Advanced Machine Learning

Bogdan Alexe,

bogdan.alexe@fmi.unibuc.ro

University of Bucharest, 2nd semester, 2021-2022

Assignment 1 submissions

Nr. Crt.	Student	Group
1	Băbeanu C Andrei	405
2	Danci C.I. Laurențiu-Cristian	406
3	Săvoiu V.C. Valentin- Marian	406
4	Ahmad Mustapha Wali	407
5	Aldea D Gabriela-Dorina	407
6	Badea A.C Adrian	407
7	BADEA M.F. GEORGE-ALEXANDI	407
8	Buican M Laura Andreea	407
9	Căpriță I Nicușor-Cătălin	407
10	Chiripuci I Adelina	407
11	Chițu Ș. Ștefan Cătălin	407
12	Ciripan I Ioana-Rodica	407
13	Dăscălescu D Dana	407
14	Dumitrescu D Teodor	407
15	Duncea R.I. Vlad-Alexandru	407
16	Ghiță A Alexandru Ștefan	407
17	Ioan IM Diana-Alina	407
18	Iordăchescu A.C. Anca-Mihaela	407
19	Lazăr O.A. Dorian	407
20	Masgras I.N. Vlad-Florian	407
21	Matei L Alexandru Cristian	407
22	Mihai G Marian Vlad	407
23	Mușat T. Bianca-Ștefania	407
24	Nicolae M.A. Andrei	407
25	Pronoza D.V. Cristian-Valentin	407
26	Radu I Vlad-Andrei	407
27	Ranete V Cristian	407
28	Sava I Vasile-Daniel	407
29	Smarandache D Mihnea	407
30	Stoica C Elena-Maria	407
31	Tălmăcel C.L. Sergiu-Victor	407
32	Tomeh Lara	407
33	Vasile D.C. Andrei-Valentin	408
34	Al Othmani Yasmin	411
35	Bechir S Sibel	411
36	Brâncoveanu J.-B. Anca-Maria	411
37	Condrat F Mihai	411
38	Guiță F.D. Bianca-Oana	411
39	Ionașcu G Mihai-Gabriel	411
40	Istrati M Lucian	411
41	Oprea C.GH Camelia-Elena	411
42	Pațilea C Cătălina Camelia	411
43	Petre O.D. Tudor	411
44	Poenaru G Gabriela-Cătălina	411
45	Antal-Burlacu Mihaela	412
46	Creanga Claudiu	412
47	Zăvelcă Miruna-Andreea	412

Recap - Computational resources of learning

For learning we need 2 type of resources:

1. *Information = training data*
    – so far we analyzed how much training data (sample size) we need in order to learn
    – *sample complexity*

2. *Computation = runtime*
    – for how much time an algorithm (that implements learning) will run, once we have sufficiently many training examples
    – *computational complexity*
    – crucial when we need fast ML applications (driver surveillance, stock exchange trading, etc)
    – runtime = number of elementary instructions executed - arithmetic operations over real numbers - in an asymptotic sense (with respect to input size) of the algorithm, e.g. *O(n) – where n is the size of the input size*

Recap - Input size parameter of learning

What should play the role of the input size parameter in learning?

- accuracy ε, confidence δ and another parameter n related to the size/ complexity of $\mathcal{X}, \mathcal{H}$
  - how much computation we need in order to get accuracy ε with confidence δ
  - want to have *efficient learning* (give a formal definition later): polynomial in $1/\varepsilon$, $1/\delta$ and n (some parameter related to the size/complexity of domain/hypothesis class: more complex hypothesis needs more computation time)

- we study the runtime in an asymptotic sense by defining a sequence of pairs $(\mathcal{X}_n, \mathcal{H}_n)_{n=1,2,...}$ and studying asymptotic complexity of learning $\mathcal{X}_n, \mathcal{H}_n$ as n grows to ∞

Formal definition of computational complexity

Definition

Given a function $f : (0,1)^2 \to \mathbb{N}$, a learning task $(\mathcal{Z}, \mathcal{H}, l)$, and a learning algorithm A, we say that A solves the learning task in time $O(f)$ if there exists some constant number c, such that for every probability distribution $\mathcal{D}$ over $\mathcal{Z}$, and input $\varepsilon, \delta \in (0,1)$, when A has access to samples generated i.i.d by $\mathcal{D}$, we have that:

