Lecture Notes in Machine Learning – Chapter 8: Relational Learning and Inductive Logic Programming

Zdravko Markov

March 14, 2004

1 Language of logic programming

1.1 Syntax

Fisrtly, we shall define briefly the language of First-Order Logic (FOL) (or Predicate calculus). The alphabet of this language consists of the following types of symbols: *variables, constants, functions, predicates, logical connectives, quantifiers and punctuation symbols.* Let us denote variables with alphanumerical strings beginning with capitals, constants – with alphanumerical strings beginning with lower case letter (or just numbers). The functions are usually denotes as f, g and h (also indexed), and the predicates – as p, q, r or just simple words as $father$, $mother$, $likes$ etc. As these types of symbols may overlap, the type of a paricular symbol depends on the context where it appears. The logical connectives are: $\wedge$ (*conjunction*), $\vee$ (*disjunction*), $\neg$ (*negation*), $\leftarrow$ or $\rightarrow$ (*implication*) and $\leftrightarrow$ (*equivalence*). The quantifiers are: $\forall$ (*universal*) and $\exists$ +*existential*). The punctuation symbols are: "(", ")" and ",".

A basic element of FOL is called *term*, and is defined as follows:

- a variable is a term;
- a constant is a term;
- if f is a n-argument function ($n \geq 0$) and $t_1, t_2, ..., t_n$ are terms, then $f(t_1, t_2, ..., t_n)$ is a term.

The terms are used to construct *formulas* in the following way:

- if p is an n-argument predicate ($n \geq 0$) and $t_1, t_2, ..., t_n$ are terms, then $p(t_1, t_2, ..., t_n)$ is a formula (called *atomic formula* or just *atom*;)
- if F and G are formulas, then $\neg F$, $F \wedge G$, $F \vee G$, $F \leftarrow G$, $F \leftrightarrow G$ are formulas too;
- if F is a formula and X – a variable, then $\forall XF$ and $\exists XF$ are also formulas.

Given the alphabet, the language of FOL consists of all formulas obtained by applying the above rules.

One of the purpose of FOL is to describe the meaning of natural language sentences. For example, having the sentence "For every man there exists a woman that he loves", we may construct the following FOL formula:

$$\forall X \exists Y man(X) \rightarrow woman(Y) \wedge loves(X, Y)$$

Or, "John loves Mary" can be written as a formula (in fact, an atom) without variables (here we use lower case letters for John and Mary, because they are constants):

$$loves(john, mary)$$

Terms/formulas without variables are called *ground* terms/formulas.

If a formula has only universaly quantified variables we may skip the quantifiers. For example, "Every student likes every professor" can be written as:

$$\forall X \forall Y is(X, student) \wedge is(Y, professor) \rightarrow likes(X, Y)$$

and also as:

$$is(X, student) \wedge is(Y, professor) \rightarrow likes(X, Y)$$

Note that the formulas do not have to be always true (as the sentences they represent).

Hereafter we define a subset of FOL that is used in logic programming.

- An atom or its negation is called *literal*.
- If A is an atom, then the literals A and $\neg A$ are called *complementary*.
- A disjunction of literals is called *clause*.
- A clause with no more than one positive literal (atom without negation) is called *Horn clause*.
- A clause with no literals is called empty clause ($\square$) and denotes the logical constant "false".

There is another notation for Horn clauses that is used in *Prolog* (a programming language that uses the syntax and implement the semantics of logic programs). Consider a Horn clause of the following type:

$$A \vee \neg B_1 \vee \neg B_2 \vee ... \vee \neg B_m,$$

where $A, B_1, ..., B_m$ $(m \geq 0)$ are atoms. Then using the simple transformation $p \leftarrow q = p \vee \neg q$ we can write down the above clause as an implication:

$$A \leftarrow B_1, B_2, ..., B_m$$

.

In Prolog, instead of $\leftarrow$ we use $:-$. So, the Prolog syntax for this clause is:

$$A :- B_1, B_2, ..., B_m$$

.

Such a clause is called *program clause* (or *rule*), where A is the clause *head*, and $B_1, B_2, ..., B_m$ – the clause *body*. According to the definition of Horn clauses we may have a clause with no positive literals, i.e.

$$:- B_1, B_2, ..., B_m,$$

.

that may be written also as

$$?- B_1, B_2, ..., B_m,$$

.

Such a clause is called *goal*. Also, if $m = 0$, then we get just A, which is another specific form of a Horn clause called *fact*.

A conjunction (or set) of program clauses (rules), facts, or goals is called *logic program*.

1.2 Substitutions and unification

A set of the type $\theta = \{V_1/t_1, V_2/t_2, ..., V_n/t_n\}$, where V_i are all different variables ($V_i \neq V_j \forall i \neq j$) and t_i – terms ($t_i \neq V_i$, $i = 1, ..., n$), is called *substitution*.

Let t is a term or a clause. Substitution θ is applied to t by replacing each variable V_i that appears in t with t_i. The result of this application is denoted by $t\theta$. $t\theta$ is also called an *instance* of t. The transformation that replaces terms with variables is called *inverse substitution*, denoted by θ^{-1}. For example, let $t_1 = f(a, b, g(a, b))$, $t_2 = f(A, B, g(C, D))$ and $\theta = \{A/a, B/b, C/a, D/b\}$. Then $t_1\theta = t_2$ and $t_2\theta^{-1} = t_1$.

Let t_1 and t_2 be terms. t_1 is *more general* than t_2, denoted $t_1 \geq t_2$ (t_2 is *more specific* than t_1), if there is a substitution θ (inverse substitution θ^{-1}), such that $t_1\theta = t_2$ ($t_2\theta^{-1} = t_1$).

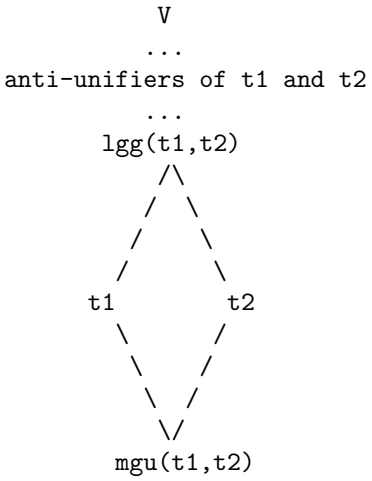
The term generalization relation induces a *lattice* for every term, where the lowemost element is the term itself and the uppermost element is a variable.

