classes in the agnostic case is NP-hard.
3. From the statistical perspective, there is no difference between the realizable and agnostic cases, both are learnable because they have finite VC dimension.
4. We have also shown that implementing ERM for 3DNF is hard even in the realizable case, yet the class is efficiently learnable by another algorithm.
5. Hardness of implementing the ERM rule for several natural hypothesis classes has motivated the development of alternative learning methods, which we will discuss in the next sessions.

Readings

1. Chapter 8 of Shai Shalev-Shwartz and Shai Ben-David Book[1]
2. Chapter 6 of Kearns and Vazirani Book[2].

[1] Shai Shalev-Shwartz and Shai Ben-David. *Understanding machine learning : From theory to algorithms*. Cambridge University Press, 2014.

[2] Michael J. Kearns and Umesh Vazirani. *An Introduction to Computational Learning Theory*. MIT Press, 1994.

References

- Michael J. Kearns and Umesh Vazirani. *An Introduction to Computational Learning Theory*. MIT Press, 1994.
- Shai Shalev-Shwartz and Shai Ben-David. *Understanding machine learning : From theory to algorithms*. Cambridge University Press, 2014.

Questions?