Inductive Logic Programming

- ILP is the learning sector employing logic programming as a representation language of the examples and concepts

Classification systems

- There are different dimensions of ILP systems [Lav94].
- They may require that all the examples are used to learn the model (**batch learners**) or they can accept them one by one (**incremental learners**).
- They can learn one preached at a time (**single predicate learners**) or more than one (**multiple predicate learners**).
- During learning they can ask questions to the user to check the validity of generalizations and/or to classify examples generated by the system (**interactive learners**) or have no interaction with the outside (**non-interactive learners**).

Classification systems

- Finally, a system can learn a theory from scratch or from a pre-existing theory of incomplete and / or inconsistent (**theory revisors**).
- Although they are independent dimensions, existing systems belong to two opposing groups.
- On the one hand there are the batch systems, noninteractive who learn from scratch a concept at a time (**empirical systems**), On the other hand there are incremental, interactive systems that make theory revision and learn more predicates at a time (**interactive or incremental systems**)

Defining the problem of ILP

Formally:

- Data:

  a set $\mathcal{P}$ of possible programs

  a set E^+ good examples

  a set E^- Negative examples

  a consistent logic program B, such that

  $B \not\models e^+$, For at least one and$^+ \in IS^+$

- Find:

  a logic program $P \in \mathcal{P}$ such that

  $\forall\ e^+ \in E^+$, B, P $\models e^+$ (P is **complete**)

  $\forall\ e^- \in E^-$, B, P $\not\models e^-$ (P is **consistent**)

Terminology

Terminology:

- B: **background knowledge**, A priori knowledge, predicates of which we already know the definition.
- E^+ E^- constitute the **training set**
- P **target program**, B,P is the program obtained by adding the P clauses after those of B.
- $\mathcal{P}$ It is told **hypothesis space**. It defines the search space. The description of such hypothesis space takes the name of **language bias**.

- B, P |= E **"P covers e"**

Example

- Learning of the predicate father.

Data

$\mathcal{P}$: Logic programs containing clauses father (X, Y): - τo with τo literal conjunction chosen from

parent(X, Y), parent(Y, X), male(X), male(Y),

female (X), female (Y)

B = {parent (john, mary), male(john)

parent (david, steve), male(david)

parent (kathy, ellen), female (kathy)}

Example

E^+ = {Father (john, mary), father (david, steve)}

E^- = {Father (kathy, ellen), father (john, steve)}

Find:

- a program to compute the predicate father that is consistent and complete with respect to E^+, E^-

Possible solution:

father (X, Y): - parent (X, Y), male(X).

Generality relation

- The systems learns a theory by searching in the space of the clauses
- In the case of inductive logic programming the space is subject to a generality relation θ-subsumption [Plo69]:
- The C1 clause **θ-subsumes** C2 if there exists a substitution θ such that C1θ ⊆ C2.

- A clause C1 and at least as general as clause C2 (and we write C1 ≥ C2) if C1 θ-subsumes C2
- C1 is more general than C2 (C1> C2) if C1 ≥ C2 but not C2 ≥ C1

Examples of report of θ-sussunzione

C1 = grandfather (X, Y)←father (X, Z).

C2 = grandfather (X, Y)←father (X, Z), parent (Z, Y).

C3 = grandfather (john, steve)←father (john, mary), parent (mary, steve).