- A terminates after performing at most $c * f(\varepsilon, \delta)$ operations;

- the output of A, denoted h_A, can be applied to predict the label of a new example while performing at most $c * f(\varepsilon, \delta)$ operations;

- the output of A is probably approximately correct; namely, with probability of at least $1 - \delta$ (over the random samples A receives):
$$L_{\mathcal{D}}(h_A) \leq \min_h L_{\mathcal{D}}(h) + \varepsilon$$

Recap - Formal definition of efficient learning

Definition (for graded hypothesis spaces)

Consider a sequence of learning problems, $(\mathcal{Z}_n, \mathcal{H}_n, l_n)_{n=1,2,\ldots}$ where problem n is defined by a domain $\mathcal{Z}_n$, a hypothesis class $\mathcal{H}_n$, and a loss function l_n. Let A be a learning algorithm designed for solving learning problems of this form. Given a function $g: N \times (0,1)^2 \rightarrow N$, we say that the runtime of A with respect to the preceding sequence is $O(g)$, if for all n, A solves the problem $(\mathcal{Z}_n, \mathcal{H}_n, l_n)$ in time $O(f_n)$, where $f_n : (0,1)^2 \rightarrow N$ is defined by $f_n(\varepsilon, \delta) = g(n, \varepsilon, \delta)$.

We say that A is an *efficient* PAC algorithm with respect to a sequence $(\mathcal{Z}_n, \mathcal{H}_n, l_n)$ if its runtime is $O(p(n, 1/\varepsilon, 1/\delta))$ for some polynomial p.

Formal definition of efficient PAC learning (Valiant 1984)

In 1984, Leslie Valiant defined efficient PAC learning: PAC learnability + require the number of examples and the runtime of the algorithm A (training + testing) to be polynomial in $1/\varepsilon$, $1/\delta$, n.

Example:

- $\mathcal{U}_n = \{h\colon B^n \rightarrow \{0,1\}\}$ - the concept class formed by all subsets of B^n
- $|\mathcal{U}_n| = 2^{2^n}$ – finite, so is PAC learnable with $m_{\mathcal{H}}(\varepsilon, \delta)$ in the order of m:

$$m \geq \left\lceil \frac{1}{\varepsilon}\left(2^n \log(2) + \log(\frac{1}{\delta})\right)\right\rceil$$

- VCdim($\mathcal{U}_n$) = 2^n is not polynomial in n
- sample complexity exponential in n, number of variables
- it is not efficient PAC-learnable in any practical sense (need polynomial sample complexity)

Recap - Examples

1. $\mathcal{H}_{conj}{}^n$ = class of conjunctions of at most n Boolean literals $x_1, \ldots, x_n$
    - in the realizable case (PAC learning) the runtime of an algorithm that finds an ERM hypothesis is $O(m_{\mathcal{H}}(\varepsilon,\delta)*n)$, so is polynomial in $1/\varepsilon$, $1/\delta$, n
    - in the agnostic (unrealizable) case: unless P = NP, there is no algorithm whose running time is polynomial in $m_{\mathcal{H}}(\varepsilon,\delta)$ and n that is guaranteed to find an ERM hypothesis for the class of Boolean conjunctions.

2. $\mathcal{H}_{rec}{}^n$ = the class of axis aligned rectangles in $\mathbf{R}^n$
    - in the realizable case (PAC learning) the runtime of an algorithm that finds an ERM hypothesis is $O(m_{\mathcal{H}}(\varepsilon,\delta)*n)$, so is polynomial in $1/\varepsilon$, $1/\delta$, n (the algorithm has to find the minimal and the maximal values among the positive instances in the training sequence).
    - in the agnostic case for every fixed n, ERM_H can be implemented in time which is polynomial in $1/\varepsilon$, $1/\delta$, n (measures the complexity of the $\mathcal{H}_{rec}{}^n$) therefore we have efficient learning
    - however, as a function of n the runtime of the algorithm implementing the ERM_H presented is exponential in n. It can be proved that there is no better algorithm (unless P = NP) than the one proposed.