A substitution, such that, when applied to two different terms make them identical, is called *unifier*. The process of finding such a substitution is called *unification*. For example, let $t_1 = f(X, b, U)$ and $t_2 = f(a, Y, Z)$. Then $\theta_1 = \{X/a, Y/b, Z/c\}$ and $\theta_2 = \{X/a, Y/b, Z/U\}$ and both unifiers of t_1 and t_2, because $t_1\theta_1 = t_2\theta_1 = f(a, b, c)$ and $t_1\theta_2 = t_2\theta_2 = f(a, b, U)$. Two thers may have more than one unifier as well as no unifiers at all. If they have at least one unifier, they also must have a *most general unifier* (mgu). In the above example t_1 and t_2 have many unifiers, but θ_2 is the most general one, because $f(a, b, U)$ is more general than $f(a, b, c)$ and all terms obtained by applying other unifiers to t_1 and t_2.

An inverse substitution, such that, when applied to two different terms makes them identical, is called *anti-unifier*. In contrast to the unifiers, two terms have always an anti-unifier. In fact, any two terms t_1 and t_2 can be made identical by applying the inverse substitution $\{t_1/X, t_2/X\}$. Consequently, for any two terms, there exists a least general anti-unifier, which in the ML terminology we usually call *least general generalization* (lgg).

For example, $f(X, g(a, X), Y, Z) = lgg(f(a, g(a, a), b, c), f(b, g(a, b), a, a)$ and all the other anti-unifiers of these terms are more general than $f(X, g(a, X), Y, Z)$, including the most general one – a variable.

Graphically, all term operations defined above can be shown in a lattice (note that the lower part of this lattice does not always exist).



```
            ...
unifiers of t1 and t2
            ...
```

1.3 Semanics of logic programs and Prolog

Let P be a logic program. The set of all ground atoms that can be built by using predicates from P with arguments – functions and constants also from P, is called *Herbrand base* of P, denoted B_P.

Let M is a subset of B_P, and $C = A$:- $B_1, ..., B_n$ $(n \geq 0)$ – a clause from P. M is a *model* of C, if for all ground instances $C\theta$ of C, either $A\theta \in M$ or $\exists B_j, B_j\theta \notin M$. Obviously the empty clause $\square$ has no model. That is way we usually use the symbol $\square$ to represent the logic constant "false".

M is a *model of a logic program* P, if M is a model of any clause from P. The intersection of all models of P is called *least Herbrand model*, denoted M_P. The intuition behind the notion of model is to show *when a clause or a logic program is true*. This, of course depends on the context where the clause appears, and this context is represented by its model (a set of ground atoms, i.e. facts).

Let P_1 and P_2 are logic programs (sets of clauses). P_2 is a *logical consequence* of P_1, denoted $P_1 \models P_2$, if every model of P_1 is also a model of P_2.

A logic program P is called *satisfiable* (intuitively, consistent or true), if P has a model. Otherwise P is unsatisfiable (intuitively, inconsistent or false). Obviously, P is unsatisfiable, when $P \models \square$. Further, the *deduction theorem* says that $P_1 \models P_2$ is equivalent to $P_1 \wedge \neg P_2 \models \square$.

An important result in logic programming is that the least Herbrand model of a program P is unique and consists of all ground atoms that are logical consequences of P, i.e.

$$M_P = \{A | A \; is \; a \; ground \; atom, P \models A\}$$

.

In particular, this applies to clauses too. We say that a clause C *covers* a ground atom A, if $C \models A$, i.e. A belongs to all models of C.

It is interesting to find out the logical consequences of a logic program P, i.e. *what follows from a logic program*. However, according to the above definition this requires an exhaustive search through all possible models of P, which is computationally very expensive. Fortunately, there is another approach, called *inference rules*, that may be used for this purpose.

An *inference rule* is a procedure I for transforming one formula (program, clause) P into another one Q, denoted $P \vdash_I Q$. A rule I is *correct and complete*, if $P \vdash_I P$ only when $P_1 \models P_2$.

Hereafter we briefly discuss a correct and complete inference rule, called *resolution*. Let C_1 and C_2 be clauses, such that there exist a pair of literals $L_1 \in C_1$ and $L_2 \in C_2$ that can be made complementary by applying a most general unifier μ, i.e. $L_1\mu = \neg L_2\mu$. Then the clause $C = (C_1 \backslash \{L_1\} \cup C_2 \backslash \{L_2\})\mu$ is called *resolvent* of C_1 and C_2. Most importantly, $C_1 \wedge C_2 \models C$.

For example, consider the following two clauses:

$C_1 = grandfather(X, Y) : -parent(X, Z), father(Z, Y).$
$C_2 = parent(A, B) : -father(A, B).$

The resolvent of C_1 and C_2 is:

$C_1 = grandfather(X, Y) : -father(X, Z), father(Z, Y),$

where the literals $\neg parent(X, Z)$ in C_1 and $parent(A, B)$ in C_2 have been made complementary by the substitution $\mu = \{A/X, B/Z\}$.