C1 θ-subsumes C2 with empty substitution θ=∅

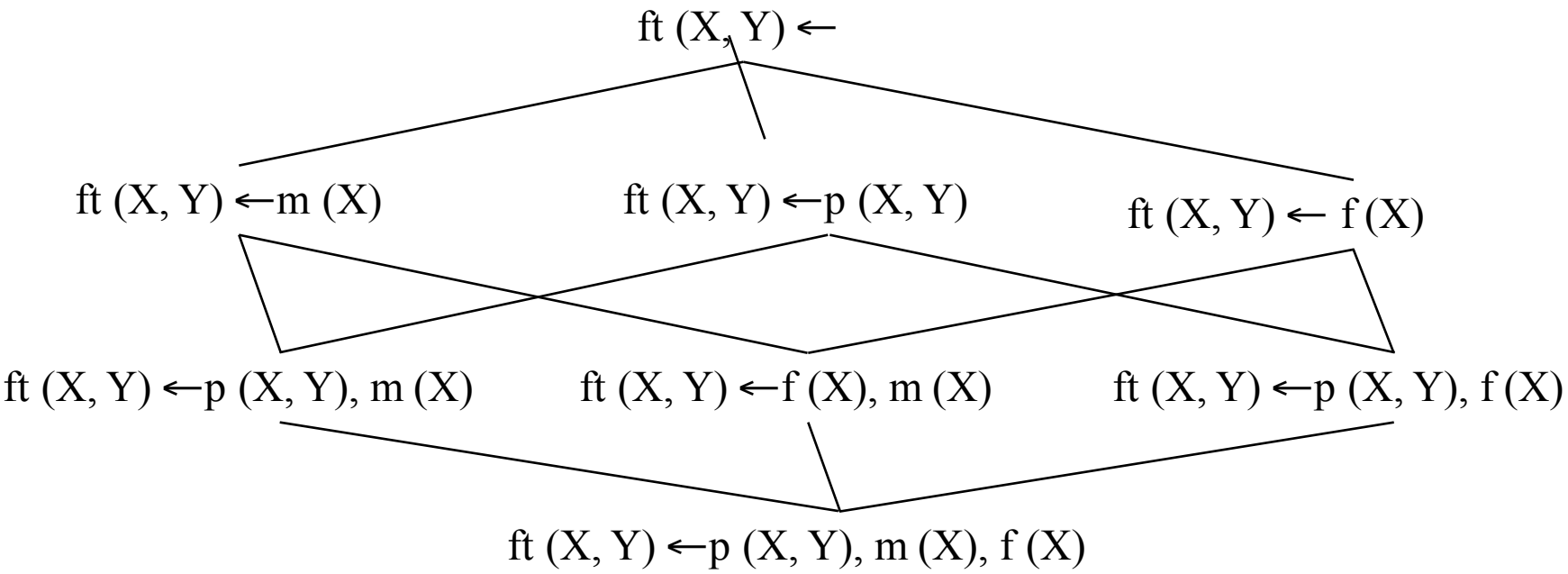
C1 θ-subsumes C3 with the substitution θ= {X / john, Y / steve, Z / mary}.

C2 θ-subsumes C3 with the substitution θ= {X / john, Y / steve, Z / mary}.

Property of θ-subsumption

- θ-subsumption has the important property that if C1 θ-subsumes C2 then C1 | = C2.
- The vice versa is not true
- For this reason θ-subsumption is used to represent the generality relation
- There θ-subsumption has another important property: it induces a lattice in the set of clauses.
- This means that each pair of clauses has a least upper bound (lub) and a greatest lower bound (glb).

θ-subsumption lattice



ILP algorithms

- The algorithms for ILP are divided into **bottom-up** is **Top-down** [Ber96] depending on whether use of generalization and specialization operators.
- The bottom-up algorithms start from the examples (specific or bottom) and generalize them getting a theory that covers them.
  - They use of generalization operators obtained by reversing the deductive rules of unification, resolution and implication.

Top-down algorithms

- The top-down algorithm learns programs by generating the clauses sequentially (one after the other).
- The generation of a clause starts from the most general clause (with empty body) and specializing it (by applying a specialized operator or refinement) until it no longer covers any negative example.

Top-down algorithm basic

TopDown algorithm (E, B, H): given the training set E and the background knowledge B returns H

H: = 0

E_{cur}: = E

repeat / * covering loop * /

generateClause (Ecur, B, C)

Add the clause to the theory H: H =∪ {C}

E_{cur}: = E_{cur}-covers (B, H, E_{cur})

//Removes from E_{cur} positive examples covered by H

until no positive examples are left

Specialization loop

GenerateClause(E, B, C): given the training set E and the background B, it returns the best clause C

Choose a predicate to learn

C: = P (X): - true.

repeat / * specialization loop* /

Find refining $C_{best} \in \rho(C)$ using a heuristic

C: = C_{best}

until no negative examples are covered

Heuristics

- The simplest heuristic is accuracy

$$A(C) = n^+(C) / (n^+(C) + n^-(C))$$

- where $n^+(C)$ $n^-(C)$ are, respectively, the positive and negative examples covered by the clause C
- Another heuristic and informativeness, defined as

$$I(C) = - \log_2(n^+(C) / (n^+(C) + n^-(C)))$$

- Other heuristics are:
  - the accuracy of gain GA (C ', C) = A (C') - A (C)
  - the information gain GI (C ', C) = I (C') - I (C)

Example

Example

$\mathcal{P}$: Father (X, Y): - α with

α ={Parent (X, Y), parent (Y, X),
male(X), male (Y), female (X), female (Y)}

B = {parent (john, mary), male (john)
parent (david, steve), male(david)
parent (kathy, ellen), female (kathy)}

E^+= {father (john, mary), father (david, steve)}
E^-= {Father (kathy, ellen), father (john, steve)}

Example

1st specialization step

father (X, Y): - true.

and it covers E^+ but also E^-

Step 2

father (X, Y): - parent (X, Y).

and it covers E^+ but also one e^- father (kathy, ellen).

Step 3

father (X, Y): - parent (X, Y), male(X).

and it covers E^+ and any E^-

The covered positive examples are removed, E^+ It becomes empty and the algorithm terminates generating the theory:

father (X, Y): - parent (X, Y), male(X).

