MODULE 3

Computational learning theory

We don’t want to phrase the definition in terms of games, so it’s time to remove the players from the picture. What we’re really concerned with is whether there’s an algorithm which can produce good hypotheses when given random data. But we have to solidify the “giving” process and exactly what limits are imposed on the algorithm.

It sounds daunting, but the choices are quite standard as far as computational complexity goes. Rather than say the samples come as a big data set as they might in practice, we want the algorithm to be able to decide how much data it needs. To do this, we provide it with a *query function* which, when accessed, spits out a sample in unit time. Then we’re interested in learning the concept with a reasonable number of calls to the query function.

And now we can iron out those words like “some” and “small” and “high” in our working definition. Since we’re going for small error, we’ll introduce a parameter $0<\varepsilon<1/2$ to represent our desired error bound. That is, our goal is to find a hypothesis h such that $errc,D(h)\leq\varepsilon$ with high probability. And as ε gets smaller and smaller (as we expect more and more of it), we want to allow our algorithm more time to run, so we limit our algorithm to run in time and space polynomial in $1/\varepsilon$.

We need another parameter to control the “high probability” part as well, so we’ll introduce $0<\delta<1/2$ to represent the small fraction of the time we allow our learning algorithm to have high error. And so our goal becomes to, with probability at least $1–\delta$, produce a hypothesis whose error is less than ε. In symbols, we want

$$PD(errc,D(h)\leq\varepsilon)>1–\delta$$

Note that the PD refers to the probability over which samples you happen to get when you call the query function (and any other random choices made by the algorithm). The “high probability” hence refers to the unlikely event that you get data which is unrepresentative of the distribution generating it. Note that this is *not* the probability over which distribution is chosen; an algorithm which learns must still learn no matter what D is.

And again as we restrict δ more and more, we want the algorithm to be allowed more time to run. So we require the algorithm runs in time polynomial in both $1/\varepsilon,1/\delta$.

And now we have all the pieces to state the full definition.

Definition: Let X be a set, and C be a concept class over X. We say that C is *PAC-learnable* if there is an algorithm $A(\varepsilon,\delta)$ with access to a query function for C and runtime $O(poly(1\varepsilon,1\delta))$, such that for all $c\in C$, all distributions D over X, and all inputs ε, δ between 0 and 1/2, the probability that A produces a hypothesis h with error at most ε is at least $1-\delta$. In symbols,

$$P_D(P_{x \sim D}(h(x) \neq c(x)) \leq \varepsilon) \geq 1-\delta$$

where the first P_D is the probability over samples drawn from D during the execution of the program to produce h. Equivalently, we can express this using the error function,

$$P_D(\text{err}_{c,D}(h) \leq \varepsilon) \geq 1-\delta$$

Excellent.

Intervals are PAC-Learnable

Now that we have this definition we can return to our problem of learning intervals on the real line. Our concept class is the set of all characteristic functions of intervals (and we'll add in the empty set for the default case). And the algorithm we proposed to learn these intervals was quite simple: just grab a bunch of sample points, take the biggest and smallest positive examples, and use those as the endpoints of your hypothesis interval.

Let's now *prove* that this algorithm can learn any interval with any distribution over real numbers. This proof will have the following form:

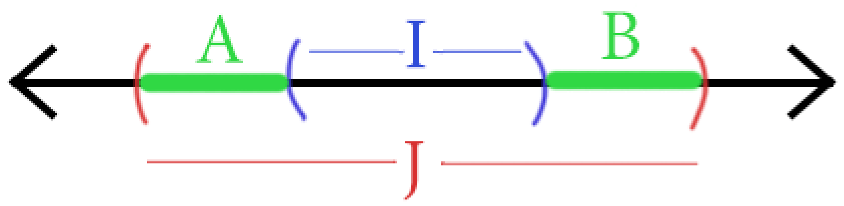
- Leave the number of samples you pick arbitrary, say m.
- Figure out the probability that the total error of our produced hypothesis is $>\varepsilon$ in terms of m.
- Pick m to be sufficiently large that this event (failing to achieve low error) happens with small probability.