Relation between consistency and PAC learning

- a learning rule is consistent if:
  - input: $\mathcal{H}$ and $S = (x_1, y_1), \ldots, (x_m, y_m)$
  - output: $h \in \mathcal{H}$, h is an ERM hypothesis, i.e. $h(x_i) = y_i$

- if VCdim($\mathcal{H}$) $\leq$ d then $\mathcal{H}$ is PAC learnable (in the realizable case) by the consistent rule with sample complexity (given by the fundamental theorem):

$$C_1 \frac{d + \log(1/\delta)}{\epsilon} \leq m_{\mathcal{H}}(\epsilon, \delta) \leq C_2 \frac{d \log(1/\epsilon) + \log(1/\delta)}{\epsilon}$$

- if d is polynomial in n and the consistent rule has runtime polynomial in the sample size $m_{\mathcal{H}}(\varepsilon, \delta)$ then we have efficient PAC learning

- so, efficient 'consistent-hypothesis-finder' → efficient PAC learning

- does the converse implication holds?
  - yes, based on randomized algorithms

Today's lecture: Overview

- Randomized algorithms

- Intractability of learning

- Improper learning

Randomized algorithms

- a randomized algorithm A is allowed to use random numbers as part of its input

- the output of the algorithm A depends on the input, so it depends on the particular sequence produced by a random number generator

- we can speak of the probability that A has a given outcome

- a randomized algorithm A 'solves' a binary decision problem (YES/NO) if it behaves in the following way:
  - the algorithm always halts and produces an output (in polynomial time)
  - if the output of A is YES then the answer is 100% right
  - if the output of A is NO then the answer is right with probability $\geq 50\%$

- randomized algorithm for the problem "number n is composite (is not prime)"
  - if the output is YES then we are sure that number n is composite (not prime)
  - if the output is NO then with probability $\geq 50\%$ number n is not composite (is prime)

- practical usefulness of randomized algorithms: repeating the algorithm several times dramatically increases the likelihood of being correct
  - if I get only NO's in m attempts then with probability of error $\leq 0.5^m$ can say that n is not composite (prime)

Randomized algorithms – primality testing

- decide wheather or not a number n is prime
- applications in cryptography

Algorithm 1.1.1 (Trial Division)
INPUT: Integer $n \geq 2$.
METHOD:
```
0      i: integer;
1      i ← 2;
2      while i · i ≤ n repeat
3          if i divides n
4              then return 1;
5          i ← i + 1;
6      return 0;
```

- n = 7483845764874895490005046457879234760435948750902645 2654305481

- n has 62 digits, sqrt(n) has 31 digits

- the basic algorithm takes 10^{13} years to output 0, n is a prime number

- efficient algorithm to primality testing?

Randomized algorithms – primality testing

- efficient algorithm to test a number being composite (not prime)?

Algorithm 1.2.1 (Lehmann's Primality Test)

INPUT: Odd integer $n \geq 3$, integer $\ell \geq 2$.
METHOD:
```
0      a, c: integer; b[1..ℓ]: array of integer;
1      for i from 1 to ℓ do
2          a ←  a randomly chosen element of {1,...,n − 1};
3          c ← a^((n−1)/2) mod n;
4          if c ∉ {1, n − 1}
5              then return 1;
6              else b[i] ← c;
7      if b[1] = ··· = b[ℓ] = 1
8          then return 1;
9          else return 0;
```

- the algorithm returns 1 = it takes the decision that the number is composite (not prime)
- the algorithm returns 0 = it takes the decision that the number is not composite (prime)
- repeat for l times lines 2-6
  - take a random number a in $\{1,\ldots, n\text{-}1\}$ and compute $c = \mathbf{a^{(n\text{-}1)/2} \ mod \ n}$
  - if n is prime then c should be 1 or n-1 (50% - 50% if a is random)
  - n = 7: 1^3 mod 7 = 1, 2^3 mod 7 = 1, 3^3 mod 7 = 6, 4^3 mod 7 = 1, 5^3 mod 7 = 6, 6^3 mod 7 = 6
  - line 5: return 1 if c is not 1 or n-1, so we are sure that n is composite (not prime)
  - if among all c there is at least one = n-1 return 0 (not composite), otherwise return 1 (composite)
  - could have errors

Randomized algorithms – primality testing

- probability of making an error?

- case 1: n is prime (the desired output is 0)
  - 50% chance that c is 1
  - 50% chance that c is n-1
  - error if all c = 1 with probability 2^{-l}

Algorithm 1.2.1 (Lehmann's Primality Test)

INPUT: Odd integer $n \geq 3$, integer $\ell \geq 2$.