By using the resolution rule we can check, if an atom A or a conjunction of atoms $A_1, A_2, ..., A_n$ logically follows from a logic program P. This can be done by applying a specific type of the resolution rule, that is implemented in Prolog. After loading the logic program P in the Prolog database, we can execute queries in the form of $? - A.$ or $? - A_1, A_2, ..., A_n.$ (in fact, goals in the language of logic programming). The Prolog system answers these queries by printing "yes" or "no" along with the substitutions for the variables in the atoms (in case of yes). For example, assume that the following program has been loaded in the database:

```
grandfather(X,Y) :- parent(X,Z), father(Z,Y).
parent(A,B) :- father(A,B).
father(john,bill).
father(bill,ann).
father(bill,mary).
```

Then we may ask Prolog, if $grandfather(john, ann)$ is true:

```
?- grandfather(jihn,ann).
yes
?-
```

Another query may be "Who are the grandchildren of John?", specified by the following goal (by typing ; after the Prolog answer we ask for alternative solutions):

```
?- grandfather(john,X).
X=ann;
X=mary;
no
?-
```

2 Lgg-based relational induction

θ-subsumption. Given two clauses C and D, we say that C *subsumes* D (or C is a *generalization* of D), if there is a substitution θ, such that $C\theta \subseteq D$. For example,

```
parent(X,Y):-son(Y,X)
```

θ-subsumes ($\theta = \{X/john, Y/bob\}$)

```
parent(john,bob):- son(bob,john),male(john)
```

because
$\{parent(X, Y), \neg son(Y, X)\}\theta \subseteq \{parent(john, bob), \neg son(bob, john), \neg male(john)\}$.

The θ-subsumption relation can be used to define an *lgg* of two clauses.
lgg **under θ-subsumption (*lgg*θ)**. The clause C is an *lgg*θ of the clauses C_1 and C_2 if C θ-subsumes C_1 and C_2, and for any other clause D, which θ-subsumes C_1 and C_2, D also θ-subsumes C. Here is an example:
$C_1 = parent(john, peter) : -son(peter, john), male(john)$
$C_2 = parent(mary, john) : -son(john, mary)$
$lgg(C_1, C_2) = parent(A, B) : -son(B, A)$

The *lgg* under θ-subsumption can be calculated by using the *lgg* on terms. $lgg(C_1, C_2)$ can be found by collecting all *lgg*'s of one literal from C_1 and one literal from C_2. Thus we have

$$lgg(C_1, C_2) = \{L | L = lgg(L_1, L_2), L_1 \in C_1, L_2 \in C_2\}$$

Note that we have to include in the result *all* literals L, because any clause even with one literal L will θ-subsume C_1 and C_2, however it will not be the least general one, i.e. an *lgg*.

When background knowledge BK is used a special form of *relative lgg* (or rlgg) can be defined on atoms. Assume BK is a set of facts, and A and B are facts too (i.e. clauses without negative literals). Then

$$rlgg(A, B, BK) = lgg(A :- BK, B :- BK)$$

The relative lgg (rlgg) can be used to implement an inductive learning algorithm that induces Horn clauses given examples and background knowledge as first order atoms (facts). Below we illustrate this algorithm with an example.

Consider the following set of facts (desribing a directed acyclic graph): $BK = \{link(1,2), link(2,3), link(3,4), link(3,5)\}$, positive examples $E^+ = \{path(1,2), path(3,4), path(2,4), path(1,3)\}$ and negative examples E^- – the set of all instances of $path(X,Y)$, such that there is not path between X and Y in BK. Let us now apply an rlgg-based version of the covering algorithm desribed in the previous section:

1. Select the first two positive examples $path(1,2)$, $path(3,4)$ and find their *rlgg*, i.e. the *lgg* of the following two clauses (note that the bodies of these clauses include also all positive examples, because they are part of BK):
   $path(1,2) :- link(1,2), link(2,3), link(3,4), link(3,5),$
   $\quad path(1,2), path(3,4), path(2,4), path(1,3)$
   $path(3,4) :- link(1,2), link(2,3), link(3,4), link(3,5),$
   $\quad path(1,2), path(3,4), path(2,4), path(1,3)$
   According to the above-mentioned algorithm this is the clause:
   $path(A,B) :- path(1,3), path(C,D), path(A,D), path(C,3),$
   $\quad path(E,F), path(2,4), path(G,4), path(2,F), path(H,F), path(I,4),$
   $\quad path(3,4), path(I,F), path(E,3), path(2,D), path(G,D), path(2,3),$
   $\quad link(3,5), link(3,\_), link(I,\_), link(H,\_), link(3,\_), link(3,4),$
   $\quad link(I,F), link(H,\_), link(G,\_), link(G,D), link(2,3), link(E,I),$
   $\quad link(A,\_), link(A,B), link(C,G), link(1,2).$

2. Here we perform an additional step, called *reduction*, to simplify the above clause. For this purpose we remove from the clause body:
   - all ground literals;
   - all literals that are not connected with the clause head (none of the head variables A and B appears in them);
   - all literals that make the clause *tautology* (a clause that is always true), i.e. body literals same as the clause head;
   - all literals that when removed do not reduce the clause coverage of positive examples and do not make the clause incorrect (covering negative examples).

   After the reduction step the clause is $path(A,B) :- link(A,B)$.

3. Now we remove from E^+ the examples that the above clause covers and then $E^+ = \{path(2,4), path(1,3)\}$.

4. Since E^+ is not empty, we further select two examples ($path(2,4)$, $path(1,3)$) and find their $rlgg$, i.e. the lgg of the following two clauses:
   $path(2,4) : -link(1,2), link(2,3), link(3,4), link(3,5),$
   $path(1,2), path(3,4), path(2,4), path(1,3)$
   $path(1,3) : -link(1,2), link(2,3), link(3,4), link(3,5),$
   $path(1,2), path(3,4), path(2,4), path(1,3)$
   which is:
   $path(A,B) : -path(1,3), path(C,D), path(E,D), path(C,3),$
   $path(A,B), path(2,4), path(F,4), path(2,B), path(G,B), path(H,4),$
   $path(3,4), path(H,B), path(A,3), path(2,D), path(F,D), path(2,3),$
   $link(3,5), link(3,\_), link(H,\_), link(G,\_), link(3,\_), link(3,4),$
   $link(H,B), link(G,\_), link(F,\_), link(F,D), link(2,3), link(A,H),$
   $link(E,\_), link(C,F), link(1,2).$
   After reduction we get $path(A,B) : -link(A,H), link(H,B)$.

The last two clauses form the sandard definition of a procedure to find a path in a graph.

