Learning from Entailment

B4M36SMU

In this tutorial, we will focus on the setting of learning from entailment at the relational level and describe the third assignment from this course.

In the setting learning from entailment, the aim is to learn a theory Φ that entails all positive examples and no negative ones. Examples are in the form of clauses (!). That is the *learning from entailment* problem is:

$$\forall o \in O^+ : \Phi \models o,$$
$$\forall o \in O^- : \Phi \not\models o.$$

Entailment & θ-subsumption

We say that γ_1 entails γ_2, (denoted $\gamma_1 \models \gamma_2$) if every model of γ_1 is also a model of γ_2. We may use entailment to partially order clauses by generality: we say that γ_1 is more general than γ_2 if $\gamma_1 \models \gamma_2$. 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 $(\gamma_1 \subseteq_\theta \gamma_2) \implies (\gamma_1 \models \gamma_2)$. However, it does hold for non-self resolving clauses.

In the previous paragraph, γ_i was assumed to be a universally quantified disjunction of literals as it is usual in the ILP literature. A similar relation holds for existentially quantified conjunctions[1], i.e. $(\gamma_1 \subseteq_\theta \gamma_2) \implies (\gamma_2 \models \gamma_1)$ where γ_1 and γ_2 are existentially quantified conjunctions.

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 γ is an lgg of clauses γ_1 and γ_2 if it satisfies the following three conditions:

1. $\gamma \subseteq_\theta \gamma_1$,
2. $\gamma \subseteq_\theta \gamma_2$,
3. $\forall \gamma'$ s.t. $\gamma' \subseteq_\theta \gamma_1 \wedge \gamma' \subseteq_\theta \gamma_2 : \gamma' \subseteq_\theta \gamma$.

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 $\gamma = LGG(\gamma_1, \gamma_2)$ even though, strictly speaking, $LGG(\gamma_1, \gamma_2)$ is not unique.

[1] Recall De Morgan's laws.

[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 $p(Bob)$.

Constructing lgg of two terms: [4] 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.
- In any other case (e.g. $lgg(A, B)$, $lgg(x, A)$, $lgg(x, y)$), return a new variable the tuple of terms given, ($x_{A,B}$, $x_{x,A}$, and $x_{x,y}$ respectively).
- 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[5]:

$$LGG(A, t) = \begin{cases} A & A = t \\ x_{A,t} & otherwise \end{cases}$$
$$LGG(x, y) = x_{x,y}$$

Where A is a constant, t is an arbitrary term, x and y are variables. New variables are denoted either by tuples they represent, e.g. $x_{A,t}$.

Constructing lgg of two clauses: Having done the basic elements for computing lgg of terms, we can construct lgg of two clauses γ_1 and γ_2. Basically, we compute lgg of all possible pairs of literals (l_1, l_2), where l_1 occurs in the γ_1 and l_2 in γ_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 the following pseudocode:

$$LGG(\gamma_1, \gamma_2) = \bigvee_{\substack{l_1 \in \gamma_1, l_2 \in \gamma_2 \\ compatibleLiterals(l_1, l_2)}} LGG(l_1, l_2)$$
$$LGG(p(t_1, t_2, \dots), p(t'_1, t'_2, \dots)) = p(LGG(t_1, t'_1), LGG(t_2, t'_2), \dots)$$

Where p is a predicate symbol, l is a literal and both γ_1 and γ_2 are clauses. The *compatibleLiterals* is a function which returns true iff both of its arguments have the same negation sign and the same predicate. Once again, recall from the lecture that lgg is a symmetric, commutative, and associative operator. Last but not least, the output of the lgg is to be reduced using the clause reduction algorithm from the previous tutorial.

This procedure allows us to traverse the lattice induced by θ-subsumption and clause reduction to find the least general generalization of two relational clauses. Intuitively speaking, each node of the lattice represents one subsume-equivalence class.