METHOD:
```
0    a, c: integer; b[1..ℓ]: array of integer;
1    for i from 1 to ℓ do
2        a ← a randomly chosen element of {1,...,n−1};
3        c ← a^((n−1)/2) mod n;
4        if c ∉ {1,n−1}
5            then return 1;
6            else b[i] ← c;
7    if b[1] = ··· = b[ℓ] = 1
8        then return 1;
9        else return 0;
```

- case 2: n is composite (the desired output is 1)
  - make an error if all c are 1 and n-1
  - it can be shown that if there exist some number a in $\{1, \ldots, n-1\}$ that satisfies $a^{(n-1)/2} \bmod n = n-1$, then more than half of the elements in $\{1, \ldots, n-1\}$ satisfy $a^{(n-1)/2} \bmod n \notin \{1, n-1\}$.
  - this means that the probability that the loop in lines 1–6 runs for l rounds is no more than 2^{-l}
  - the probability that output 0 is produced cannot be larger than 2^{-l}
  - take l around 1000, the probability will be very closed to 100% for practical purposes

Randomized algorithms – primality testing

- computational complexity?

- do rapid exponentiation in line 3
  - $2 \log_2 n$ multiplications and divisions of numbers less than n^2

- represent a number n on $\log_{10}(n)$ digits

- overall we have $O((\log_{10} n)^3)$ – suitable on modern computers

Algorithm 1.2.1 (Lehmann's Primality Test)

INPUT: Odd integer $n \geq 3$, integer $\ell \geq 2$.

METHOD:

```
0       a, c: integer; b[1..ℓ]: array of integer;
1       for i from 1 to ℓ do
2           a ←  a randomly chosen element of {1,…,n−1};
3           c ← a^((n−1)/2) mod n;
4           if c ∉ {1, n−1}
5               then return 1;
6               else b[i] ← c;
7       if b[1] = ⋯ = b[ℓ] = 1
8           then return 1;
9           else return 0;
```

P $\subseteq$ RP $\subseteq$ NP

- P = class of decision problems that are quickly (polynomial runtime) solvable

- RP = class of decision problems that are quickly (polynomial runtime) solvable by randomized algorithms

- NP = class of decision problems that are quickly checkable (polynomial time)

We have that:
- P $\subseteq$ RP, as we use the same algorithm with polynomial time to solve a problem from P that does not used the random inputs

- RP $\subseteq$ NP, as the solution of the randomized algorithm can be quickly (in polynomial time) checkable

- It is believed that P $\neq$ NP

P = RP ≠ NP ?

- The scientific comunity believes that the following relation holds:
$$P = RP \neq NP$$

- In 2002, formal proof that there exist a polynomial algorithm to check the primality of a number. This indirectly suggests that P = RP.

PRIMES is in P

Manindra Agrawal   Neeraj Kayal
Nitin Saxena*

Department of Computer Science & Engineering
Indian Institute of Technology Kanpur
Kanpur-208016, INDIA
Email: {manindra,kayaln,nitinsa}@iitk.ac.in

Abstract

We present an unconditional deterministic polynomial-time algorithm that determines whether an input number is prime or composite.

Relation between consistency and PAC learning

Theorem

Let $\mathcal{H}$ be a hypothesis class that is efficient PAC learnable. Then, there exists a randomized algorithm which solves the problem of finding a hypothesis in $\mathcal{H}$ consistent with a given training sample, and which has runtime polynomial in m (the length of the training sample).

Proof

Let S^* be a training sample, with $|S^*| = m^*$ and let h^* be the function that generates the labels on points in S^*.

Consider the following probability distribution over domain $\mathcal{X}$:

$$\mathcal{D}(x) = \begin{cases} 1/m^*, & \text{if x occurs in } S^* \\ 0, & \text{otherwise} \end{cases}$$

Relation between consistency and PAC learning

$\mathcal{H}$ is efficient PAC learnable, so there exists a learning algorithm A such that:
given $\varepsilon = 1/(m*+1)$, $\delta = \frac{1}{2}$, $f = h*$, *distribution* $\mathcal{D}$ over $\mathcal{X}$ when we run the learning algorithm A on a training set S, consisting of m $\geq$ $m_{\mathcal{H}}(1/(m*+1), 1/2)$ examples sampled i.i.d. from $\mathcal{D}$ and labeled by $h*$ the algorithm A returns a hypothesis $h_S \in \mathcal{H}$ such that, with probability at least $1-1/2 = 1/2$ (over the choice of examples), $L_{D,h*}(h_S)$ $\leq 1/(m*+1)$.