Bottom-up algorithms

- The bottom-up algorithms are based on the concept of least general generalization (lgg). The lgg of two clauses C1 and C2 is a clause C that:
  - is more general than C1 and C2,
  - is the most specific among the most general clauses C1 and C2.

Least general generalization

- θ-subsumption has the important property that introduces a lattice in the set of clauses. This means that each pair of clauses has a least upper bound (lub) and a greatest lower bound (glb).

- **Definition: least general generalization [Plo69]**
- The least general generalization of two clauses C1 and C2, denoted by lgg (C1, C2), is the least upper bound of C1 and C2 in the lattice of the θ-subsumption.

Algorithm to calculate the lgg

- The lgg of two **terms** f1(s1,..., sn) and f2(t1, ..., tn) is:
  - f1(lgg (s1, t1), ..., lgg (sn, tn)) if f1 = f2 or
  - a new variable V if f1 ≠ f2.
- Sa same variable is used to generalize the same terms.
- Examples:
  - lgg (f (a, b, c), f (a, c, d)) = f (a, X, Y)
  - lgg(f(a,a), f(b,b)) = f (lgg(a,b), lgg (a, b)) = f (X, X)

Algorithm to calculate the lgg

- Thelgg of two **literals** L1 = (~) p (s1, ..., sn) and L2 = (~) q (t1, ..., tn) is undefined if L1 and L2 do not have the same predicate symbol and sign, otherwise it is defined as

- lgg (L1, L2) = (~) p (lgg (s1, t1), ... lgg (sn, tn))
- **Examples:**

  lgg (parent (john, mary), parent (john, steve)) = parent (john, X)

  lgg (parent (john, mary), ~ parent (john, steve)) = undefined

  lgg (parent (john, mary), father (john, steve)) = undefined

Algorithm to calculate the lgg

The lgg of two clauses C1 = {L1, ..., Ln}
C2 = {K1, ..., Km} is defined as:

lgg (C1, C2) = {lgg (Li, Kj) | Li $\in$ C1, Kj $\in$ C2
and lgg (Li, Kj) is defined}

Lgg of two clauses

Examples:

C1 = father (john, mary) ← parent (john, mary), male (john)

C2 = father (david, steve) ← parent (david, steve), male(david)

lgg (C1, C2) = father (X, Y) ← parent (X, Y), male(X)

C1 = win (conf1) ← occ (place1, x, conf1), occ(Place2, o, conf1)

C2 = win (conf2) ← occ (place1, x, conf2), occ(Place2, x, conf2)

lgg (C1, C2) = win (Conf) ← occ (place1, x, Conf), occ(L, x, Conf),
occ(M, Y, Conf), occ(Place2, Y, Conf)

Bottom-up methods

- In the bottom-up methods, the clauses are generated starting from the most specific clause covering one or more positive examples and no negative example and applying repeatedly generalization operators to clause until it can no longer be further generalized without covering negative examples.
- Typically there is also a further step in which you eliminate by literal body of the generated clause until the clause does not cover the negative examples

Bibliography

[Ber96] F. and D. Bergadano Gunetti, *Inductive Logic Programming - From Machine Learning to Software Engineering*, MIT Press, Cambridge, Massachusetts, 1996

[Cam94] RM Cameron-Jones and J. Ross Quinlan, *Efficient Top-Down Induction of Logic Programs*, SIGART, 5, p. 33-42, 1994.

[Lav94] Lavrac N. and S. Dzeroski, *Inductive Logic Programming Techniques and Applications*, Ellis Horwood, 1994

Bibliography

[Mug95] Stephen Muggleton, *reverse entailment and progol*New January Comput., 13: 245-286, ftp: //ftp.cs.york.ac.uk/ Pub / ML_GROUP /Papers/InvEnt.ps.gz.

[Mug90] S. Muggleton and C. Feng, *Efficient induction of logic programs*, Proceedings of the 1st Conference on Algorithmic learning Theory Ohmsma, Tokyo, p. 368-381, 1990.

[Plo69] GD Plotkin. *A note on inductive generalization*. In B.Meltzer and D. Michie, editors, *Machine Intelligence 5*, pages 153-163. Edinburgh University Press, Edinburgh1969.

Bibliography

[Qui91] JR Quinlan, *Certain literals in inductive logic programming*, Proceedings of the Twelfth International Joint Conference on Artificial Intelligence, Morgan Kaufmann, 1991.

[Qui90] JR Quinlan, *Learning logical definitions from relations*, Machine Learning, 5: 239-- 266, 1990.

[Qui93a] JR Quinlan and RM Cameron-Jones, *FOIL: A Midterm Report*, Proceedings of the 6th European Conference on Machine Learning, Springer-Verlag ", 1993.