3 Searching the space of relational hypotheses

In this section we shall discuss a basic algorithm for learning Horn clauses from examples (ground facts), based on general to specific search embeded in a covering strategy. At each pass of the outermost loop of the algorithm a new clause is generated by θ-subsumption specialization of the most general hypothesis $\top$. Then the examples covered by this clause are removed and the process continues until no uncovered exampes are left. The negative examples are used in the inner loop that finds individual clauses to determine when the current clause needs further specialization. Two types of specialization operators are applied:

1. Replacing a variable with a term.
2. Adding a literal to the clause body.

These operators are minimal with respect to θ-subsumption and thus they ensure an exhaustive search in the θ-subsumption hierarchy.

There are two stopping conditions for the inner loop (terminal nodes in the hierarchy):

- Correct clauses, i.e. clauses covering at least one positive example and no negative examples. These are used as components of the final hypothesis.
- Clauses not covering any positive examples. These are just omitted.

Let us consider an illustration of the above algorithm. The target predicate is $member(X,L)$ (returning true when X is a member of the list L). The examples are
$E^+ = \{member(a,[a,b]), member(b,[b]), member(b,[a,b])\}$,
$E^- = \{member(x,[a,b])\}$.
The most general hypothesis is $\top = member(X,L)$. A part of the generalization/specialization graph is shown in Figure 1. The terminal nodes of this graph:
$member(X,[X|Z])$
$member(X,[Y|Z]) : -member(X,Z)$
are correct clauses and jointly cover all positive examples. So, the goal of the algorithm is to reach these leaves.

A key issue in the above algorithm is the search stategy. A possible approach to this is the so called iterative deepening, where the graph is searched iteratively at depths 1, 2, 3,..., etc. until no more specializations are needed. Another appraoch is a depth-first search with an evaluation function (hill climbing). This is the approach taken in the popular system FOIL that is briefly described in the next section.

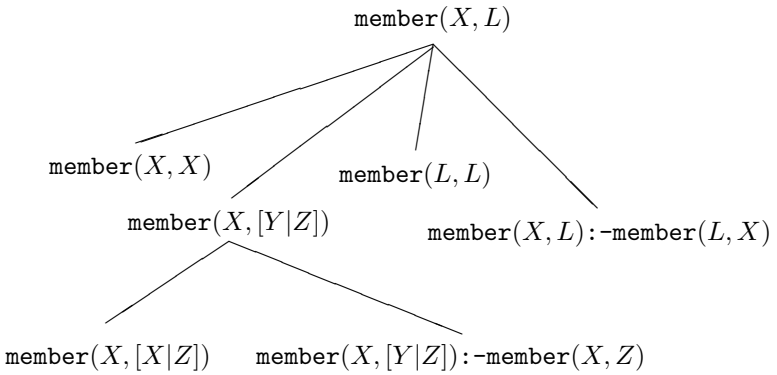


Figure 1: A generalization/specialization graph for $member(X, L)$

4 Heuristic search – FOIL

4.1 Setting of the problem

Consider the simple relational domain also discussed elsewhere – the *link* and *path* relations in a directed acyclic graph. The background knowledge and the positive examples are:

$BK = \{link(1,2), link(2,3), link(3,4), link(3,5)\}$
$E^+ = \{path(1,2), path(1,3), path(1,4), path(1,5),$
$path(2,3), path(2,4), path(2,5), path(3,4), path(3,5)\}$

The negative examples can be specified explicitly. If we assume however, that our domain is closed (as the particular *link and path* domain) the negative examples can be generated automatically using the *Closed World Assumption (CWA)*. In our case these are all ground instances of the *path* predicate with arguments – constants from E^+. Thus

$E^- = \{path(1,1), path(2,1), path(2,2), path(3,1), path(3,2), path(3,3),$
$path(4,1), path(4,2), path(4,3), path(4,4), path(4,5), path(5,1), path(5,2),$
$path(5,3), path(5,4), path(5,5)\}$

The problem is to find a hypothesis H, i.e. a Prolog definition of *path*, which satisfies the *necessity* and *strong consistency* requirements of the induction task. In other words we require that $BK \wedge H \vdash E^+$ and $BK \wedge H \nvdash E^-$. To check these condition we use *logical consequence* (called also *cover*).

4.2 Illustrative example

We start from the *most general* hypothesis

$$H_1 = path(X, Y)$$

Obviously this hypothesis covers all positive examples E^+, however many negative ones too. Therefore we have to specialize it by adding body literals. Thus the next hypothesis is

$$H_2 = path(X, Y) : -L.$$

The problem now is to find a proper literal L. Possible candidates are literals containing only variables with predicate symbols and number of arguments taken from the set E^+, i.e.

$L \in \{link(V_1, V_2), path(V_1, V_2)\}$.

Clearly if the variables V_1, V_2 are both different from the head variables X and Y, the new clause H_2 will not be more specific, i.e. it will cover the same set of negatives as H_1. Therefore we impose a restriction on the choice of variables, based on the notion of *old variables*. Old variables are those appearing in the previous clause. In our case X and Y are old variables. So, we require *at least one* of of V_1 and V_2 to be an old variable.

Further, we need a criterion to choose the *best literal* L. The system described here, FOIL uses an *information gain measure* based on the ratio between the number of positive and negative examples covered. Actually, each newly added literal has *to decrease the number of covered negatives maximizing at the same time the number of uncovered positives*. Using this criterion it may be shown that the best candidate is $L = link(X, Y)$. That is

$$H_2 = path(X, Y) : -link(X, Y)$$

This hypothesis does not cover any negative examples, hence we can stop further specialization of the clause. However there are still uncovered positive examples. So, we save H_2 as a part of the final hypothesis and continue the search for a new clause.

To find the next clause belonging to the hypothesis we exclude the positive examples covered by H_2 and apply the same algorithm for building a clause using the rest of positive examples. This leads to the clause $path(X, Y) : -link(X, Z), path(Y, Z)$, which covers these examples and is also correct. Thus the final hypothesis H_3 is the usual definition of path:

```
path(X,Y):-link(X,Y).
path(X,Y):-link(X,Z),path(Z,Y).
```

4.3 Algorithm FOIL

An algorithm based on the above ideas is implemented in the system called FOIL (First Order Inductive Learning) [1]. Generally the algorithm consists of two nested loops. The inner loop constructs a clause and the outer one adds the clause to the predicate definition and calls the inner loop with the positive examples still uncovered by the current predicate.

