ILP Learning problem

Target relation: variabilized head of the desired PROLOG program.

Examples: Positive and negative ground examples of the target relation.

Background Knowledge: A set of literals that can be used as conditions in the program and their definition.

Find: A definition for the target relation.

Example

Target Relation: `father(A,B)`

Training Instances:

⊕:	⊖:
`father(christopher,arthur).`	`father(penelope,arthur).`
`father(christopher,victoria).`	`father(christopher,penelope).`

Background Knowledge:

```
male(christopher).          parent(christopher,arthur).
male(arthur).               parent(christopher,victoria).
                            parent(penelope,arthur).
female(victoria).           parent(penelope,victoria).
female(penelope).
```

Separate-and-Conquer Rule Learning

Finds a complete and consistent theory

To Find a Theory:

1. Add a rule that covers many positive and no negative examples (*generalization*)
2. Remove the examples that are covered by the rule (*separate*)
3. If there are uncovered positive examples (*incompleteness*) goto 1. (*conquer*)

To Find a Rule:

1. Add a condition that excludes many negative and few positive examples (*specialization*)
2. If the rule still covers negative examples (*inconsistency*) goto 1.

Basic Algorithm

```
procedure SeparateAndConquer(Examples)

Theory = ∅
while Positive(Examples) ≠ ∅
        Clause = ∅
        Cover = Examples
        while Negative(Cover) ≠ ∅
                Clause = Clause ∪ SelectCondition(Clause, Cover)
                Cover = Cover(Clause, Cover)
        Examples = Examples − Cover
        Theory = Theory ∪ Clause
return(Theory)
```

Selecting a Condition

1. For each predicate in the background knowledge generate all possible variabilization with variables used in head and body of $Clause$ and with new variables.

2. Choose the best literal according to some heuristic.

```
procedure SelectCondition(Clause,Examples)

Vmax = 0
Literals = AllPossibleLiterals(Clause)
for Literal in Literals
    V = HeuristicValue(Clause,Literal,Examples)
    if V > Vmax
        Vmax = V
        BestLiteral = Literal
return(BestLiteral)
```

AllPossibleLiterals

All predicates from the background knowledge with all possible variabilizations will be considered as conditions for the rule.

Example:
Head = `father(A,B)`

Possible Conditions to Extend the Clause:

male(A)	\+male(A)	parent(A,A)	parent(B,A)
male(B)	\+male(B)	\+parent(A,A)	\+parent(B,A)
female(A)	\+female(A)	parent(A,B)	parent(B,B)
female(B)	\+female(B)	\+parent(A,B)	\+parent(B,A)

```
%%%
%%% EXAMPLE FOR SEPARATE-AND-CONQUER
%%% LEARNING ALGORITHM
%%%

| ?- foil(father(A,B),Clauses).

2 positive and 2 negative instances left.

male(A):          covers 2 + and 1 -
\+male(A):        covers 0 + and 1 -
male(B):          covers 1 + and 1 -
\+male(B):        covers 1 + and 1 -
female(A):        covers 0 + and 1 -
\+female(A):      covers 2 + and 1 -
female(B):        covers 1 + and 1 -
\+female(B):      covers 1 + and 1 -
parent(A,A):      covers 0 + and 0 -
\+parent(A,A):    covers 2 + and 2 -
parent(A,B):      covers 2 + and 1 -
\+parent(A,B):    covers 0 + and 1 -
parent(B,A):      covers 0 + and 0 -
\+parent(B,A):    covers 2 + and 2 -
parent(B,B):      covers 0 + and 0 -
\+parent(B,A):    covers 2 + and 2 -

Chose literal male(A) (covers 2 + and 1 -)
male(A):          covers 2 + and 1 -
\+male(A):        covers 0 + and 0 -
male(B):          covers 1 + and 0 -
\+male(B):        covers 1 + and 1 -
female(A):        covers 0 + and 0 -
\+female(A):      covers 2 + and 1 -
female(B):        covers 1 + and 1 -
\+female(B):      covers 1 + and 0 -
parent(A,A):      covers 0 + and 0 -
\+parent(A,A):    covers 2 + and 1 -
parent(A,B):      covers 2 + and 0 -
\+parent(A,B):    covers 0 + and 1 -
parent(B,A):      covers 0 + and 0 -
\+parent(B,A):    covers 2 + and 1 -
parent(B,B):      covers 0 + and 0 -
\+parent(B,A):    covers 2 + and 1 -

Chose literal parent(A,B) (covers 2 + and 0 -)

Found clause: father(A,B):-male(A),parent(A,B)


Clauses = [(father(A,B):-male(A),parent(A,B))] ?

yes
| ?-
```