We want to design a randomized algorithm A* for finding a hypothesis which agrees (training error = 0) with the given training sample S*. We do the following:
- compute m = $m_{\mathcal{H}}(1/(m*+1), 1/2)$
- construct a training sample S, |S| = m with sample i.i.d from $\mathcal{D}$
- run the given PAC learning algorithm A on S
- check if the hypothesis A(S) = h_S agrees with S* (the probability > $\frac{1}{2}$)
- if the hypothesis h_S does not agree with S* return NO. Else return h_S. h_S will make 0 error (based on $\mathcal{D}$) so is consistent.

A* will succeed with probability greater than $\frac{1}{2}$. If A is polynomial than also A* is polynomial.

Do we always have efficient PAC learning?

Theorem

Let $\mathcal{H}$ be a hypothesis class that is efficient PAC learnable. Then, there exists a randomized algorithm which solves the problem of finding a hypothesis in $\mathcal{H}$ consistent with a given training sample, and which has runtime polynomial in m (the length of the training sample).

We will show that there exist classes that are PAC learnable (in the realizable case) for which the sample complexity is polynomial but the runtime is not polynomial:

- learning sometimes is intractable

- "proper" learning vs "improper" learning

Intractability of learning

Learning 3-Term DNF

Consider $\mathcal{H}_{3DNF}{}^n$ = class of 3-term disjunctive normal form formulae consisting of hypothesis of the form h: $\{0,1\}^n \to \{0,1\}$,

$$h(\mathbf{x}) = A_1(\mathbf{x}) \vee A_2(\mathbf{x}) \vee A_3(\mathbf{x}),$$

where $A_1(\mathbf{x})$ is a Boolean conjunction (in $\mathcal{H}_{conj}{}^n$) with at most n Boolean literals $x_1, \ldots, x_n$.

The output of $h(\mathbf{x})$ is 1 if either $A_1(\mathbf{x})$ or $A_2(\mathbf{x})$ or $A_3(\mathbf{x})$ outputs the label 1. If all three conjunctions output the label 0 then $h(\mathbf{x}) = 0$.

$|\mathcal{H}_{3DNF}{}^n| = 3^{3n} < \infty$, so is PAC learnable with sample complexity $3n \log(3/\delta)/\varepsilon$ (polynomial in $1/\varepsilon$, $1/\delta$, n).

However, from the computational perspective, this learning problem is hard. There is no polynomial time algorithm (unless RP = NP) that properly learns from training data. So ERM_H is not efficient.

We will show that if $\mathcal{H}_{3DNF}{}^n$ is efficient PAC learnable then the problem of graph 3-coloring problem (which is shown to be NP-complete) is in RP.

Learning 3-Term DNF

Theorem

Unless NP has randomized polynomial time algorithms, there is no proper efficient PAC-learning algorithm for the class $\mathcal{H}_{3DNF}{}^n$.

Proof idea

We consider that we can efficiently PAC learn the class $\mathcal{H}_{3DNF}{}^n$. From the preceding theorem there exist a randomized algorithms A^* that can decide consistency of 3-term DNF.

We show that the 3-term DNF problem can be reduced to the problem of the graph 3-coloring problem (which is NP-complete). So there exist an algorithm in RP that solves a problem in NP. So $RP = NP$, which we consider to be false, as we believe that $P = RP \neq NP$.
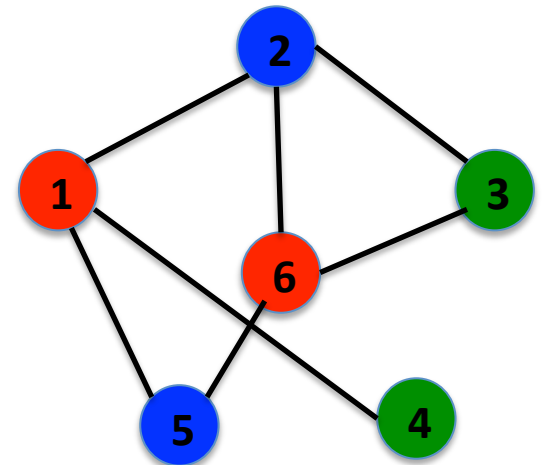
The graph 3-coloring problem