The algorithm has several critical points, which are important for its efficiency and also can be explored for further improvements: +

- The algorithm performs a search strategy by choosing the *locally best branch* in the search tree and further exploring it *without backtracking*. This actually is a *hill climbing* strategy which may drive the search in a local maximum and prevent it from finding the best global solution. Particularly, this means that in the inner loop there might be a situation when there are still uncovered negative examples and there is no proper literal literal to be added. In such a situation we can allow the algorithm to add a new literal without requiring an increase of the information gain and then to proceed in the usual way. This means to force a further step in the search tree hopping to escape from the local maximum. This further step however *should not lead to decrease of the information gain* and also *should not complicate the search space* (increase the branching). Both requirements are met if we choose *determinate literals* for this purpose.

  Using determinate literals however does not guarantee that the best solution can be found. Furthermore, this can complicate the clauses without actually improving the hypothesis with respect to the *sufficiency* and *strong consistency*.

- When dealing with *noise* the *strong consistency* condition can be weakened by allowing the inner loop to terminate even when the current clause covers some of the negative examples. In other words these examples are considered as noise.

- If the set of positive examples is *incomplete*, then CWA will add the missing positive examples to the set of negative ones. Then if we require strong consistency, the constructed hypothesis will be specialized to exclude the examples, which actually we want to generalize. A proper *stopping condition* for the inner loop would cope with this too.

5 Inductive Logic Programming

5.1 ILP task

Generally *Inductive Logic Programming (ILP)* is an area integrating Machine Learning and Logic Programming. In particular this is a version of the induction problem, where all languages are subsets of Horn clause logic or Prolog.

The setting for ILP is as follows. B and H are logic programs, and E^+ and E^- – usually sets of ground facts. The conditions for construction of H are:

- *Necessity:* $B \nvdash E^+$
- *Sufficiency:* $B \wedge H \vdash E^+$
- *Weak consistency:* $B \wedge H \nvdash []$
- *Strong consistency:* $B \wedge H \wedge E^- \nvdash []$

The strong consistency is not always required, especially for systems which deal with noise. The necessity and consistency condition can be checked by a theorem prover (e.g. a Prolog interpreter). Further, applying *Deduction theorem* to the sufficiency condition we can transform it into

$$B \wedge \neg E^+ \vdash \neg H \tag{1}$$

This condition actually allows to infer *deductively* the hypothesis from the background knowledge and the examples. In most of the cases however, the number of hypotheses satisfying (1) is too large. In order to limit this number and to find only useful hypotheses some additional criteria should be used, such as:

- *Extralogical restrictions* on the background knowledge and the hypothesis language.
- *Generality* of the hypothesis. The simplest hypothesis is just E^+. However, it is too specific and hardly can be seen as a generalization of the examples.
- *Decidability and tractability* of the hypothesis. Extending the background knowledge with the hypothesis should not make the resulting program indecidable or intractable, though logically correct. The point here is that such hypotheses cannot be tested for validity (applying the sufficiency and consistency conditions). Furthermore the aim of ILP is to construct real working logic programs, rather than just elegant logical formulae.

In other words condition (1) can be used to generate a number of initial approximations of the searched hypothesis, or to evaluate the correctness of a currently generated hypothesis. Thus the problem of ILP comes to *construction of correct hypotheses and moving in the space of possible hypotheses* (e.g. by generalization or specialization). For this purpose a number of techniques and algorithms are developed.

5.2 Ordering Horn clauses

A logic program can be viewed in two ways: as a *set of clauses* (implicitly conjoined), where each clause is a *set of literals* (implicitly disjoined), and as a logical formula in *conjunctive normal form* (conjunction of disjunction of literals). The first interpretation allows us to define a clause ordering based on the subset operation, called θ - *subsumption*.

5.2.1 θ-subsumption

θ-subsumption. Given two clauses C and D, we say that C *subsumes* D (or C is a *generalization* of D), iff there is a substitution θ, such that $C\theta \subseteq D$.

For example,

```
parent(X,Y):-son(Y,X)
```

θ-subsumes ($\theta = \{X/john, Y/bob\}$)

```
parent(john,bob):- son(bob,john),male(john)
```

since

$\{parent(X,Y), \neg son(Y,X)\}\theta \subseteq \{parent(john,bob), \neg son(bob,john), \neg male(john)\}$.

θ-subsumption can be used to define an *lgg* of two clauses.

lgg **under θ-subsumption (*lgg*θ)**. The clause C is an *lgg*θ of the clauses C_1 and C_2 iff C θ-subsumes C_1 and C_2, and for any other clause D, which θ-subsumes C_1 and C_2, D also θ-subsumes C.

Consider for example the clauses $C_1 = p(a) \leftarrow q(a), q(f(a))$ and $C_2 = p(b) \leftarrow q(f(b))$. The clause $C = p(X) \leftarrow a(f(X))$ is an *lgg*θ of C_1 and C_2.

The *lgg* under θ-subsumption can be calculated by using the *lgg* on terms. Consider clauses C_1 and C_2. $lgg(C_1, C_2)$ can be found by collecting all *lgg*'s of one literal from C_1 and one literal from C_2. Thus we have

$$lgg(C_1, C_2) = \{L | L = lgg(L_1, L_2), L_1 \in C_1, L_2 \in C_2\}$$

Note that we have to include in the result *all* such literals L, because any clause even with one literal L will θ-subsume C_1 and C_2, however it will not be the least general one, i.e. an *lgg*.

5.2.2 Subsumption under implication

When viewing clauses as logical formulae we can define another type of ordering using *logical consequence (implication)*.

Subsumption under implication. The clause C_1 is *more general* than clause C_2, (C_1 *subsumes under implication* C_2), iff $C_1 \vdash C_2$. For example, $(P :- Q)$ is more general than $(P :- Q, R)$, since $(P :- Q) \vdash (P :- Q, R)$.