Which literals shall we prefer?

We have a clause c' that covers p' positive and n' negative examples. How shall we select a literal l such that $c = c' \cup l$ is "as good as" possible.

- The goal is to find a *complete* and *consistent* program, i.e. a program that covers all positive and no negative clauses.
- Clauses should therefore cover many positive examples without covering any negative examples.
- To find such a clause we select literals that exclude many negative and only few positive examples.

Problem: Define a measurement $M(c)$ for the quality of a clause.

Search Heuristics

Coverage of $\oplus$: $C(c) = p$

Idea: Select a literal that covers as many positive examples as possible.

Problem: Optimizes only one criterion. The literal `true` will be best, because it covers all positive examples. However, this is a bad choice, because it does not exclude any negative examples.

Exclusion of $\ominus$: $E(c) = (-)n$

Idea: Select a literal that excludes as many negative examples as possible.

Problem: Optimizes only one criterion. The literal `false` will be best, because it excludes all negative examples. This is a bad choice, because it also excludes all positive examples.

Better Search Heuristics

Accuracy: $A(c) = \frac{p}{p+n}$

Idea: Percentage of positive examples of all covered examples. Will be best (1.0) when only positive and no negative examples are covered and worst (0.0) when only negative examples are covered.

Problem: Insensitive to the number of covered examples (a clause covering 1 positive example is as good as a clause covering all positive examples.

Information Content: $I(c) = -\log_2 \frac{p}{p+n}$

Idea: An event X with probability $P(X)$ contains an information of $-\log_2 P(X)$ bits *(Shannon, 1948)*. The less information I need to determine the class ($\oplus$ or $\ominus$) the better.

Problem: See above.

Good Heuristics

Weighted Heuristics: $WH(c) = w \times H(c)$

Idea: Multiply a heuristic with a weight (which usually depends on the number of covered examples).

Gain Heuristics: $HG(c) = H(c) - H(c')$

Idea: The difference between the heuristic values of the old clause c' and the new clause $c = c' \cup l$. Often used in connection with weighted heuristics.

(Simplified) Weighted Information Gain:

$$WIG(c) = p \times (\log_2 \frac{p}{p+n} - \log_2 \frac{p'}{p'+n'})$$

Idea: Each of the p positive examples covered by c would need an information of $I(c')$ bits for determining its class label in the old clause and $I(c)$ bits in the new clause. Therefore we gain $p \times (I(c') - I(c))$ bits by adding l to c'.

```
| ?- foil(father(A,B),Clauses).

2 positive and 2 negative instances left.

male(A):         Gain: 0.83 (2+/1-)
\+male(A):       Gain: 0.00 (0+/1-)
male(B):         Gain: 0.00 (1+/1-)
\+male(B):       Gain: 0.00 (1+/1-)
female(A):       Gain: 0.00 (0+/1-)
\+female(A):     Gain: 0.83 (2+/1-)
female(B):       Gain: 0.00 (1+/1-)
\+female(B):     Gain: 0.00 (1+/1-)
parent(A,A):     Gain: 0.00 (0+/0-)
\+parent(A,A):   Gain: 0.00 (2+/2-)
parent(A,B):     Gain: 0.83 (2+/1-)
\+parent(A,B):   Gain: 0.00 (0+/1-)
parent(B,A):     Gain: 0.00 (0+/0-)
\+parent(B,A):   Gain: 0.00 (2+/2-)
parent(B,B):     Gain: 0.00 (0+/0-)
\+parent(B,B):   Gain: 0.00 (2+/2-)

Chose literal male(A). Gain: 0.83 (2+/1-).
male(A):         Gain: 0.00 (2+/1-)
\+male(A):       Gain: 0.00 (0+/0-)
male(B):         Gain: 0.58 (1+/0-)
\+male(B):       Gain:-0.42 (1+/1-)
female(A):       Gain: 0.00 (0+/0-)
\+female(A):     Gain: 0.00 (2+/1-)
female(B):       Gain:-0.42 (1+/1-)
\+female(B):     Gain: 0.58 (1+/0-)
parent(A,A):     Gain: 0.00 (0+/0-)
\+parent(A,A):   Gain: 0.00 (2+/1-)
parent(A,B):     Gain: 1.17 (2+/0-)
\+parent(A,B):   Gain: 0.00 (0+/1-)
parent(B,A):     Gain: 0.00 (0+/0-)
\+parent(B,A):   Gain: 0.00 (2+/1-)
parent(B,B):     Gain: 0.00 (0+/0-)
\+parent(B,B):   Gain: 0.00 (2+/1-)
Chose literal parent(A,B). Gain: 1.17 (2+/0-).

Found clause: father(A,B):-male(A),parent(A,B)

Clauses = [(father(A,B):-male(A),parent(A,B))]

yes
| ?-
```

Introducing New Variables

In some domains it is necessary to introduce new variables in the body of a clause.

Example: `grandfather(A,B) :- father(A,C), parent(C,B).`

Method: The new variable can be bound to all values that can appear in the place of the new variable in the background knowledge for a fixed binding of the old variables.

Counting:

- count whether there is a binding that yields a positive example or not (counting *instances*)
- count for all possible bindings for the new variable whether they yield a positive examples or not (counting *proofs*)

Family Relations Examples

Figure 2 from (Quinlan, 1990)

New Variables — Example

Target Relation: `grandfather(A,B)`

Training Instances:

⊕:
```
grandfather(christopher,colin).
grandfather(christopher,charlotte).
grandfather(andrew,colin).
grandfather(andrew,charlotte).
```

⊖:
```
grandfather(penelope,colin).
grandfather(christine,charlotte).
grandfather(victoria,christopher).
grandfather(james,charlotte).
grandfather(christopher,james).
grandfather(andrew,james).
```

Background Knowledge:

```
father(X,Y).
parent(X,Y).
```

Counting Instances

Empty Clause:

Coverage:	$4\oplus/6\ominus$
Information Content:	$-\log_2 \frac{4}{4+6} = 1.32$ bits

Selecting literal `father(A,C)`*:*

Coverage:	$4\oplus/3\ominus$
Information Content:	$-\log_2 \frac{4}{4+3} = 0.81$ bits
Information Gain:	$4 \times (1.32 - 0.81) = 2.05$ bits

Selecting literal `parent(C,B)`*:*

Coverage:	$4\oplus/0\ominus$
Information Content:	0 bits
Information Gain:	$4 \times (0.81 - 0.0) = 3.23$ bits

Counting Proofs

The program keeps a set of *tuples* consisting of possible instantiations for the variables in the clause plus the classification.

Training Tuples for `grandparent(A,B)`*:*

$<\oplus,\mathtt{christopher},\mathtt{colin}>$	$<\oplus,\mathtt{christopher},\mathtt{charlotte}>$
$<\oplus,\mathtt{andrew},\mathtt{colin}>$	$<\oplus,\mathtt{andrew},\mathtt{charlotte}>$
$<\ominus,\mathtt{penelope},\mathtt{colin}>$	$<\ominus,\mathtt{christine},\mathtt{charlotte}>$
$<\ominus,\mathtt{victoria},\mathtt{christopher}>$	$<\ominus,\mathtt{james},\mathtt{charlotte}>$
$<\ominus,\mathtt{christopher},\mathtt{james}>$	$<\ominus,\mathtt{andrew},\mathtt{james}>$

Empty Clause:

Coverage: $4\oplus/6\ominus$

Information Content: $-\log_2 \frac{4}{4+6} = 1.32$ bits

When a new variable is introduced by a literal, the tuple set is extended by an additional column. One tuple is generated for each possible instantiation of this variable. The classification is inherited from the tuple without the additional column.

