VI Efficient Learning.

1. Introduction
2. Review of PAC Learning and Potential Learnability
3. Introduction to Complexity Theory
4. The Efficiency of PAC Learning
5. Summary of Lecture

1 Introduction

We begin with a review of PAC learning and potential learnability from the last two lectures.

At the end of the last lecture we saw that we need to study efficiency.

We introduce complexity theory, which studies the relationship between how much time an algorithm needs to run and the size of its input.

This gives us a way of saying:

- whether an algorithm is efficient or not.
- whether a problem is intrinsically hard or not.

Finally, we relate m_0, our estimate of the number of examples needed for PAC learning, to the efficiency of a learning algorithm.

2 Review of PAC Learning and Potential Learnability

We're interested in learning problems where we have:

- a teacher with a target concept $t \in C$ in mind.
- a learner searching for hypotheses which approximate the target concept.
- a training sequence of labelled examples of the target concept.

The training sequence contains a subset of the example space X.

The learner is tested by asking it to classify examples which it has not seen.

Probably Approximately Correct (PAC) Learning

The distributions of examples during training and testing can affect performance heavily.

PAC learning assumes the distribution is the same during training and testing.

- This minimises the effect of the distribution on errors made during testing.
- It also lets us define the error of a hypothesis in terms of a distribution.

$$\mathrm{error}_\mu(h) \text{ and not just } \mathrm{error}(h)$$

Notes: For simplicity, we'll normally just write "error". Anthony and Biggs write this $\mathrm{er}_\mu(h)$ or $\mathrm{er}(h)$.

What kind of Guarantees can we Make?

What kind of guarantee about learning can we make in this model?

Can we guarantee the algorithm will find the target concept?

In an infinite space we can't see all examples, so we can only *approximate* the target concept.

Examples are selected according to a distribution. By chance, they will sometimes not reflect the distribution so it is only *probable* that we can approximate the target concept well.

(**Note:** In a finite example space we can potentially see all examples, in which case we can probably *exactly* learn the target concept.)

Definition of PAC Learning

The more data an algorithm is trained on, the more likely it is to find a good approximation of the target concept.

An algorithm is PAC for a hypothesis space H if it is possible to see enough data to make:

$$\Pr(\mathrm{error} < \epsilon) \quad > \quad 1 - \delta$$

where $0 \leq \epsilon \leq 1$ and $0 \leq \delta \leq 1$.

Potential Learnability

A hypothesis space H is potentially learnable if it is possible to see enough data to make:

$$\Pr(\text{error of any consistent hypothesis} \geq \epsilon) \quad > \quad 1 - \delta$$

If H is potentially learnable, then any consistent algorithm is PAC for H.

Efficiency and PAC Learning

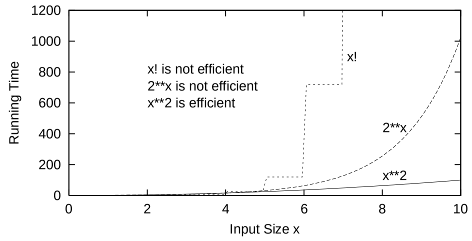
So far we have asked if it is possible to see enough data to probably approximately learn a target concept.

We also saw that the enumeration algorithm does this for any finite hypothesis space, but that it can take a very long time.

Now we ask whether it is possible to probably approximately learn a target concept *in a reasonable amount of time.*

3 Introduction to Complexity Theory

How does the *running time* of an algorithm depend on input size?



Input size can be measured in different ways, e.g., string length.

Measuring Running Time

Running time depends on how fast the computer is.

- So we count the number of operations needed instead of the time needed.

Implementations of algorithms differ.

- So we don't count operations exactly, but only look at the form (polynomial?, exponential?, factorial?) of relationship between input and running time.

Polynomial Functions

A *polynomial function* involves sums of powers of a variable. E.g.,

$$f(x) = 8x^5 - 7x^4 - 6x^3 + 4x^2 - 3x + 2$$

is a polynomial in the variable x. A simple polynomial is:

$$f(x) = x^2$$

Exponential Functions

An *exponential* function raises a variable to a power. E.g.,

$$f(x) = 2^x$$

is exponential in the variable x.

O-notation

Let A be an algorithm which accepts inputs of varying size s.

The running time of A is $O(f(s))$ if, for any s, the number of operations needed is at most $Kf(s)$, where K is some constant.

What is the running time of this algorithm?

```
let the string s1 = {1,2,3} be the input to the algorithm

for each element e1 in s1
    print e1
```

The running time is 3 (or $O(3)$) since there is one operation for each of 3 inputs (each input is printed once).

If we allow $s1$ to have any length s, the running time is $O(s)$.

What is the running time of this algorithm?

```
s1 = {1,2,3}

for each element e1 in s1
    for each element e2 in s1
        print e1 * e2
```

What about this one?

```
s1 = {1,2,3}

for each element e1 in s1
    for each element e2 in s1
        print e1 * e2
        print e1/e2
```


Worst-case and Average-case Time

The *worst-case* time is the longest time for any input.

The *average-case* time is the average over all inputs.

What are the average and worst-case times for the following?

```
let s1 be a random permutation of the string "abcdefghi"
position = 1
for each letter l in s1
    if l = "c" then
        output position
        quit
    else position = position + 1
```

What are the worst and average case times if $s1$ consists of n unique characters?