So fix any distribution D over the real line and say we have our m samples, we picked the max and min, and our interval is $I=[a_1,b_1]$ when the target concept is $J=[a_0,b_0]$. We can notice one thing, that our hypothesis is contained in the true interval, $I \subset J$. That's because the sample never lie, so the largest sample we saw must be smaller than the largest possible positive example, and vice versa. In other words $a_0<a_1<b_1<b_0$. And so the probability of our hypothesis producing an error is just the probability that D produces a positive example in the two intervals $A=[a_0,a_1], B=[b_1,b_0]$.

This is all setup for the second bullet point above. The total error is at most the sum of the probabilities that a positive sample shows up in each of A,B separately.

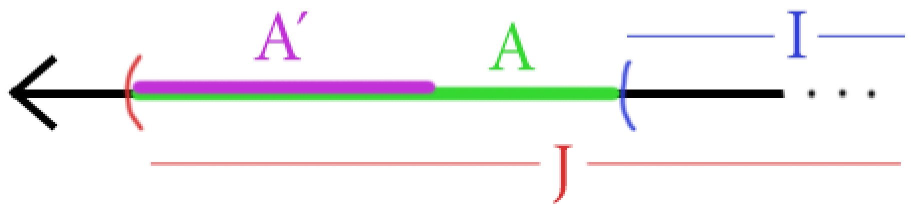
$$\text{err}_{J,D} \leq P_{x \sim D}(x \in A) + P_{x \sim D}(x \in B)$$

Here's a picture.



The two green intervals are the regions where error can occur.

If we can guarantee that each of the green pieces is smaller than $\varepsilon/2$ with high probability, then we'll be done. Let's look at A, and the same argument will hold for B. Define A′ to be the interval [a0,y] which is so big that the probability that a positive example is drawn from A′ under D is *exactly* $\varepsilon/2$. Here's another picture to clarify that.



The pink region A' has total probability weight epsilon/2, and if the green region A is larger, we risk too much error to be PAC-learnable.

We'll be in great shape if it's already the case that $A \subset A'$, because that implies the probability we draw a positive example from A is at most $\varepsilon/2$. So we're worried about the possibility that $A' \subset A$. But this can only happen if we never saw a point from A′ as a sample during the run of our algorithm. Since we had m samples, we can compute in terms of m the probability of never seeing a sample from A′.

The probability of a single sample not being in A′ is just $1-\varepsilon/2$ (by definition!). Recalling our basic probability theory, two draws are independent events, and so the probability of missing A′ m times is equal to the product of the probabilities of each individual miss. That is, the probability that our chosen A contributes error greater than $\varepsilon/2$ is at most

$$PD(A' \subset A) \leq (1-\varepsilon/2)m$$

The same argument applies to B, so we know by the union bound that the probability of error $>\varepsilon/2$ occurring in either A or B is at most the sum of the probabilities of large error in each piece, so that

$$PD(errJ,D(I) > \varepsilon) \leq 2(1-\varepsilon/2)m$$

Now for the third bullet. We want the chance that the error is big to be smaller than δ, so that we'll have low error with probability $>1-\delta$. So simply set

$$2(1-\varepsilon/2)m \le \delta$$

And solve for m. Using the fact that $(1-x) \le e-x$ (which is proved by Taylor series), it's enough to solve

$$2e-\varepsilon m/2 \le \delta,$$

And a fine solution is $m \ge (2/\varepsilon \log(2/\delta))$.

Now to cover all our bases: our algorithm simply computes m for its inputs ε, δ, queries that many samples, and computes the tightest-fitting interval containing the positive examples. Since the number of samples is polynomial in $1/\varepsilon, 1/\delta$ (and our algorithm doesn't do anything complicated), we comply with the time and space bounds. And finally we just proved that the chance our algorithm will misclassify an ε fraction of new points drawn from D is at most δ. So we have proved the theorem:

Theorem: Intervals on the real line are PAC-learnable.