Exercise

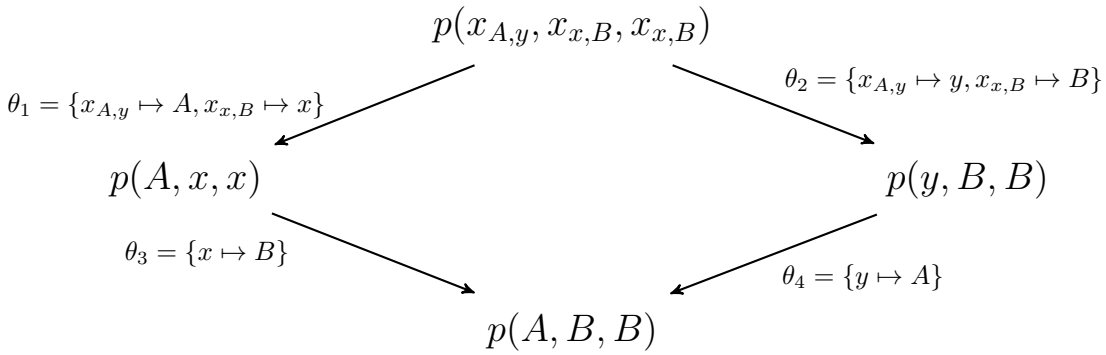
- Write down the definition of lgg.
- Given two clauses γ_1 and γ_2, what is the size of $lgg(\gamma_1, \gamma_2)$?
- Compute lgg of $\gamma_1 = p(x, y)$ and $\gamma_2 = p(a, a)$.
- Compute lgg of $\gamma_1 = p(x, A)$ and $\gamma_2 = p(A, x)$.
- Does this equation $lgg(p(x, x, A), p(B, B, A)) = p(x', y', z')$ hold? If not, why?
- Compute unification and anti-unification (lgg) of $\gamma_1 = p(A, x, x)$ and $\gamma_2 = p(y, B, B)$, [2]. Write down γ_1, γ_2 and the results obtained to visualize the lattice induced by θ-subsumption.
- 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 above.

[4] In this tutorial, we will inspect only relational (function-free) lgg. First-order setting (with function symbols) will come in an upcoming tutorial.

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

Solution:

- See the **LGG** section. There are three constraints, do not miss one.
- Depending on given clauses. The lower bound would be zero – in the case that there is no pair of compatible literals in γ_1 and γ_2. The worst-case upper bound would be $|\gamma_1| \cdot |\gamma_2|$.
- $lgg(\gamma_1, \gamma_2) = p(x', y')$
- $lgg(\gamma_1, \gamma_2) = p(x_{x,A}, x_{A,x})$
- No. Although the resulting clause θ-subsumes the input ones, it is not the *least* – the correct result is $p(x', x', A)$.
- $lgg(\gamma_1, \gamma_2) = p(x', y', y')$, unification of γ_1 and γ_2 is $p(A, B, B)$.



Lgg of γ_1 and γ_2 is at the top of the lattice. Corresponding substitutions lie on edges to show how a clause higher in the lattice can θ-subsume a lower clause. At the bottom, the unification of γ_1 and γ_2 lies; see, the most general unifier (mgu) is $\theta_{mgu} = \{x \mapsto B, y \mapsto A\}$.

new variable	γ_1 (θ_1)	γ_2 (θ_2)
$x_{A,y}$	A	y
$x_{x,B}$	x	B

- $lgg(\gamma_1, \gamma_2) = \neg e(x_{y,x}, x_{z,y}) \vee \neg e(x_{y,z}, x_{z,y}) \vee \neg e(x_{z,x}, x_{y,y}) \vee \neg e(x_{z,z}, x_{y,y})$ (non-reduced result)
- $\neg e(x_{y,x}, x_{z,y})$

Relative Least General Generalization

In some cases, we may possess some additional information about the world, let say a background theory $\mathcal{B}$. Then, the learning task changes to searching a hypothesis Φ which entails just positive observations with the background theory, i.e.

$$\forall o \in O^+ \colon \mathcal{B} \cup \{\Phi\} \models o,$$
$$\forall o \in O^- \colon \mathcal{B} \cup \{\Phi\} \not\models o.$$

We will restrict ourselves to a background theory $\mathcal{B}$ such that it is a finite set of function-free ground facts, e.g. $\mathcal{B} = \{human(Adam), human(Eva)\}$. In general, the theory can contain non-ground clauses, but this out of the scope of this tutorial.

