Chapter 8

Inductive Logic Programming

8.1 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 (see Chapter 2), 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

$$B \wedge \neg E^+ \vdash \neg H \tag{8.1}$$

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.

8.2 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.*

8.2.1 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 (*lggθ*)**. The clause C is an *lggθ* 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 *lggθ* 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*.

8.2.2 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.

8.2.3 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.

8.3 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".

```
C1    C2
 \     /
  \  /
   \/
   C
```

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)
```

8.4 Predicate Invention

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

```
C1     A      C2
 \     /\     /
  \   /  \   /
   \/     \/
   B1     B2
```

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".

8.5 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.

8.6 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"):

```
   p(B,A):-B>A      min(A,[B|C]):-min(A,C),p(B,A)      p(B,B)
         \                      /\                      /
          \                    /  \                    /
           \                  /    \                  /
            \                /      \                /
         {B/Y,A/Y}          /  {A/X,B/X,C/Y}      {B/X}
              \            /          \            /
               \    {A/X,B/Y,C/Z}      \          /
                \        /              \        /
                 \      /                \      /
                  \    /                  \    /
                   \  /                    \  /
                    \/                      \/
     min(X,[Y|Z]):-min(X,Z),Y>X    min(X,[X|Y]):-min(X,Y)
```

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

8.7 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) (Section 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 [Quinlan, 1990].