CSE 513T Theory of Artificial Intelligence & Machine Learning Jan 20, 2015

Lecture 3

Lecturer: Brendan Juba *Scribe: Tzai-Shuen Chen*

Today's topic: Basics of algorithms and complexity

1 Relative power of representations (Cont'd)

2 Efficient Computation: algorithms and Circuits

3 Computation Problems and NP-completeness

1 Relative power of representations (Cont'd)

1.1 recap

Last time we saw the elimination algorithm for learning conjunctions (ANDs of literals, where literals are variables or negations). By using the same algorithm as a subroutine, we can also learn disjunctions (ORs of literals), i.e., given access to examples $x^{(1)}, ..., x^{(m)}$ drawn from a common $\mathcal{D}$, and labeled by some $c \in \mathcal{C}$, $c(x^{(1)}, ..., c(x^{(m)}))$, the elimination algorithm would return a representation h such that $P_r[c(x) = h(x)] \geq 1 - \epsilon$ with probability $1 - \delta$. The idea is to apply De Morgan's Law: $l_1 \wedge l_2 \wedge ... \wedge l_n = \neg(\neg l_1 \vee \neg l_2 \vee ... \vee \neg l_n)$. For each given example $(x_1^{(i)}, ..., x_n^{(i)}, b^{(i)})$, we give the bitwise negation $(\neg x_1^{(i)}, ..., \neg x_n^{(i)}, \neg b^{(i)})$. Since there's a disjunction c such that $c(x^{(1)}) = b^{(1)}$, the negated examples are labeled by a conjunction. When the conjunction h is returned from the algorithm, we flip all of the labels' polarity to get a disjunction. Since there is a 1-1 correspondence between the negated examples and the original examples, errors in the resulting disjunction are made with the same probability as they would for the conjunction. Therefore, the bound on the error made by the conjunction of negated examples translates directly into a bound on the error of the disjunction. Thus, we have $P_r[c(x) = h(x)] \geq 1 - \epsilon$. We call using an algorithm for one problem as a subroutine to solve another problem a *reduction*.

1.2 Disjunctive Normal Form

In the following we would discuss another expression called disjunctive normal form, DNF. The DNF is an OR of ANDs of literals, e.g.,

$$\underbrace{(x_1 \wedge \neg x_3 \wedge x_7 \wedge x_{11})}_{\text{term of size 4}} \vee (x_2 \wedge x_3 \wedge_{42}) \vee x_4$$

The DNF can express both conjunctions and disjunctions. In fact, we can express every Boolean function f as a DNF [1]. To do this, we first select each row of the truth table that gives true value and make a conjunction of the literals (we call the conjunction a *term*). Then, we make a disjunction of these terms to form the DNF. Observe that:

- The DNF we construct is true at each point where target f is true, and
- since each of these terms is false everywhere but at a single point, it is false everywhere else.

The DNF we construct exactly represents the original Boolean function.

[1] see http://en.wikipedia.org/wiki/Canonical_normal_form

- Example:

x	y	z	f	*notation*
0	0	0	0	$\bar{x}\bar{y}\bar{z}$
0	0	1	1	$\bar{x}\bar{y}z$
0	1	0	0	$\bar{x}y\bar{z}$
0	1	1	0	$\bar{x}yz$
1	0	0	1	$x\bar{y}\bar{z}$
1	0	1	1	$x\bar{y}z$
1	1	0	0	$xy\bar{z}$
1	1	1	1	xyz

  Here we use the notations that $xy := x \wedge y; x + y := x \vee y$. To construct a DNF for f:

  - Step 1: select the rows where the value of the function is true, i.e., $\bar{x}\bar{y}z$, $x\bar{y}\bar{z}$, $x\bar{y}z$, and xyz.
  - Step 2: make a disjunction of these terms, $f = \bar{x}\bar{y}z + x\bar{y}\bar{z} + x\bar{y}z + xyz$

1.3 Decision Tree and DNF

Suppose we want to write a decision tree as a DNF, we can do this by taking every path that leads to true value as one term in DNF.

- DT $\Rightarrow$ DNF: slightly less efficient since some nodes in the paths may overlap. Note that each leaf is a node in the tree, so the total number of terms is less or equal to the size of the tree. Also, the size of each term would be less or equal to the size of the tree. Therefore, the size of the DNF would be $O(T^2)$, where T is the size of decision tree.
- DNF $\Rightarrow$ DT: Consider the formula

$$(x_1 \wedge x_2) \vee (x_1 \wedge x_2) \vee ...(x_{n-1} \wedge x_n)$$

  Any decision tree for this DNF would have size larger or equal to $2^{\Omega(n)}$. Decision trees are strictly weaker than DNF since the size after transformation grows at exponential rate.