As an exercise, see if you can generalize the argument to axis-aligned rectangles in the plane. What about to arbitrary axis-aligned boxes in d dimensional space? Where does d show up in the number of samples needed? Is this still efficie**First-Order Logic:**
All expressions in first-order logic are composed of the following attributes:

1. constants — e.g. tyler, 23, a
2. variables — e.g. A, B, C
3. predicate symbols — e.g. male, father (True or False values only)
4. function symbols — e.g. age (can take on any constant as a value)
5. connectives — e.g. ∧, ∨, ¬, →, ←
6. quantifiers — e.g. ∀, ∃

Term: It can be defined as any constant, variable or function applied to any term. e.g. **age(bob)**
Literal: It can be defined as any predicate or negated predicate applied to any terms. **e.g. female(sue), father(X, Y)**
It has 3 types:

- **Ground Literal —** a literal that contains no variables. e.g. female(sue)
- **Positive Literal —** a literal that does not contain a negated predicate. e.g. female(sue)
- **Negative Literal —** a literal that contains a negated predicate. e.g. father(X,Y)

Clause – It can be defined as any disjunction of literals whose variables are universally quantified.

```
where,


M₁, M₂, ...,Mₙ --> literals (with variables universally quantified)
V --> Disjunction (logical OR)
```

Horn clause — It can be defined as any clause containing exactly one positive literal.

Since,

and

Then the horn clause can be written as

```
where,

H --> Horn Clause
L₁,L₂,...,Lₙ --> Literals

(A ⇠ B) --> can be read as 'if B then A' [Inductive Logic]

and

∧ --> Conjunction (logical AND)
∨ --> Disjunction (logical OR)
¬ --> Logical NOT
```

First Order Inductive Learner (FOIL)

In machine learning, first-order inductive learner (FOIL) is a rule-based learning algorithm. It is a natural extension of SEQUENTIAL-COVERING and LEARN-ONE-RULE algorithms. It follows a Greedy approach.

Inductive Learning:

Inductive learning analyzing and understanding the evidence and then using it to determine the outcome. It is based on Inductive Logic.

