SMU ILP: Learning from Entailment

In this tutorial, we will focus on a different setting of ILP, the setting of learning from entailment. In this setting, the aim is to learn a theory T such that the theory entails all positive examples and no negative ones, i.e.

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

For this tutorial and the homework assignment, we will assume that the input data do not contain noise.

Logic

This section contains definitions of FOL terms which are necessary for this tutorial. It is expected that the reader is familiar with the similar section from the previous tutorial. Note that for simplicity of notation, this tutorial considers constant symbols to be function symbols of arity 0.

The core term of this tutorial is θ-subsumption. We say that γ_1 *θ-subsumes* γ_2, $\gamma_1 \subseteq_\theta \gamma_2$, iff there exists a substitution θ such that $\gamma_1\theta \subseteq \gamma_2$ in the sense of literals. For example, for $\gamma_1 = p(X, Y)$ and $\gamma_2 = p(a, Z)$ there holds that $\gamma_1 \subseteq_\theta \gamma_2$. To check this by the definition, firstly find a substitution, e.g. $\theta = \{X \mapsto a, Y \mapsto Z\}$, and check the sets of literals, e.g. $\gamma_1\theta = \{p(a, Z)\} \subseteq \{p(a, Z)\} = \gamma_2$. Note here that the $=$ relation is a little bit overloaded, meaning that a clause is the same as set of literals it contains. On the contrary, $\gamma_2 \not\subseteq_\theta \gamma_1$. But in general, it may happen that two clauses are *subsume-equivalent*, i.e. $\gamma_i \approx_\theta \gamma_j$ iff $\gamma_i \subseteq_\theta \gamma_j$ and $\gamma_j \subseteq_\theta \gamma_i$. Similarly to $\subseteq_\theta$ relation, we may define relation $\subset_\theta$ which expresses that a clause *strictly θ-subsumes* another clause.

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

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

FOL setting, having and predicate in a clause with both positive and negative sing does not necessary result in a tautology. In the case when we restrict to non-self-resolving clauses only, the θ-subsumption becames equal to the entailment.

Clause Reduction

There may exist two or more clauses which are subsume-equivalent, e.g. $\gamma_1 \approx_\theta \gamma_2 \approx_\theta \gamma_3 \dots$. These clauses are tight up together by one thing – they have the same semantic meaning. For the computation complexity of clauses space traversal, it would be great to visit only one representative (canonical-like) instead of all possible clauses of one $\approx_\theta$-class. For this task, we may traverse only the space of *reduced clauses*. We say that a clause γ is reduced iff there is no γ' such that $\gamma' \approx_\theta \gamma$ and $|\gamma'| < |\gamma|$. 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 shorther 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'

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. We may in fact 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 referred as *anti-unification*.

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

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.

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. To formally incorporate this extension to the lgg computation scheme, add two more rules:

$$LGG(f(t_1,\dots),f'(t'_1,\dots)) = \begin{cases} f' & isa(f,f') \\ f & isa(f',f) \\ X & f \neq f' \\ f(LGG(t_1,t'_1),\dots) & otherwise \end{cases}$$

Exercise

- list all pairs s.t. $\gamma_1 \subseteq_\theta \gamma_2$ from the following clauses
  (a) $p(X)$
  (b) $q(Y) \vee p(Y)$
  (c) $p(a) \vee p(b) \vee p(Z)$
  (d) $q(a)$
  (e) $q(X) \vee p(Z) \vee q(Z)$
- 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 $\gamma_1 = e(X,Z) \vee \neg e(X,Y) \vee \neg e(Y,Z)$ and $\gamma_2 = \neg e(X,Y) \vee \neg e(Z,Y) \vee m(Z)$
- reduce the clause obtained earlier
- reduce $p(X,Y) \vee p(a,b) \vee m(B) \vee m(Y)$

Homework Assignment

Download and unzip the archive for this tutorial/assignment. If you haven't installed the necessary SW, see *readme* for instructions. If you haven't gone trough the notebook from the previous tutorial, *minlog*, see that for the basic functionality of the library. In any case, see notebook *minlogHW* for getting into touch with the new functionalities of the library needed for this task.

Implement class *LGGResolver* in file *lggAgent.py*, which represents an online learning agent in the setting of learning from entailment. Recall that this setting differs a little bit from the lecture; an observation with class (*pos*/*neg*) is given instead of using a time delayed reward. Also recall that for this task, we assume that the data do not contain noise. The agent may be instantiated with variable *taxonomical* set to a set of literals corresponding to a taxonomy of the form $isa(car, vehicle)$. It is always ensured that literals in the set describe a forest, i.e. set of rooted oriented trees. In the case that *taxonomical* is not *None*, agent run lgg enhanced by the given taxonomical information as described earlier in this document. By default, no taxonomy is provided, e.g. *taxonomical=None*.

Implement method *seeObservation(sample,reduceClause)*. The first parameter is just a sample, i.e. an observation the agent gets from the environment in the form of a clause with a class (*pos*/*neg*). The agent updates his hypothesis by

computing lgg upon his current hypothesis and the sample given iff the sample's class is positive but the sample is not θ-subsumed by the agent's hypothesis. The second parameter is simply a boolean stating whether the agent stores only non-reduced clauses in his mind.

Implement your solution into the file *lggAgent.py*; do not change other files. Use file *hw.py* to run agent. In case of any issue with the provided library, especially an issue concerning installation, post your issue to the SMU forum, so that other students with the same issue can find issue-related solutions easily.

Create two pairs of clauses and compute their lgg by hand; see the following section for more details.

Grading

You can obtain up to 12 points from this assignment; see the following list with possible gains:

- 5 points: implement the on-line lgg-learning agent without the reduction step (called by *seeObservation(sample)*)
- 3 points: implement the reduction step (called by *seeObservation(sample,reduceClause=True)*)
- 2 points: implement the lgg enhanced by taxonomical information
- 1 points*: create two clauses and compute their lgg (without the reduction step) such that the final clause is a reduced one
- 1 points*: create two clauses and compute their lgg (without the reduction step) such that the final clause is not a reduced one

However, you need to obtain at least 6 points in order to get the solution accepted. Considering the manual computation, i.e. *, cases:

- an assignment of the anti-unification process (pair of terms to a variable) must be present
- plain input-output of your implementation, e.g. with different variable names, etc., will not be accepted
- do not use clauses from the lectures, tutorials, and trivial ones
- select reasonable short clauses

Format of the submitted solution is up to you (PDF, txt). However, in the case that your logic notation differs from the tutorial's one, you must define notation used in the solution.

Submit your solution containing

- source codes, i.e. *lggAgent.py*

- a PDF or a plain text file containing clauses and their lggs (the last two possibilites in table of points)

by Monday, May 14 to brute system.
Cooperation with other colleagues is strictly prohibited.

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.