Before jumping to the lgg enriched by a background theory, we have to define a few things, which are mostly analogical to the previous part. The core idea here is to use θ-subsumption with respect to a background theory $\mathcal{B}$, that is, $\gamma_1 \subseteq^{\mathcal{B}}_{\theta} \gamma_2$ if there exists a substitution θ such that $\gamma_1\theta \subseteq \gamma_2 \vee \neg\mathcal{B}$. While having defined $\subseteq^{\mathcal{B}}_{\theta}$, the definitions of θ-subsume equivalence with respect to $\mathcal{B}$, $\approx^{\mathcal{B}}_{\theta}$, strict θ-subsumption relative to $\mathcal{B}$, etc., are analogical to θ-subsumption ones.

Now, we will define the *relative least general generalization* with respect to a background theory $\mathcal{B}$ of two clauses γ_1 and γ_2, which is given by

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

Note that the only difference with lgg is replacing $\subseteq_\theta$ by $\subseteq_\theta^{\mathcal{B}}$. The last step here is to formally define the computation of rlgg, which can be processed as follows:

$$RLGG_{\mathcal{B}}(\gamma_1, \gamma_2) = LGG(\gamma_1 \vee_{l \in \mathcal{B}} \neg l, \gamma_2 \vee_{l \in \mathcal{B}} \neg l).$$

Consider $\gamma_1 = p(x)$, $\gamma_2 = q(x)$ and $\mathcal{B} = \{r(A, B), r(A, A)\}$, rlgg of these two is:

$$\neg r(A, B) \vee \neg r(A, A) \vee \neg r(A, x_{A,B}) \vee \neg r(A, x_{B,A})$$

One can easily see that γ_1 and γ_2 do not have any lgg, yet rlgg products a non-empty disjunction of literals. However, this follows from the fact that the clause above is not reduced. So, we have to reduce the clause relative to $\mathcal{B}$. Fortunately, it is as easy as using the clause reduction algorithm from the previous tutorial with a slight modification: use $\subseteq_\theta^{\mathcal{B}}$ instead of $\subseteq_\theta$.

Let us see the following example of clause reduction relative to background theory $\mathcal{B}$ which is similar to the previous:

- $\gamma_1 = p(x)$
- $\gamma_2 = q(x)$
- $\mathcal{B} = \{r(A), r(B)\}$

$$
\begin{aligned}
RLGG_{\mathcal{B}}(\gamma_1, \gamma_2) &= LGG(p(x) \vee \neg r(A) \vee \neg r(B), q(x) \vee \neg r(A) \vee \neg r(B)) = \\
&= \neg r(x_{A,B}) \vee \neg r(x_{B,A}) \vee \neg r(A) \vee \neg r(B)
\end{aligned}
$$

The reduction procedure relative to $\mathcal{B}$ of the clause obtained above follows:

1. Notation: removed literal (crossed out ~~text~~), literals after $\mathcal{B}$ unfolding (underline).
2. $\gamma' = \neg r(x_{A,B}) \vee \neg r(x_{B,A}) \vee \neg r(A) \vee \neg r(B)$
3. Removal of $\neg r(x_{A,B})$: $\neg r(x_{A,B}) \vee \neg r(x_{B,A}) \vee \neg r(A) \vee \neg r(B) \subseteq_\theta \neg r(x_{B,A}) \vee \neg r(A) \vee \neg r(B) \vee \neg \mathcal{B}$, e.g. $\theta_1 = \{x_{A,B} \mapsto A\}$
   $\gamma' = \neg r(x_{B,A}) \vee \neg r(A) \vee \neg r(B)$
4. Removal of $\neg r(x_{B,A})$: $\neg r(x_{A,B}) \vee \neg r(x_{B,A}) \vee \neg r(A) \vee \neg r(B) \subseteq_\theta \neg r(A) \vee \neg r(B) \vee \neg \mathcal{B}$, e.g. $\theta_2 = \{x_{A,B} \mapsto A, x_{B,A} \mapsto A\}$
   $\gamma' = \neg r(A) \vee \neg r(B)$
5. Removal of $\neg r(A)$: $\neg r(x_{A,B}) \vee \neg r(x_{B,A}) \vee \neg r(A) \vee \neg r(B) \subseteq_\theta$ ~~$\neg r(A) \vee$~~ $\neg r(B) \vee$ $\underline{\neg r(A) \vee \neg r(B)}$, use θ_2
   $\gamma' = \neg r(B)$
6. Removal of $\neg r(B)$: $\neg r(x_{A,B}) \vee \neg r(x_{B,A}) \vee \neg r(A) \vee \neg r(B) \subseteq_\theta$ ~~$\neg r(B) \vee$~~ $\underline{\neg r(A) \vee \neg r(B)}$, use θ_2
   $\gamma' = \{\}$
7. See, the reduction of $\neg r(x_{A,B}) \vee \neg r(x_{B,A}) \vee \neg r(A) \vee \neg r(B)$ relative to $\mathcal{B}$ is $\{\}$.