The above definition can be further extended by involving a *theory* (a logic program).

Subsumption relative to a theory. We say that C_1 subsumes C_2 w.r.t. theory T, iff $P \wedge C_1 \vdash C_2$.

For example, consider the clause:

```
cuddly_pet(X) :- small(X), fluffy(X), pet(X)        (C)
```

and the theory:

```
pet(X) :- cat(X)                                    (T)
pet(X) :- dog(X)
small(X) :- cat(X)
```

Then C is more general than the following two clauses w.r.t. T:

```
cuddly_pet(X) :- small(X), fluffy(X), dog(X)        (C1)
cuddly_pet(X) :- fluffy(X), cat(X)                  (C2)
```

Similarly to the terms, the ordering among clauses defines a lattice and clearly the most interesting question is to find the *least general generalization* of two clauses. It is defined as follows. $C = lgg(C_1, C_2)$, iff $C \geq C_1$, $C \geq C_2$, and any other clause, which subsumes both C_1 and C_2, subsumes also C. If we use a relative subsumption we can define a *relative least general generalization (rlgg)*.

The subsumption under implication can be tested using *Herbrand's theorem*. It says that $F_1 \vdash F_2$, iff for every substitution σ, $(F_1 \wedge \neg F_2)\sigma$ is false ([]). Practically this can be done in the following way. Let F be a clause or a conjunction of clauses (a theory), and $C = A :- B_1, ..., B_n$ - a clause. We want to test whether $F \wedge \neg C$ is always false for any substitution. We can check that by skolemizing C, adding its body literals as facts to F and testing whether A follows from the obtained formula. That is, $F \wedge \neg C \vdash []$ is equivalent to $F \wedge \neg A \wedge B_1 \wedge ... \wedge B_n \vdash []$, which in turn is equivalent to $F \wedge B_1 \wedge ... \wedge B_n \vdash A$. The latter can be checked easily by Prolog resolution, since A is a ground literal (goal) and $F \wedge B_1 \wedge ... \wedge B_n$ is a logic program.

5.2.3 Relation between θ-subsumption and subsumption under implication

Let C and D be clauses. Clearly, if $C\theta$-subsumes D, then $C \vdash D$ (this can be shown by the fact that all models of C are also models of D, because D has just more disjuncts than C). However, the opposite is not true, i.e. from $C \vdash D$ does not follow that C θ-subsumes D. The latter can be shown by the following example.

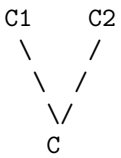
Let $C = p(X) \leftarrow q(f(X))$ and $D = p(X) \leftarrow q(f(f(X)))$. Then $C \vdash D$, however C does not θ-subsume D.

5.3 Inverse Resolution

A more constructive way of dealing with clause ordering is by using *the resolution principle*. The idea is that the resolvent of two clauses is subsumed by their conjunction. For example, $(P \vee \neg Q \vee \neg R) \wedge Q)$ is more general than $P \vee \neg R$, since $(P \vee \neg Q \vee \neg R) \wedge Q) \vdash (P \vee \neg R)$. The clauses C_1 and C_2 from the above example are resolvents of C and clauses from T.

The resolution principle is an effective way of deriving logical consequences, i.e. *specializations*. However when building hypothesis we often need an algorithm for inferring *generalizations* of clauses. So, this could be done by an inverted resolution procedure. This idea is discussed in the next section.

Consider two clauses C_1 and C_2 and its resolvent C. Assume that the resolved literal appears positive in C_1 and negative in C_2. The three clauses can be drawn at the edges of a "V" – C_1 and C_2 at the arms and C – at the base of the "V".



A resolution step derives the clause at the base of the "V", given the two clauses of the arms. In the ILP framework we are interested to infer the clauses at the arms, given the clause at the base. Such an operation is called *"V" operator*. There are two possibilities.

A "V" operator which given C_1 and C constructs C_2 is called *absorption*. The construction of C_1 from C_2 and C is called *identification*.

The "V" operator can be derived from the equation of resolution:

$$C = (C_1 - \{L_1\})\theta_1 \cup (C_2 - \{L_2\})\theta_2$$

where L_1 is a positive literal in C_1, L_2 is a negative literal in C_2 and $\theta_1\theta_2$ is the *mgu* of $\neg L_1$ and L_2.

Let $C = C'_1 \cup C'_2$, where $C'_1 = (C_1 - \{L_1\})\theta_1$ and $C'_2 = (C_2 - \{L_2\})\theta_2$. Also let $D = C'_1 - C'_2$. Thus $C'_2 = C - D$, or $(C_2 - \{L_2\})\theta_2 = C - D$. Hence:

$$C_2 = (C - D)\theta_2^{-1} \cup \{L_2\}$$

Since $\theta_1\theta_2$ is the *mgu* of $\neg L_1$ and L_2, we get $L_2 = \neg L_1\theta_1\theta_2^{-1}$. By θ_2^{-1} we denote an *inverse substitution*. It replaces terms with variables and uses *places* to select the term arguments to be replaced by variables. The places are defined as n-tuples of natural numbers as follows. The term at place <i> within $f(t_0, .., t_m)$ is t_i and the term at place $<i_0, i_1, .., i_n>$ within $f(t_0, .., t_m)$ is the term at place $<i_1, .., i_n>$ within t_{i_0}. For example, let $E = f(a, b, g(a, b))$, $Q = f(A, B, g(C, D))$. Then $Q\sigma = E$, where $\sigma = \{A/a, B/b, C/a, D/b\}$. The inverse substitution of σ is $\sigma^{-1} = \{<a,<0>>/A,<b,<1>>/B,<a,<2,0>>/C,<b,<2,1> /D\}$. Thus $E\sigma^{-1} = Q$. Clearly $\sigma\sigma^{-1} = \{\}$.

