Learning from Entailment

B4M36SMU

In this tutorial, we will focus on the setting of learning from entailment, which differs from learning from interpretations introduced in the previous tutorial. In this setting, the aim is to learn a theory Φ such that it entails all positive examples and no negative one, i.e.

$$\Phi \vdash o \; \forall o \in O^+,$$
$$\Phi \nvdash o \; \forall o \in O^-.$$

Clause Reduction

From the previous tutorial we already know θ-subsumption and subsume-equivalence, i.e. $\gamma_1 \approx_\theta \gamma_2$ iff $\gamma_1 \subseteq_\theta \gamma_2 \wedge \gamma_2 \subseteq_\theta \gamma_1$. Similarly to equivalence, we may define a strict θ-subsumption relation which expresses that one clause θ-subsumes a second one, but not the other way around; i.e. we say that γ_1 *strictly θ-subsumes* γ_2 iff $\gamma_1 \subseteq_\theta \gamma_2 \wedge \gamma_2 \not\subseteq_\theta \gamma_1$.

Base on these relations we can define equivalence classes, i.e. all θ-subsume equivalent clauses are in one class for which we select the shortest one to be their representative. This, in turn, allows us to traverse the clauses space more efficiently. We say that a clause γ is reduced if for no γ', $\gamma' \subset \gamma$, $\gamma' \approx_\theta \gamma$. In other words, the reduced clause of a clause γ is in the same subsume-equivalence class as γ, but has the lowest number of literals.

If a clause after deleting one of its literals is in the same θ-subsume equivalence class, then the shorter one can be used instead of the original one. We may compute a reduced clause of a clause by applying this strategy in sequence on every literal of the original clause. See the following pseudocode for this algorithm which expects a clause γ as input and returns reduction of this clause:

$\gamma' \leftarrow \gamma$
forall $l \in \gamma$ **do**
  if $\gamma \subseteq_\theta (\gamma' \setminus \{l\})$ **then**
    $\gamma' \leftarrow \gamma' \setminus \{l\}$
  end
end
return γ'

Exercise

- reduce $p(x, y) \vee p(A, B) \vee m(b) \vee m(y)$

Entailment & θ-subsumption

We say that γ_1 entails γ_2, $\gamma_1 \vdash \gamma_2$[1] iff every model of γ_1 is also a model of γ_2. We may use entailment as a measurement of generality in the space of clauses, i.e. if γ_1 is more general than γ_2 if $\gamma_1 \vdash \gamma_2$. However, to check whether one clause entails another is computationally expensive. Fortunately, the undecidable entailment operator can be approximated by the NP-complete θ-subsumption, i.e. $(\gamma_1 \subseteq_\theta \gamma_2) \implies (\gamma_1 \vdash \gamma_2)$. The opposite does not apply because of *self-resolving* clauses; these are clauses containing the same predicate symbol in both head and body of a clause, i.e. with different negation sings. In FOL setting, having a predicate in a clause with both positive and negative sing does not necessarily result in a tautology. However, when we restrict to non-self-resolving clauses only, the θ-subsumption becomes equal to the entailment; see proof on white board.

napsat dukaz pro tohle na tabuli

[1] Note that in other literature you may find the same functionality symbolized by $\models$ operator.

LGG

A set of all possible atoms, extended by special symbols $\bot$ and $\top$ (false and true), partially ordered by θ-subsumption, forms a complete lattice in which the greatest lower bound (*glb*) and the least general generalization (*lgg*) exist for each two pair of atoms [1]. The $\bot$ symbol is a result of lgg in the case that the input atoms are not compatible, e.g. $lgg(p(x), q(y))$; $\top$ has the same meaning for glb. In fact, we may extend the definition and operate with a lattice over literals, not only atoms, as it can be straightforwardly seen that $lgg(\neg p(x), p(x))$ is equal to $\bot$[2]. So, within this lattice, glb corresponds to unification of two literals and lgg corresponds to the exactly opposite operation, therefore it is sometimes called *anti-unification*.

So, to formally define lgg of two clauses:

$$\begin{aligned} LGG(\gamma_1, \gamma_2) &= \gamma \\ s.t.\quad & \gamma \subseteq_\theta \gamma_1 \\ & \gamma \subseteq_\theta \gamma_2 \\ & \forall \gamma' s.t. \gamma' \subseteq_\theta \gamma_1 \wedge \gamma' \subseteq_\theta \gamma_2 : \gamma' \subseteq_\theta \gamma \end{aligned}$$

The only missing part is the declarative description of the computation algorithm of an lgg of two clauses:

$$\begin{aligned} LGG(\gamma_1, \gamma_2) &= \bigvee_{\substack{l_1 \in \gamma_1, l_2 \in \gamma_2 \\ samePredNeg(l_1, l_2)}} LGG(l_1, l_2) \\ LGG(p(t_1, t_2, \dots), p(t'_1, t'_2, \dots)) &= p(LGG(t_1, t'_1), LGG(t_2, t'_2), \dots) \\ LGG(f(t_1, \dots), f'(t'_1, \dots)) &= \begin{cases} x' & f \neq f' \\ f(LGG(t_1, t'_1), \dots) & otherwise \end{cases} \\ LGG(f(\dots), x) &= x' \\ LGG(x, y) &= x' \end{aligned}$$

where p stands for a predicate symbol, f stands for a function (or a constant) symbol, and x' stands for a variable. However, it is important to note that each pair of terms (t_1, t_2) is anti-unified with exactly one variable, where t_1 is a term from l_1 and t_2 is a term from l_2. The $samePredNeg$ is a function which returns true iff both of its arguments have the same negation sign and the same predicate; i.e. it is just different notation for the *selection* process as stated in the lectures. Recall from [3] that lgg is a symmetric, commutative, and associative operator.

Exercise

- compute unification and anti-unification of $\gamma_1 = p(A, x, f(x))$ and $\gamma_2 = p(y, f(B), f(f(B)))$, [2]
- compute lgg of $\gamma_1 = e(x, z) \vee \neg e(z, y) \vee \neg e(y, z)$ and $\gamma_2 = \neg e(x, y) \vee \neg e(z, y) \vee m(z)$
- reduce the clause obtained earlier

LGG with Taxonomical Information

Consider the following case when one is asked to compute lgg of

$\gamma_1 = animal(dog)$

$\gamma_2 = animal(mammal)$

The standard lgg of these would be $animal(X)$. However, in some specific cases, a user may know some taxonomical information about the constants of the domain, e.g. the relationship that a dog is a mammal. It is noticeable that lgg enhanced with this taxonomical information would produce $animal(mammal)$ as it is the least common ancestor in the extended lattice.

[2] Lgg can be defined in multiple ways. In some of them, lgg for these two literals is undefined.

Exercise

- Given the taxonomy
  - $isa(direwolf, dog)$
  - $isa(dog, mammal)$
  - $isa(cow, mammal)$

  and clauses
  - $\gamma_1 = \neg spieces(tim, dog) \vee voice(tim, bark)$
  - $\gamma_2 = \neg spieces(nymeria, direwolf) \vee voice(nymeria, bark)$
  - $\gamma_3 = \neg spieces(milka, cow) \vee voice(milka, bu)$

  compute $lgg(lgg(\gamma_1, \gamma_2), \gamma_3)$.

References

[1] Luc De Raedt. *Logical and relational learning*. Springer Science & Business Media, 2008.

[2] Shan-Hwei Nienhuys-Cheng and Ronald De Wolf. *Foundations of inductive logic programming*. Vol. 1228. Springer Science & Business Media, 1997.

[3] Filip Železný and Jiří Kléma. *SMU textbook*. 2017.