The graph 3-coloring problem.
Given as input an undirected graph $G = (V,E)$ with $V = \{1, 2, …, n\}$ and edge set $E \subseteq V \times V$, determine if there is an assignment of a color to each element of V such that at most 3 different colors are used, and for every edge $(u, v) \in E$, vertex u and vertex v are assigned different colors.
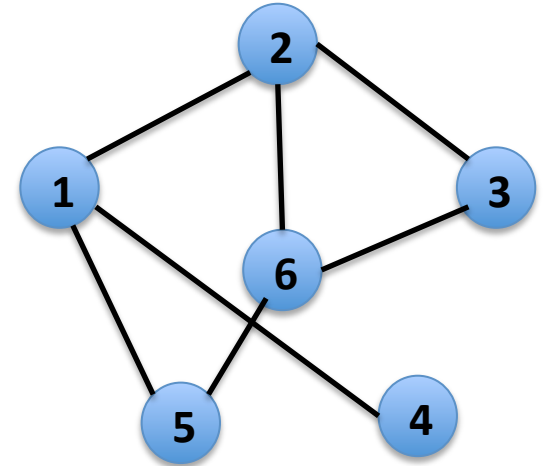
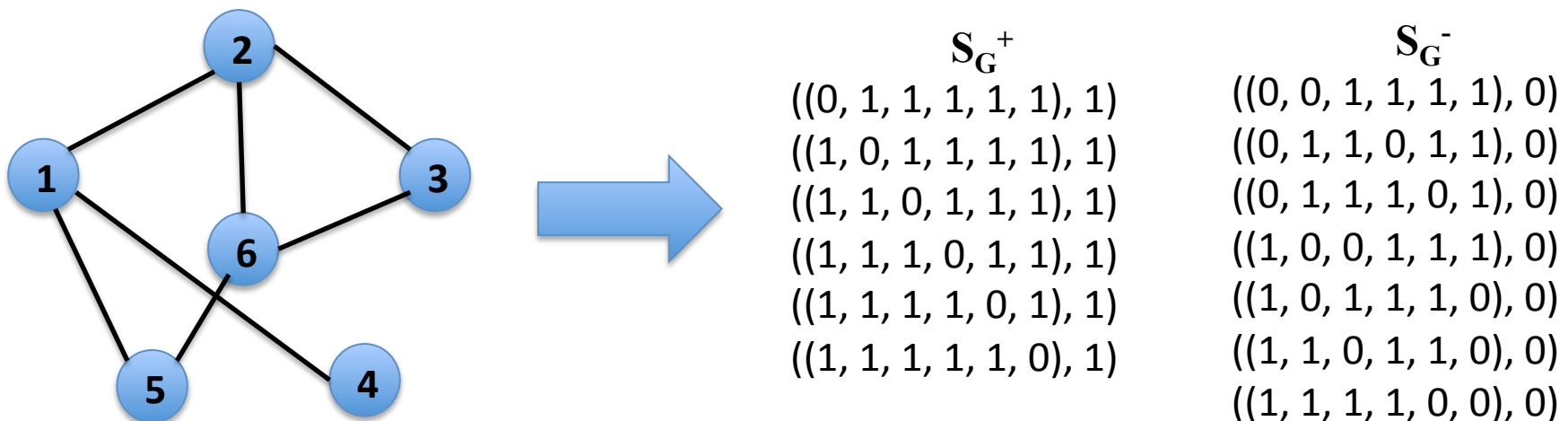$c : V \rightarrow \{R, G, B\}$ is a proper coloring function if for every $(u,v) \in E$ we have that $c(u) \neq c(v)$

Reduction of graph 3-coloring problem to the 3-term DNF problem

We reduce an instance G = (V, E) to a labeled set S_G of examples (of *n* bits) consisting of |V| positive examples (S_G^+) and |E| negative examples (S_G^-).