Further, substituting L_2 into the above equation we get

$$C_2 = ((C - D) \cup \{\neg L_1\}\theta_1)\theta_2^{-1}$$

The choice of L_1 is unique, because as a positive literal, L_1 is the head of C_1. However the above equation is still not well defined. Depending on the choice of D it give a whole range of solutions, i.e. $\oslash \cap D \cap C'_1$. Since we need the *most specific* C_2, D should be $\oslash$. Then we have

$$C_2 = (C \cup \{\neg L_1\}\theta_1)\theta_2^{-1}$$

Further we have to determine θ_1 and θ_2^{-1}. Again, the choice of most specific solution gives that θ_2^{-1} has to be empty. Thus finally we get the *most specific solution of the absorption operation* as follows:

$$C_2 = C \cup \{\neg L_1\}\theta_1$$

The substitution θ_1 can be partly determined from C and C_1. From the resolution equation we can see that $C_1 - \{L_1\})$ θ-subsumes C with θ_1. Thus a part of θ_1 can be constructed by matching literals from C_1 and C, correspondingly. However for the rest of θ there is a free choice, since θ_1 is a part of the *mgu* $\neg L_1$ and L_2 and L_2 is unknown. This problem can be avoided by assuming that every variable within L_1 also appear in C_1. In this case θ can be fully determined by matching all literals within $(C_1 - \{L_1\})$ with literals in C. Actually this is a constraint that all variables in a head (L_1) of a clause (C_1) have to be found in its body $(C_1 - \{L_1\})$. Such clauses are called *generative* clauses and are often used in the ILP systems.

For example, given the following two clauses

```
mother(A,B) :- sex(A,female),daughter(B,A)                    (C1)
grandfather(a,c) :- father(a,m),sex(m,female),daughter(c,m)   (C )
```

the absorption "V" operator as derived above will construct

```
grandfather(a,c) :- mother(m,c),father(a,m),
                    sex(m,female),daughter(c,m)               (C2)
```

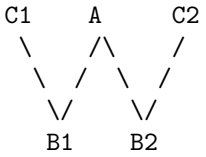
Note how the substitution θ_1 was found. This was done by unifying a literal from `C` – `daughter(c,m)` with a literal from `C1` – `daughter(B,A)`. Thus $\theta_1 = \{A/m, B/c\}$ and $L_1\theta_1 =$ `mother(m,c)`. (The clause `C1` is generative.)

The clause `C2` can be reduced by removing the literals `sex(m,female)` and `daughter(c,m)`. This can be done since these two literals are redundant (`C2` without them resolved with `C1` will give the same result, `C`). Thus the result of the absorption "V" operator is finally

```
grandfather(a,c) :- mother(m,c),father(a,m)                    (C2)
```

5.4 Predicate Invention

By combining two resolution V's back-to-back we get a *"W" operator*.



Assume that C_1 and C_2 resolve on a common literal L in A and produce B_1 and B_2 respectively. The "W" operator constructs A, C_1 and C_2, given B_1 and B_2. It is important to note that the literal L does not appear in B_1 and B_2. So, the "W" operator has to introduce a *new predicate symbol*. In this sense this predicate is *invented* by the "W" operator.

The literal L can appear as negative or as positive in A. Consequently there are to types of "W" operators - *intra-construction* and *inter-construction* correspondingly.

Consider the two resolution equations involved in the "W" operator.

$$B_i = (A - \{L\})\theta_{A_i} \cup (C_i - \{L_i\})\theta_{C_i}$$

where $i \in \{1, 2\}$, L is negative in A, and positive in C_i, and $\theta_{A_i}\theta_{C_i}$ is the *mgu* of $\neg L$ and L_i. Thus $(A - \{L\})$ θ-subsumes each clause B_i, which in turn gives one possible solution $(A - \{L\}) = lgg(B_1, B_2)$, i.e.

$$A = lgg(B_1, B_2) \cup \{L\}$$

Then θ_{A_i} can be constructed by matching $(A - \{L\})$ with literals of B_i.

Then substituting A in the resolution equation and assuming that θ_{C_i} is empty (similarly to the "V" operator) we get

$$C_i = (B_i - lgg(B_1, B_2)\theta_{A_i}) \cup \{L_i\}$$

Since $L_i = \neg L\theta_{A_i}\theta_{C_i}^{-1}$, we obtain finally

$$C_i = (B_i - lgg(B_1, B_2)\theta_{A_i}) \cup \{\neg L\}\theta_{A_i}$$

For example the intra-construction "W" operator given the clauses

```
grandfather(X,Y) :- father(X,Z), mother(Z,Y)                   (B1)
grandfather(A,B) :- father(A,C), father(C,B)                   (B2)
```

constructs the following three clauses (the arms of the "W").

```
p1(_1,_2) :- mother(_1,_2)                                     (C1)
p1(_3,_4) :- father(_3,_4)                                     (C2)
grandfather(_5,_6) :- p1(_7,_6), father(_5,_7)                 (A )
```

The "invented" predicate here is `p1`, which obviously has the meaning of "parent".

5.5 Extralogical restrictions

The background knowledge is often restricted to *ground facts*. This simplifies substantially all the operations discussed so far. Furthermore, this allows all ground hypotheses to be derived directly, i.e. in that case $B \wedge \neg E^+$ is a set of positive and negative literals.

The hypotheses satisfying all logical conditions can be still too many and thus difficult to construct and generate. Therefore *extralogical* constraints are often imposed. Basically all such constraint restrict the language of the hypothesis to a smaller subset of Horn clause logic. The most often used subsets of Horn clauses are:

- *Function-free clauses* (Datalog). These simplifies all operations discussed above. Actually each clause can be transformed into a function-free form by introducing new predicate symbols.
- *Generative clauses*. These clauses require all variables in the clause head to appear in the clause body. This is not a very strong requirement, however it reduces substantially the space of possible clauses.
- *Determinate literals*. This restriction concerns the body literals in the clauses. Let P be a logic program, $M(P)$ – its model, E^+ – positive examples and $A : -B_1, ..., B_m$, $B_{m+1}, ..., B_n$ – a clause from P. The literal B_{m+1} is *determinate*, iff for any substitution θ, such that $A\theta \in E^+$, and $\{B_1, ..., B_m\}\theta \subseteq M(P)$, there is a *unique* substitution δ, such that $B_{m+1}\theta\delta \in M(P)$.