Generally, we will regard one representation to be "at least as expressive" as another if we can encode the other representation with at most a polynomial increase in representation size.

2 Efficient Computation: algorithms and Circuits

Definition 1 (Efficient algorithm)**.** *We say that an algorithm is efficient if it runs in* poly*-time in its input parameters.*

Why polynomial time? The reasons are:

(1) Mathematical convenience: poly-time is the smallest class of algorithm that (a) contains all linear time algorithm, and (b) closed under the use of subroutines.

(2) Experience in Practice: poly-time algorithms must exploit some kind of "structure" in the problem. Once this structure is identified, algorithms that are efficient in practice usually follow.

(3) Invariance: Every currently viable model of computation (e.g., RAM machine and Turing machine) can be simulated with at most poly-time overhead. But in general, we cannot guarantee less than a poly-time overhead.

3 Computation Problems and NP-completeness

3.1 Computational Problems

Definition 2 (Relation problems)**.** *For sets X (inputs) and Y (outputs), the computational task corresponding to the relation $R \subseteq X \times Y$ is given $x \in X$, find $y \in Y$ such that $(x, y) \in R$.*

Let $X = \{x^{(i)}\}_{i=1}^{m}$, where $x^{(i)}$ consists of lists of labeled examples over n attributes. Let Y encode the set of conjunctions on arbitrary number of attributes. Then, the Elimination Algorithm solves the relation problem.

Definition 3 (Consistent)**.** *we say that a representation C is consistent with a set of labeled examples if $\forall i$, $c(x^{(i)}) = b^{(i)}$.*

$Consis_{conj} = \{([(x^{(1)}, b^{(1)}), \ldots, (x^{(m)}, b^{(m)})], c) : c \text{ is consistent with the } b\text{'s } \}$

3.2 NP-completeness

In this section, we want to show that no poly-time algorithm exists for some problem R. Given no poly-time algorithm exists to solve problem S, if we can show that, given a poly-time algorithm for R, we can construct a poly-time algorithm for S (through reduction), then (by contrapositive) no poly-time algorithm exist for R. Simply put, we have to find a problem S for which we believe no efficient algorithm exists, and show that the problem S can be reduced to the problem R.

Definition 4 (NP-search problem)**.** *A relation problem R is an NP search problem if*

1. *R is "*poly*-balanced": there is a fixed* poly *$p(n)$ such that whenever for a given $x \in X$, there is a y such that $(x, y) \in R$, then there is also some y' such that $|y'| \leq p(|x|)$ and $(x, y') \in R$*
2. *Efficiently verifiable: there is a* poly*-time algorithm for evaluating the Boolean function*

$$R(x, y) = \begin{cases} 1, & \text{if } (x, y) \in R, \\ 0, & \text{otherwise} \end{cases}$$

Definition 5 (NP-complete)**.** *An NP-search problem R is NP-complete if for every (other) NP-search problems, there is an algorithms for S, using an algorithm for R, such that if the algorithm for R runs in* poly*-time, so does the algorithm for S.*

Corollary 6. *A* poly*-time algorithm for any NP-complete problem implies there are poly-time algorithm for every NP-search problem.*

First example of NP-complete problem

Definition 7. *A Boolean circuit is given by a directed acyclic graph, DAG, in which*

1. *The sources are labeled either by attributes $x_1, \ldots, x_n$ or by constraints 0 or 1*
2. *The (K) sinks are distinctly labeled by $y_1, \ldots, y_k$ and have in-degree 1*
3. *The internal nodes are labeled by $\wedge$, $\vee$, or $\neg$, where $\vee$ and $\wedge$ nodes have in-degree 2, and $\neg$ nodes have in-degree 1*

Boolean circuits compute a f^n, $f : \{0,1\}^n \to \{0,1\}^k$. Given $(x_1, \ldots, x_n) \in 0, 1^n$, assign the corresponding Boolean value to nodes in the circuit. Each internal node obtains a value by applying its label f^n to the incoming "edges" vertices. We "read off" the output from $y_1, \ldots, y_k$ (which just obtain the values of the nodes above them).

Definition 8 (CIRCUIT-SAT)**.** *CIRCUIT-SAT* $= \{(c, x), c(x) = 1\}$, *where* c *is the circuit and* x *is the input.*

Theorem 9 (Cook-Levin)**.** *CIRCUIT-SAT is NP-complete*

Proposition 10. *Suppose that there is a* poly*-time algorithm for computing a* $f(f^n : \{0,1\}^n \to \{0,1\}^{k(n)})$. *Then there exists an algorithm that given as input an integer* n*, runs in* poly*-time and outputs a circuit* c_n *such that* $\forall x \in \{0,1\}^n, c_n(x) = f_n(x)$.