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 that entails all positive examples and no negative ones. Whereas examples were interpretations in *learning from interpretations*, they are clauses (!) in *learning from entailment*[1]. That is the *learning from entailment* problem is:

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

Clause Reduction

From the previous tutorial we already know what *θ-subsumption* is and what it means when we say that two clauses are *subsume-equivalent*.

Reminder (Subsumption and subsume-equivalence): We say that a clause C θ-subsumes a clause D (denoted $C \subseteq_\theta D$) if there exists a substitution θ such that $C\theta \subseteq D$ (here we treat both clauses as sets of literals and "$\subseteq$" denotes the standard "subset" relation). We say that C and D are subsume-equivalent (denoted $C \approx_\theta D$) if $C \subseteq_\theta D \wedge D \subseteq_\theta C$.

We also define *strict θ-subsumption* which expresses that one given clause θ-subsumes another given clause, but not the other way around; i.e. we say that C *strictly θ-subsumes* D, $C \subset_\theta D$, if $C \subseteq_\theta D \wedge D \not\subseteq_\theta C$.

Using these relations we can define equivalence classes of clauses: all subsume equivalent clauses are in one class for which we select a shortest one to be their representative (all of these shortest clauses are the same up to renaming of variables – i.e. up to *isomorphism*). This, in turn, allows us to traverse the space of clauses more efficiently. We say that a clause C is reduced if there is no C', $C' \subsetneq C$ such that $C' \approx_\theta C$. In other words, the reduced clause of a clause C is in the same subsume-equivalence class as C, but has the smallest number of literals.

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

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

Exercise

- Find a reduction of the clause: $\forall x, y, b \colon p(x, y) \vee p(A, B) \vee m(b) \vee m(y)$
- Find a reduction of the clause: $\forall x, y, b \colon p(x, y) \vee p(A, b) \vee m(b) \vee m(y)$

[1] Note that we use $\vdash$ for entailment relation, i.e. $C_1 \vdash C_2$ means that C_1 entails C_2. However, in literature you may find another symbol for expressing the same functionality, e.g. $\models$.

Entailment & θ-subsumption

We say that C_1 entails C_2, (denoted $C_1 \vdash C_2$) if every model of C_1 is also a model of C_2. We may use entailment to partially order clauses by generality: we say that C_1 is more general than C_2 if $C_1 \vdash C_2$. However, checking entailment is generally not possible because it is an undecidable problem. Fortunately, the entailment operator can be approximated θ-subsumption, which is "only" NP-complete ("only" is used here in comparison to "undecidable"). In particular, we have $(C_1 \subseteq_\theta C_2) \implies (C_1 \vdash C_2)$. The converse does not hold in general (proof shown on the white board during the tutorial). However, it does hold for non-self resolving clauses.

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 lgg of two incompatible atoms, that is atoms based on different predicates, is $\bot$, e.g. $lgg(p(x), q(y)) = \bot$. $\top$ 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*.

Next we extend the definition of lgg to clauses. A clause C is an lgg of clauses C_1 and C_2 if it satisfies the following three conditions:

1. $C \subseteq_\theta C_1$,
2. $C \subseteq_\theta C_2$,
3. $\forall C'$ s.t. $C' \subseteq_\theta C_1 \wedge C' \subseteq_\theta C_2 : C' \subseteq_\theta C$.

Note that lgg of two clauses is not unique.[3] Importantly, though, any two lggs of the same two clauses are guaranteed to be subsume-equivalent. Therefore, in what follows, we will sometimes "pretend" that there is just one lgg and we will, for instance, write $C = LGG(C_1, C_2)$ even though, strictly speaking, $LGG(C_1, C_2)$ is not unique.