  For example, consider the program

```
p(A,D):-a(A,B),b(B,C),c(C,D).
a(1,2).
b(2,3).
c(3,4).
c(3,5).
```

  Literals $a(A, B)$ and $b(B, C)$ are determinate, but $c(C, D)$ is not determinate.

5.6 Illustrative examples

In this section we shall discuss three simple examples of solving ILP problems.

Example 1. Single example, single hypothesis.

Consider the background knowledge B

```
haswings(X):-bird(X)
bird(X):-vulture(X)
```

and the example $E^+ = \{haswings(tweety)\}$. The ground unit clauses, which are logical consequences of $B \wedge \neg E^+$ are the following:

$C = \neg bird(tweety) \wedge \neg vulture(tweety) \wedge \neg haswings(tweety)$

This gives three most specific clauses for the hypothesis. So, the hypothesis could be any one of the following facts:

```
bird(tweety)
vulture(tweety)
haswings(tweety)
```

Example 2.

Suppose that $E^+ = E_1 \wedge E_2 \wedge ... \wedge E_n$ is a set of ground atoms, and C is the set of ground unit positive consequences of $B \wedge \neg E^+$. It is clear that

$$B \wedge \neg E^+ \vdash \neg E^+ \wedge C$$

Substituting for E^+ we obtain

$$B \wedge \neg E^+ \vdash (\neg E_1 \wedge C) \vee (\neg E_2 \wedge C) \vee ... \vee (\neg E_n \wedge C)$$

Therefore $H = (E_1 \vee \neg C) \wedge (E_1 \vee \neg C) \wedge ... \wedge (E_n \vee \neg C)$, which is a set of clauses (logic program).

Consider an example.

$B = \{father(harry, john), father(john, fred), uncle(harry, jill)\}$

$E^+ = \{parent(harry, john), parent(john, fred)\}$

The ground unit positive consequences of $B \wedge \neg E^+$ are

$C = father(harry, john) \wedge father(john, fred) \wedge uncle(harry, jill)$

Then the most specific clauses for the hypothesis are $E_1 \vee \neg C$ and $E_2 \vee \neg C$:

```
parent(harry,john):-father(harry,john),
                    father(john,fred),
                    uncle(harry,jill)

parent(john,fred):-father(harry,john),
                    father(john,fred),
                    uncle(harry,jill)
```

Then $lgg(E_1 \vee \neg C, E_2 \vee \neg C)$ is

```
parent(A,B):-father(A,B),father(C,D),uncle(E,F)
```

This clause however contains redundant literals, which can be easily removed if we restrict the language to determinate literals. Then the final hypothesis is:

```
parent(A,B):-father(A,B)
```

Example 3. Predicate Invention.

$B = \{min(X, [X]), 3 > 2\}$

$E^+ = \{min(2, [3, 2]), min(2, [2, 2])\}$

The ground unit-positive consequences of $B \wedge \neg E^+$ are the following:

$C = min(2, [2]) \wedge min(3, [3]) \wedge 3 > 2$

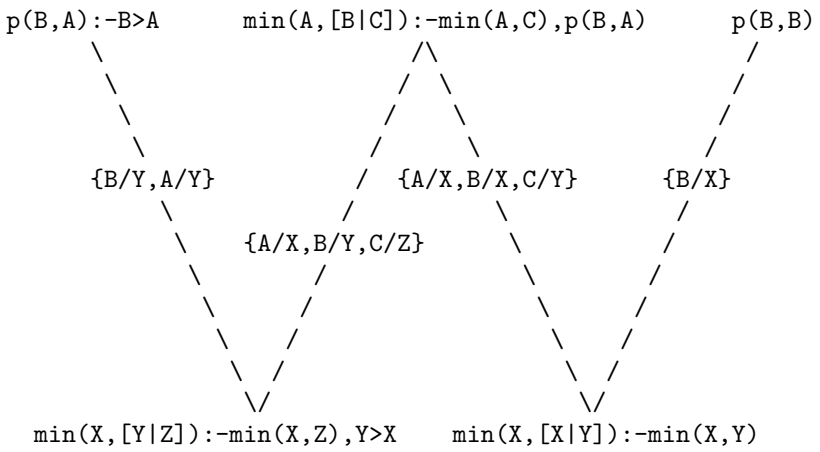
As before we get the two most specific hypotheses:

```
min(2,[3,2]):-min(2,[2]),min(3,[3]),3>2
min(2,[2,2]):-min(2,[2]),min(3,[3]),3>2
```

We can now generalize and simplify these clauses, applying the restriction of determinate literals.

```
min(X,[Y|Z]):-min(X,Z),Y>X
min(X,[X|Y]):-min(X,Y)
```

Then we can apply the "W"-operator in the following way (the corresponding substitutions are shown at the arms of the "W"):



Obviously the semantics of the "invented" predicate `p` is "$\geq$" (greater than or equal to).

5.7 Basic strategies for solving the ILP problem

Generally two strategies can be explored:

- *Specific to general search.* This is the approach suggested by condition (1) allowing deductive inference of the hypothesis. First, a number of most specific clauses are constructed and then using "V", "W", *lgg* or other generalization operators this set is converged in one of several generalized clauses. If the problem involves negative examples, then the currently generated clauses are tested for correctness using the strong consistency condition. This approach was illustrated by the examples.

- *General to specific search.* This approach is mostly used when some heuristic techniques are applied. The search starts with the most general clause covering E^+. Then this clause is further specialized (e.g. by adding body literals) in order to avoid covering of E^-. For example, the predicate $parent(X, Y)$ covers E^+ from example 2, however it is too general and thus coves many other irrelevant examples too. So, it should be specialized by adding body literals. Such literals can be constructed using predicate symbols from B and E^+. This approach is explored in the system FOIL [1].

References

[1] J. R. Quinlan. Learning logical definitions from relations. *Machine Learning*, 5:239–266, 1990.