Each vertex *i* is encoded as the vector with 0 on position *i* and 1 on all other positions (0-hot encoding) and has label +1: $((1, 1,\ldots 1, 0, 1, \ldots 1), +1)$

Each edge $(i, j) \in$ E is encoded as the vector with 0 on positions *i* and *j* and 1 on all other positions and has label 0: $((1,\ldots 1, 0, 1, \ldots 1, 0, 1,\ldots,1), 0)$

S_G^+
((0, 1, 1, 1, 1, 1), 1)
((1, 0, 1, 1, 1, 1), 1)
((1, 1, 0, 1, 1, 1), 1)
((1, 1, 1, 0, 1, 1), 1)
((1, 1, 1, 1, 0, 1), 1)
((1, 1, 1, 1, 1, 0), 1)

S_G^-
((0, 0, 1, 1, 1, 1), 0)
((0, 1, 1, 0, 1, 1), 0)
((0, 1, 1, 1, 0, 1), 0)
((1, 0, 0, 1, 1, 1), 0)
((1, 0, 1, 1, 1, 0), 0)
((1, 1, 0, 1, 1, 0), 0)
((1, 1, 1, 1, 0, 0), 0)

Reduction of graph 3-coloring problem to the 3-term DNF problem

The graph G is 3-colorable $\Leftrightarrow$ $S_G = S_G^+ \cup S_G^-$ is consistent with some 3-term DNF formula.

Proof "$\Longrightarrow$" Suppose graph G is 3-colorable. Fix such a coloring.

Let R = set of all vertices with color red (= {1,6} in figure)

Let T_R the conjunction of all variables $x_1, x_2, \ldots, x_n$ whose index does not appear in R. ($T_R = x_2 \wedge x_3 \wedge x_4 \wedge x_5$)

Then, the positive example corresponding to a vertex i in R satisfies T_R because x_i does not appear in T_R.

Every negative example corresponding to an edge (i, j) does not satisfy T_R, as either x_i or x_j appear in T_R.
Do the same reasoning for colors green and blue and obtain T_B, T_G.
Take the 3-term DNF formula as $T_R \vee T_B \vee T_G$.
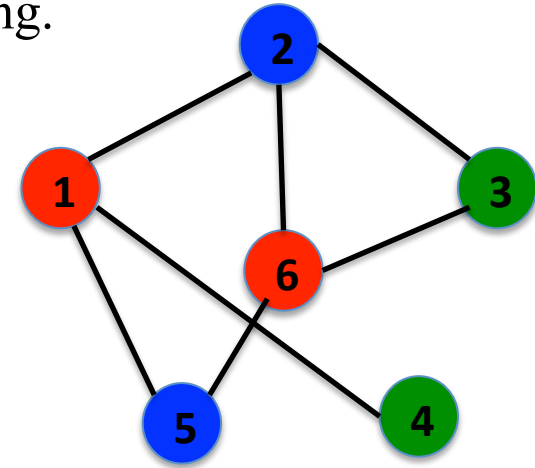
Reduction of graph 3-coloring problem to the 3-term DNF problem

The graph G is 3-colorable $\Leftrightarrow$ $S_G = S_G^+ \cup S_G^-$ is consistent with some 3-term DNF formula.

Proof "$\Leftarrow$" Suppose that the formula $T_R \vee T_B \vee T_G$ is consistent with S_G.

Define a coloring of G as follows: the color of vertex i is red/green/blue if the positive example corresponding to the vertex i satisfies $T_R/T_G/T_B$ (choose random color for ties).

If $i \neq j$ have the same color (let's say red) then:
$T_R(1, ..., 1, 0, 1, ..., 1, 1, 1, ..., 1) = 1$ (0 on position i)
$T_R(1, ..., 1, 1, 1, ..., 1, 0, 1, ..., 1) = 1$ (0 on position j)

So literals $x_i, \bar{x}_i, x_j, \bar{x}_j$ and $\bar{x}_k$ ($k \neq i, j$) cannot appear in T_R
So we have that:
$T_R(1, ..., 1, 0, 1, ..., 1, 0, 1, ..., 1) = 1$. So $(i, j) \notin E$.
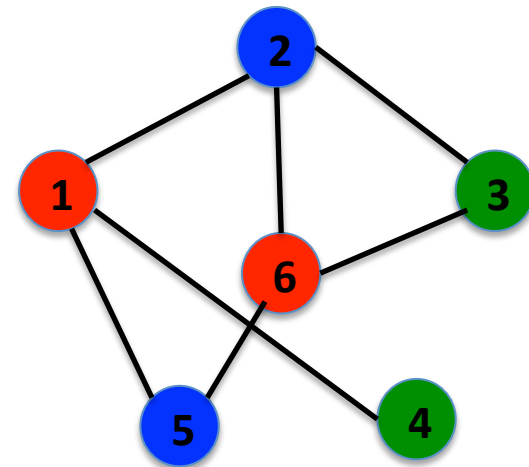
Main result

The graph G is 3-colorable $\Leftrightarrow$ $S_G = S_G^+ \cup S_G^-$ is consistent with some 3-term DNF formula.