Counting Proofs (Ctd)

Selecting literal `father(A,C)`*:*

Coverage:	$8\oplus/6\ominus$ (!!!)
Information Content:	$-\log_2 \frac{8}{8+6} = 0.81$ bits
Information Gain:	$4(!!!) \times (1.32 - 0.81) = 2.05$ bits

Training Tuples for `grandparent(A,B) :- father(A,C)`*:*

$<\oplus$,christopher,colin,victoria$>$	$<\oplus$,andrew,colin,james$>$
$<\oplus$,christopher,colin,arthur$>$	$<\oplus$,andrew,colin,jennifer$>$
$<\oplus$,christopher,charlotte,victoria$>$	$<\oplus$,andrew,charlotte,james$>$
$<\oplus$,christopher,charlotte,arthur$>$	$<\oplus$,andrew,charlotte,jennifer$>$
$<\ominus$,james,charlotte,colin$>$	$<\ominus$,james,charlotte,charlotte$>$
$<\ominus$,christopher,james,victoria$>$	$<\ominus$,christopher,james,arthur$>$
$<\ominus$,andrew,james,james$>$	$<\ominus$,andrew,james,jennifer$>$

Selecting literal `parent(C,B)`*:*

Coverage:	$4\oplus/0\ominus$ (!!!)
Information Content:	0 bits
Information Gain:	$8 \times (0.81 - 0.0) = 3.23$ bits

Learning Recursive Concepts

Recursive literals can be evaluated like any other literal when the positive training instances have been added to the background knowledge.

Ordering of Clauses: Base case may be learned after recursive clause ⇒ *reordering* after learning

Missing Examples: All examples of a recursive chain must be present.

Example: If `ancestor(colin,victoria)` and `ancestor(colin, christopher)` is known, we cannot learn a recursive rule for the latter, unless we also know `ancestor(victoria,christopher)`.

Infinite recursive call: Some variabilizations of the recursive call have to be forbidden to prevent trivial concepts like

```
ancestor(A,B) :- ancestor(A,B).
```

Assuring Recursive Soundness

1. Admit only different variabilizations in head and body.

   Problem: `married(X,Y) :- married(Y,X).`

2. Recursive calls must contain a variable that does not appear in the head of the clause.

   Problem: `parent(X,Y) :- married(X,Z),parent(Z,Y).`

3. Establish an *irreflexive partial order* of the constants and make sure that $A_i \leq B_i$ or $B_i \leq A_i$ for recursive calls of the form

   $$\texttt{r(}\ldots A_i \ldots\texttt{) :- } \ldots\texttt{, r(}\ldots B_i \ldots\texttt{).}$$

   This ensures that the recursive call will eventually reach the "largest" or "smallest" element in the domain and terminate.

   Problem: `r(A,X) :- r(A-1,Y),r(A+1,Z),q(Y,Z,X).`

4. Establish an order between variabilizations of recursive literals (Cameron-Jones & Quinlan, IJCAI-93).

FOIL

First **O**rder **I**nductive **L**earner (Quinlan, 1990)

Characteristics:

- **Instance Space:** PROLOG ground facts
- **Hypothesis Space:** Recursive function-free PROLOG programs
- **Search Direction:** top-down (general-to-specific) search
- **Search Strategy:** separate-and-conquer, hill-climbing
- **Search Heuristic:** weighted information gain
- **Enhancements:** determinate literals, noise handling, checkpoints, implicit definition of negative examples (closed world), ...

`C` *implementation* available from `ftp://ftp.cs.su.oz.au/pub/foil6.sh`