Why use Worst-case Time?

We normally use worst-case time. Why?

Because we want to provide a guarantee about performance. E.g.,

- It is guaranteed that an n-bit input to the algorithm will not take more than 2^n operations.

Polynomial time = easy problem

An algorithm with running time $O(s^r)$ for some fixed integer $r \geq 1$ is a *polynomial time* algorithm.

Problems which can be solved by a polynomial time algorithm are easy.

- To show a problem is easy, give a polynomial time algorithm for it.
- How can we show a problem is hard? I.e., how can we show there is no polynomial time algorithm for it?

Hard Problems

To show a problem is hard, equate it with a problem which is *NP-Complete.*

NP-Complete problems are believed to be hard:

- The unproved but widely believed $P \neq NP$ *conjecture* holds that there is no polynomial time algorithm for any NP-Complete problem.

NP-Complete problems are said to be *NP-hard.*

Satisfiability: A Hard Problem

Satisfiability is a well-known NP-Complete problem:

$\boxed{\text{Given a Boolean formula } \phi \text{ in } n \text{ variables, is there a positive example of } \langle\phi\rangle\text{?}}$

E.g., suppose we have 3 variables: $u_1u_2u_3$, and the following Boolean formula:

$$(u_1 \vee \bar{u}_2) \wedge (u_1 \vee u_3) \wedge (u_2 \vee \bar{u}_3) \wedge (\bar{u}_1 \vee u_3) \quad (1)$$

Is there a way of assigning truth values to $u_1u_2u_3$ so that (1) is true? I.e., are there any positive examples of this concept?

Satisfiability: A Hard Problem

u_1	u_2	u_3	$(u_1 \vee \bar{u}_2)$	$(u_1 \vee u_3)$	$(u_2 \vee \bar{u}_3)$	$(\bar{u}_1 \vee u_3)$
0	0	0				
0	0	1				
0	1	0				
0	1	1				
1	0	0				
1	0	1				
1	1	0				
1	1	1				

How to Show a Problem is Hard

Suppose:

- We want to show that Π is a hard problem.
- We know another problem Π_0 is NP-Complete.

Suppose we can show that if there is a polynomial time algorithm for Π there must also be one for Π_0.

If so, Π is NP-hard: if it turns out $P \neq NP$, then there is no polynomial time algorithm for Π.

We'll see an example in the next lecture.

4 The Efficiency of PAC Learning

When will an algorithm PAC learn *efficiently*?

The *sample complexity* of L on a concept space C is the number of labelled examples needed for L to PAC learn a target concept in C.

L learns efficiently only if the sample complexity is a polynomial function of the input size.

Sample Complexity

In lecture 5 we saw a formula for the number of examples needed for a consistent algorithm in a finite space to reach confidence δ and accuracy ϵ:

$$m_0 = \left\lceil \frac{1}{\epsilon} \ln \frac{|H|}{\delta} \right\rceil \quad (2)$$

m_0 is an upper bound on the sample complexity.

So, an algorithm can PAC learn efficiently if m_0 is polynomial in the input.

Note: If m_0 is not polynomial in the input, it may still be possible to learn H efficiently. m_0 is an upper bound on sample complexity.

Input Size and $|H|$

The input size (number of variables) defines the size of the hypothesis space $|H|$.

- E.g. $|B_n| = 2^{2^n}$, and $|B_1| < |B_2| < |B_3| \ldots$

From (2), for m_0 to be polynomial in the input, $\ln |H|$ must be polynomial in the input.

Examples

Example: Boolean functions

If $H = B_n$ then $\ln |H| = \ln |B_n| = \ln 2^{2^n}$, which is not polynomial in n. Consequently m_0 does not show that B_n can be PAC learnt efficiently. (In fact, B_n cannot be learned efficiently, but we have not shown this.)

Example: Monomials

If $H = M_n$ then $\ln |H| = \ln |M_n| = \ln 3^n$, which is polynomial in n. Consequently M_n can be PAC learnt efficiently.

Note: Even if a class C cannot be learned efficiently, because we use worst-case complexity, many instances within C *can* be learned efficiently.

5 Summary of Lecture

We reviewed the ideas of PAC learning and potential learnability:

- An algorithm L is *PAC* for a space H if it is possible to see enough data to make:

$$\Pr(\text{error} < \epsilon) \quad > \quad 1 - \delta$$

- A hypothesis space H is *potentially learnable* if it is possible to see enough data to make:

$$\Pr(\text{error of any consistent hypothesis} \geq \epsilon) \quad > \quad 1 - \delta$$

Summary of Complexity Theory

Of course we want the amount of data/time needed to be reasonable.

An algorithm is *efficient* if its *running time* (number of operations) is a *polynomial* function of the input size.

We measure *worst-case time* because we want to guarantee performance will not be worse than some value, no matter what the input is.

In *O-notation*, the running time of A is $O(f(s))$ if, for any input size s, the number of operations needed is at most $Kf(s)$, where K is some constant.

To show an algorithm Π is *hard*, we show that if there is a polynomial time algorithm for Π there must also be one for an NP-Complete algorithm Π_0. (We'll see an example in the next lecture.)

Summary of PAC Learning and Complexity

The sample complexity of L is the number of samples needed for L to PAC learn H.

m_0 gives us an upper bound on sample complexity.

So if m_0 is not polynomial in the input, the algorithm cannot PAC learn efficiently.