So, the class $\mathcal{H}_{3DNF}^n$ is not efficient PAC learnable under the assumption that NP-complete problems cannot be solved with high probability by a probabilistic polynomial–time algorithm (technically, under the assumption RP $\neq$ NP).

The technique can be used to generalize the result for $\mathcal{H}_{kDNF}^n$ (k –term DNF formulae), where k $\geq$ 3.

"Consistent learning is hard" $\nRightarrow$ "learning is hard"

"Consistent learning is hard" $\Rightarrow$ "*proper* learning is hard"
A proper learning algorithm is a learning algorithm that must output h $\in \mathcal{H}$

Improper learning

Improper learning of 3-term DNFs

Use the distribution rule to obtain:
$(a \wedge b) \vee (c \wedge d) = (a \vee b) \wedge (a \vee d) \wedge (b \vee c) \wedge (b \vee d)$

Apply for the 3-term DNF formula to obtain a 3-CNF formulae:

$$A_1 \vee A_2 \vee A_3 = \bigwedge_{u \in A_1, v \in A_2, w \in A_3} (u \vee v \vee w)$$

So we have that a 3-term DNF can be viewed as a conjunction of $(2n)^3$ variables:

$$H_{3DNF}^n \subseteq H_{conj}^{(2n)^3} \qquad \left| H_{conj}^{(2n)^3} \right| = 3^{(2n)^3} + 1$$

We can efficiently PAC learn the new class of conjunctions, $H_{conj}^{(2n)^3}$ with sample complexity $(n^3 + \log(1/\delta))/\varepsilon$. The overall runtime of this approach is polynomial in $1/\varepsilon$, $1/\delta$, n. Pay polynomially in sample complexity, gain exponentially in computational complexity.

Improper learning
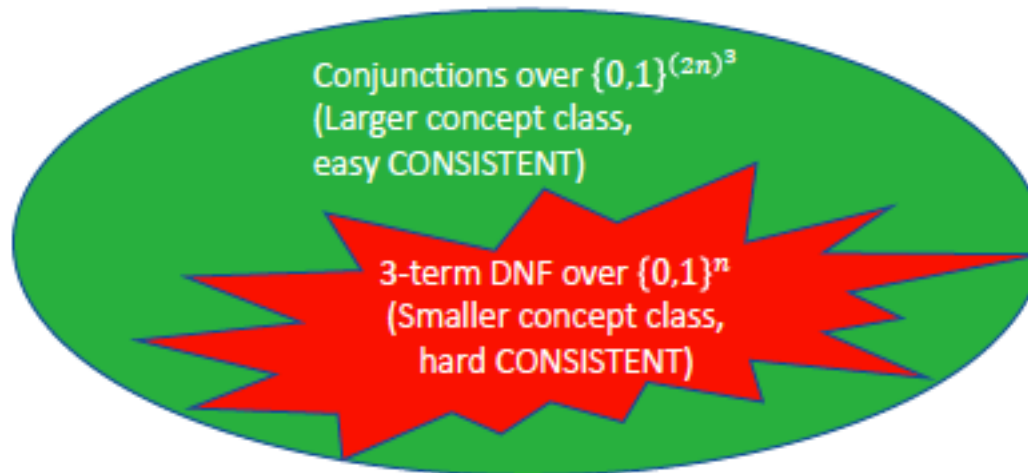
	3-term DNF	Conjunctions over $\{0,1\}^{(2n)^3}$	
Sample complexity	$O\left(\dfrac{n + log\frac{1}{\delta}}{\varepsilon}\right)$	$O\left(\dfrac{n^3 + log\frac{1}{\delta}}{\varepsilon}\right)$	🔺
CONSISTENT Computational-Complexity	NP-Hard	$O\left(n^3 \times \dfrac{n^3 + log\frac{1}{\delta}}{\varepsilon}\right)$	🔻

Next time