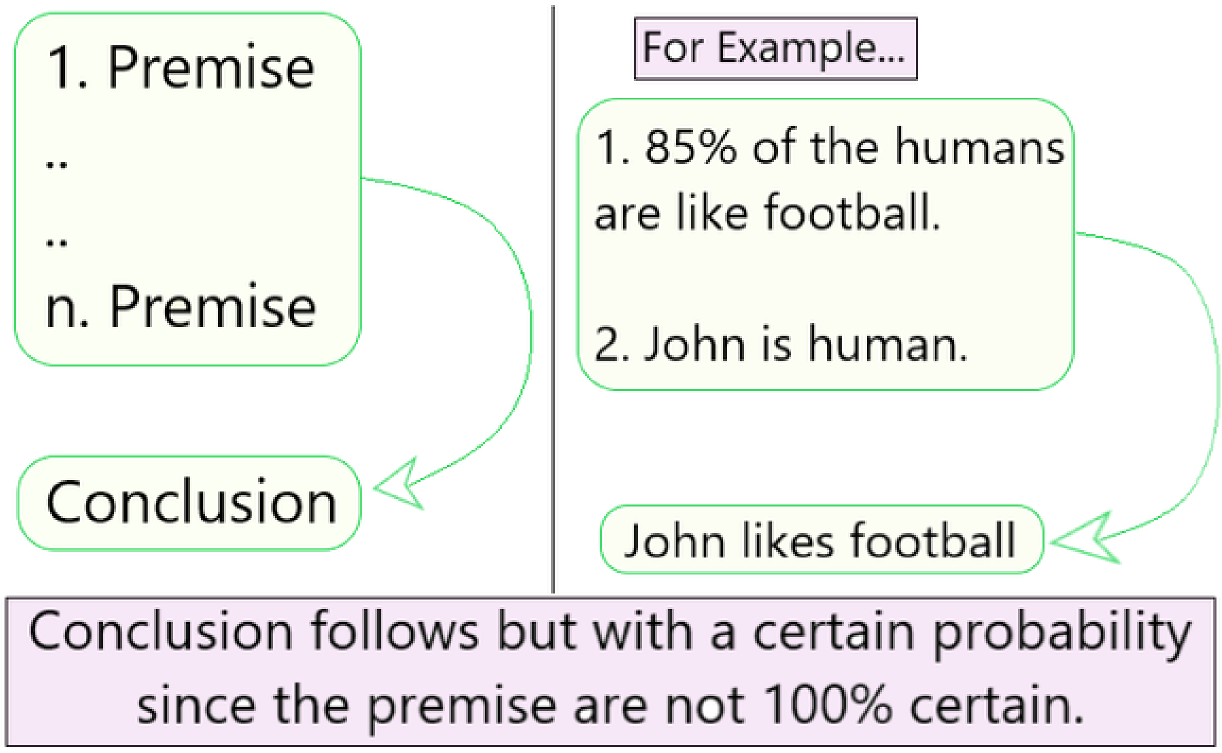

1. Premise
..
..
n. Premise
Conclusion
For Example...
1. 85% of the humans are like football.
2. John is human.
John likes football
Conclusion follows but with a certain probability since the premise are not 100% certain.

Fig 1: Inductive Logic

Algorithm Involved

```
FOIL(Target predicate, predicates, examples)

• Pos ← positive examples
• Neg ← negative examples
• Learned rules ← {}
• while Pos, do
    //Learn a NewRule
    – NewRule ← the rule that predicts target-predicate with no
preconditions

    – NewRuleNeg ← Neg
    – while NewRuleNeg, do
        Add a new literal to specialize NewRule
        1. Candidate_literals ← generate candidates for newRule based
on Predicates
        2. Best_literal ←
                 $\mathrm{argmax}_{L \in \mathrm{Candidate\ literals}}$ Foil_Gain(L,NewRule)

        3. add Best_literal to NewRule preconditions
        4. NewRuleNeg ← subset of NewRuleNeg that satisfies NewRule
preconditions
    – Learned rules ← Learned rules + NewRule
    – Pos ← Pos – {members of Pos covered by NewRule}

• Return Learned rules
```

Working of the Algorithm:

In the algorithm, the inner loop is used to generate a new best rule. Let us consider an example and understand the step-by-step working of the algorithm.