Exercise

- Compute rlgg of γ_1 and γ_2 relative to $\mathcal{B}$.
  - $\gamma_1 = moveable(A)$
  - $\gamma_2 = moveable(B)$
  - $\mathcal{B} = \{on(A, C), on(C, D), on(B, D), shape(A, Box), shape(B, Box), shape(C, Box), shape(D, Floor)\}$
- Reduce the clause obtained above relative to $\mathcal{B}$.

Solution:

- $rlgg_{\mathcal{B}}(\gamma_1, \gamma_2) = lgg(\gamma_1 \vee \neg\mathcal{B}, \gamma_2 \vee \neg\mathcal{B}) =$
  $= lgg(moveable(A) \vee \neg on(A,C) \vee \neg on(C,D) \vee \neg on(B,D) \vee \neg shape(A,Box) \vee \neg shape(B,Box) \vee$
  $\vee \neg shape(C,Box) \vee \neg shape(D,Floor),$
  $moveable(B) \vee \neg on(A,C) \vee \neg on(C,D) \vee \neg on(B,D) \vee \neg shape(A,Box) \vee \neg shape(B,Box) \vee$
  $\vee \neg shape(C,Box) \vee \neg shape(D,Floor)) =$
  $= \neg on(A,C) \vee \neg on(B,D) \vee \neg on(C,D) \vee \neg on(x_{A,B}, x_{C,D}) \vee \neg on(x_{A,C}, x_{C,D}) \vee \neg on(x_{B,A}, x_{D,C}) \vee$
  $\neg on(x_{B,C}, D) \vee \neg on(x_{C,A}, x_{D,C}) \vee \neg on(x_{C,B}, D) \vee \neg shape(A,Box) \vee \neg shape(B,Box) \vee \neg shape(C,Box) \vee$
  $\neg shape(D,Floor) \vee \neg shape(x_{A,B}, Box) \vee \neg shape(x_{A,C}, Box) \vee \neg shape(x_{A,D}, x_{Box,Floor}) \vee$
  $\neg shape(x_{B,A}, Box) \vee \neg shape(x_{B,C}, Box) \vee \neg shape(x_{B,D}, x_{Box,Floor}) \vee \neg shape(x_{C,A}, Box) \vee$
  $\neg shape(x_{C,B}, Box) \vee \neg shape(x_{C,D}, x_{Box,Floor}) \vee \neg shape(x_{D,A}, x_{Floor,Box}) \vee \neg shape(x_{D,B}, x_{Floor,Box}) \vee$
  $\neg shape(x_{D,C}, x_{Floor,Box}) \vee moveable(x_{A,B})$
- After reduction relative to $\mathcal{B}$, we get $moveable(x_{A,B})$. The reduction relative to $\mathcal{B}$ removes all negated facts from $\mathcal{B}$. We can actually remove these, i.e. to do $rlgg_{\mathcal{B}}(\gamma_1, \gamma_2) \setminus \neg\mathcal{B}$, to easy the computation. However, note, that it is necessary to proceed with reduction relative to $\mathcal{B}$ thereafter; otherwise, we could end up with an incorrect result.

Homework Assignment

You are going to learn the rules of a card game from observing plays. You know the following: The game is played with a fragment of cards; each card has a value and shape. The only known restriction is for the values of cards, e.g. from two to ace; the number of shapes may vary from game to game. There are two players and both draw the same number of cards. Which player won is determined only by the cards they drew. The rules may be unfairly biased in favor of one of the players. You repeatedly observe what cards are dealt and which player won. From this, you are to learn to predict which player will win depending on which cards both players dealt.

Implementation Details

Download codes from faculty gitlab https://gitlab.fel.cvut.cz/svatoma1/smu-ilp-assignment .[6] Your task is to implement the behavior of the agent in *student/agent.py* by implementing methods *receiveSample*, *receiveReward* and *getHypothesis*. These methods work in similar ways as in the previous homework.