Constructing lgg of two terms: The basic step to compute lgg of two clauses is to compute firstly lgg of two terms t_1 and t_2. The simplest rule comes with resolving two constants: if these two constants are equal, then return the constant. The second simplest rule comes with resolving two variables or one constant and a variable, e.g. $t_1 = Adam$ and $t_2 = x$. In this situation, lgg returns a new variable for the tuple $(Adam, x)$. The third rule to compute lgg of two terms is applicable when at least one of them is a compound term. If there is only one compound term, e.g. $t_1 = fatherOf(Adam)$ and $t_2 = x$, or function symbols of the terms differ, e.g. $t_1 = fatherOf(Adam)$ and $t_2 = motherOf(Adam)$, lgg returns a new variable for this tuple (similarly to the basic rule). Finally, if both terms have the same function symbol, e.g. $t_1 = fatherOf(Adam)$ and $t_2 = fatherOf(x)$; then lgg returns a compound term with the same function symbol applied to lgg-ed arguments of these compound terms, e.g. $fatherOf(LGG(Adam, x))$ which equals to $father(x_{Adam,x})$. 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. We can sum up these rules in the following pseudocode[4]:

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

[3] Strictly speaking, lgg of atoms is not unique either but lggs of atoms differ only in renaming of variables. For instance, both $p(x)$ and $p(y)$ are lggs of the pair of atoms $p(Alice)$ and $p(Bob)$.

[4] Recall that lgg is a symmetric operator, thus rules behave similarly for symmetric inputs.

$$LGG(A, B) = \begin{cases} A & A = B \\ x_{A,B} & otherwise \end{cases}$$
$$LGG(x, y) = x_{x,y}$$
$$LGG(f(\dots), x) = x'$$
$$LGG(f(t_1, \dots), f'(t'_1, \dots)) = \begin{cases} x' & f \neq f' \\ f(LGG(t_1, t'_1), \dots) & otherwise \end{cases}$$

Where A is a constant, B is an arbitrary term, x and y are variables, and f/n is a function symbol of arity n. New variables are denoted either by tuples they represent, e.g. $x_{A,B}$, or by prime symbol, e.g. x'.

Constructing lgg of two clauses: Having done the basic elements for computing lgg of terms, we can construct lgg of two clauses C_1 and C_2. Basically, we compute lgg of all possible pairs of literals (l_1, l_2), where l_1 occurs in the C_1 and l_2 in C_2, for which it holds that they have the same predicate symbol and negation sign. Lgg of two literals is a predicate with the same predicate symbol and negation sign applied to lgg-ed arguments of the literals. See following pseudocode:

$$LGG(C_1, C_2) = \bigvee_{\substack{l_1 \in C_1, l_2 \in C_2 \\ samePredNeg(l_1, l_2)}} LGG(l_1, l_2)$$
$$LGG(p(t_1, t_2, \dots), p(t'_1, t'_2, \dots)) = p(LGG(t_1, t'_1), LGG(t_2, t'_2), \dots)$$

Where p is a predicate symbol, l is a literal and C is a clause. 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. Once again, recall from [3] that lgg is a symmetric, commutative, and associative operator.

Exercise

- compute unification and anti-unification (lgg) of $C_1 = p(A, x, f(x))$ and $C_2 = p(y, f(B), f(f(B)))$, [2]
- compute lgg of $C_1 = e(x, z) \vee \neg e(z, y) \vee \neg e(y, z)$ and $C_2 = \neg e(x, y) \vee \neg e(z, y) \vee m(z)$
- reduce the clause obtained above

LGG with Taxonomical Information

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

$C_1 = animal(dog)$

$C_2 = animal(mammal)$

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.

Exercise

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

- $isa(dog, mammal)$
- $isa(cow, mammal)$

and clauses

- $C_1 = \neg species(tim, dog) \vee voice(tim, bark)$
- $C_2 = \neg species(nymeria, direwolf) \vee voice(nymeria, bark)$
- $C_3 = \neg species(milka, cow) \vee voice(milka, bu)$

compute $lgg(lgg(C_1, C_2), C_3)$.

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.