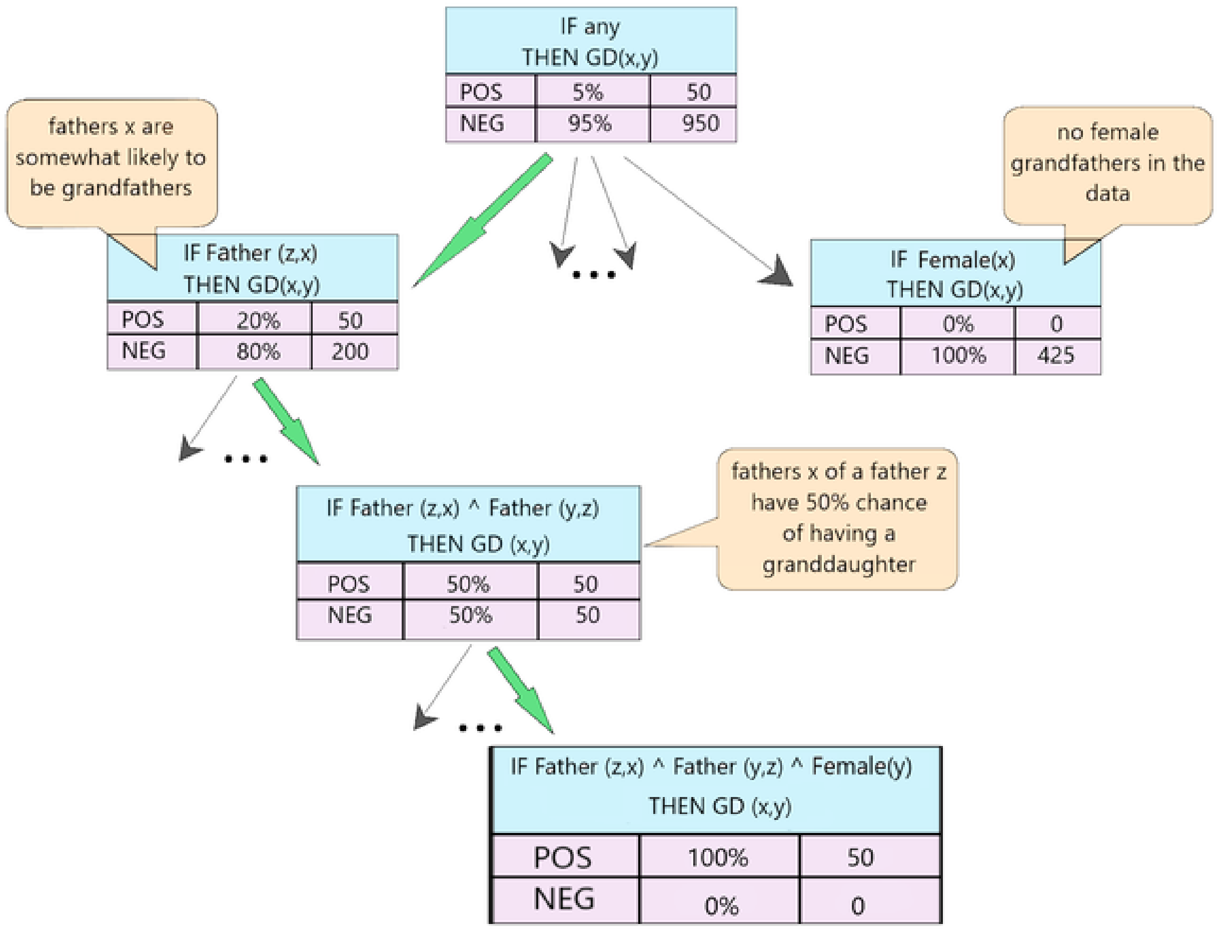

IF any THEN GD(x,y)
POS 5% 50
NEG 95% 950
fathers x are somewhat likely to be grandfathers
IF Father (z,x) THEN GD(x,y)
POS 20% 50
NEG 80% 200
no female grandfathers in the data
IF Female(x) THEN GD(x,y)
POS 0% 0
NEG 100% 425
IF Father (z,x) ^ Father (y,z) THEN GD (x,y)
POS 50% 50
NEG 50% 50
fathers x of a father z have 50% chance of having a granddaughter
IF Father (z,x) ^ Father (y,z) ^ Female(y) THEN GD (x,y)
POS 100% 50
NEG 0% 0

Fig 2 : FOIL Example

Say we are tying to predict the **Target-predicate-** *GrandDaughter(x,y)*. We perform the following steps: **[Refer Fig 2]**

Step 1 - NewRule = GrandDaughter(x,y)
Step 2 -

2.a - Generate the candidate_literals.
(Female(x), Female(y), Father(x,y), Father(y.x), Father(x,z), Father(z,x), Father(y,z), Father(z,y))

2.b - Generate the respective candidate literal negations.
(¬Female(x), ¬Female(y), ¬Father(x,y), ¬Father(y.x),

¬Father(x,z), ¬Father(z,x), ¬Father(y,z), ¬Father(z,y))

Step 3 - FOIL might greedily select Father(x,y) as most promising, then
NewRule = GrandDaughter(x,y) ← Father(y,z) [Greedy approach]

Step 4 - Foil now considers all the literals from the previous step as well as:
(Female(z), Father(z,w), Father(w,z), etc.) and their negations.

Step 5 - Foil might select Father(z,x), and on the next step Female(y) leading to
NewRule = GrandDaughter (x,y) ← Father(y,z) ∧ Father(z,x) ∧ Female(y)

Step 6 - If this greedy approach covers only positive examples it terminates
the search for further better results.

FOIL now **removes all positive examples covered by this new rule.**
If more are left then the outer while loop continues.

FOIL: Performance Evaluation Measure

The performance of a new rule is not defined by its entropy measure (like the *PERFORMANCE* method in Learn-One-Rule algorithm).
FOIL uses a gain algorithm to determine which new specialized rule to opt. Each rule's utility is estimated by the number of bits required to encode all the positive bindings. **[Eq.1]**

where,

L is the candidate literal to add to rule R

p_0 = number of positive bindings of R
n_0 = number of negative bindings of R
p_1 = number of positive binding of R + L
n_1 = number of negative bindings of R + L
t = number of positive bindings of R also covered by R + L

FOIL Algorithm is another rule-based learning algorithm that extends on the Sequential Covering + Learn-One-Rule algorithms and uses a different Performance metrics (other than entropy/information gain) to determine the best rule possible.