- ***recieveSample***: Firstly, the environment gives the agent an observed play by calling *recieveSample* and the agent returns 0 or 1 if the first player wins or the second one.
- ***recieveReward***: After that, the environment returns reward to the agent by calling *recieveReward* with 0 or -1 indicating that the agent's prediction was correct or not.
- ***getHypothesis***: Upon a request of the environment, by calling *getHypothesis*, the agent returns a string representation of its current hypothesis.

There are two more important classes, both of them can be found in *src/game.py*.

- The *Rule* class is an "interface" with one method deciding the winner given both player's hands. You may find two implementations of this "interface", *Sum* and *Flush*.
- The *Oracle* class serves as a card generator.

The cards are in the form of $VALUE\_OF\_SHAPE$, e.g. $KING\_OF\_HEARTS$. A **play** is then a tuple of hands, i.e. a list of cards a player holds; for example

$$([ACE\_OF\_HEARTS, ACE\_OF\_LEAVES], [OBER\_OF\_BELLS, UNTER\_OF\_LEAVES])$$

is a possible observed play in which the first player has got two aces while the second one has not got any ace. It is ensured that the values of cards are in the scope of two, three, four,..., ten, jack, queen, king, ace with respective values (2, 3, 4,..., 10, 11, 12, 13, 14).

[6] The structure of the classes is the same as the composition of first-order logic syntax described in the previous ILP tutorial. The framework is typed, so it is very beneficial to use some clever IDE for Python, which can help you greatly.

Environment set-up: Besides the fact that a player's hand is represented as a list of raw cards, the agent has to deal with different parameterizations (*oracles*) of the game and the target concept, which may vary from game to game. For example, in one run of the players' battle, each one picks only two cards, while in another battle they pick four. Similarly, the target concept may vary as well (*Rule*). For example, one time the player with a bigger total sum of cards values he holds wins (the *Sum* rule), while in another setting a specific combination of cards determines the winner, e.g. some kind of straight[7]. Moreover, the target concept may be a conjunction of these, e.g. the first player wins only if it has a flush and the total sum of his cards is bigger than the opponent's one. For each battle, i.e. a sequence of observed plays, it holds that the target concept and the battle settings do not change during single plays (see *Oracle* in an action).

The agent knows that order of the cards on a player's hand does not matter.

Scoring of the assignment, from which you can obtain 15 points[8], is done in the following way:

- Correct implementation of the main loop concerning on-line learning of FOL-based (generalizing) agent is worth 5 points.
- You may create up to two new concepts (beside *Sum* and *Flush*) by a description in a PDF report altogether with an implementation of the *Rule* "interface" in *student/agent.py*. For each new concept, you will obtain 1 point.
- Design and implement a suitable FOL representation to learn the concepts. You will obtain 1 point for each concept (Sum, Flush, your two concepts) your agent is able to learn using the FOL representation.
- Write a brief PDF report describing behavior of your agent[9], new concepts, and your representation, discuss the pros and cons of the representation, altogether with possible bottlenecks. Run and describe a few experiments, i.e. games with different *Oracle* settings; report experiments' settings altogether with observations. You can obtain up to 4 points for the report.

Do not overfit your concepts to the game of poker, i.e. the concept of full house does not make any sense if each player has only two cards in his hand. Try to come up with interesting concepts, e.g. a straight, and avoid using *attribute-value* approach by generating a single feature-predicate by python implementation; thereafter, there would be not much first-order / relational learning. If, however, you feel there is no other way to represent the concept, you should defend the design in the report.

This homework assignment has to be elaborated independently with no collaboration at all[10].

Submit an archive containing:

- *student/agent.py* file containing your implementation of the agent and new concepts
- a PDF report

The file *run/main.py* will need some changes in order to do experiments, but together with *student/agent.py*, these are the only two files you should modify. If you need a change in the rest of the codes, say so using course forum, e-mail, ,or a git issue.

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.

[7] *Straight* is a sequence of increasing values.

[8] In case you find a bug in the provided code, you may get one extra point, but the maximum of points from the assignment is still 15 points. Please report these bugs at the course forum, so that only one point is distributed among all students for each found bug.

[9] What does his hypothesis mean?

[10] Think for your future position in a government.