

Inductive Logic Programming

James Cussens

University of York, UK

Overview of the day

This lecture 'Vanilla' ILP

Next lecture ILP for grammar induction

Practical Exercises on both sorts of ILP

Overview of this lecture

- The basic problem
- Searching
- Scoring
- Using Aleph

The basic ILP problem

- Given B , E - both logic programs, find H , a logic program $\in \mathcal{H}$ such that

$$B, H \vdash E$$

- Need to stop H being 'too strong', e.g. $H = \text{false}$
Use negative examples or syntactic constraints
- Need some way of stopping H being 'too weak', e.g. $H = E$
Bias towards simplicity or syntactic constraints

The basic ILP problem - Version 2

Normally negative examples are assumed . . .

Given: Positive examples P , Negative examples N and Background knowledge B .

Find $H \in \mathcal{H}$ st

1. $\forall e \in P : B \wedge H \models e$ (H is complete)
2. $\forall e \in N : B \wedge H \not\models e$ (H is consistent)

Learning daughter/2

INPUT

<i>Training examples</i>		<i>Background knowledge</i>
<i>daughter(mary, ann).</i>	\oplus	<i>mother(ann, mary). female(ann).</i>
<i>daughter(eve, tom).</i>	\oplus	<i>mother(ann, tom). female(mary).</i>
<i>daughter(tom, ann).</i>	\ominus	<i>father(tom, eve). female(eve).</i>
<i>daughter(eve, ann).</i>	\ominus	<i>father(tom, ian).</i>
		<i>parent(X, Y) \leftarrow mother(X, Y)</i>
		<i>parent(X, Y) \leftarrow father(X, Y)</i>

OUTPUT

daughter(X, Y) \leftarrow female(X), parent(Y, X)

Learning daughter/2 – Version 2

INPUT

<i>Training examples</i>		<i>Background knowledge</i>
<i>daughter(mary, ann).</i>	\oplus	<i>mother(ann, mary). female(ann).</i>
<i>daughter(eve, tom).</i>	\oplus	<i>mother(ann, tom). female(mary).</i>
<i>daughter(tom, ann).</i>	\ominus	<i>father(tom, eve). female(eve).</i>
<i>daughter(eve, ann).</i>	\ominus	<i>father(tom, ian).</i>

OUTPUT

daughter(X, Y) ← female(X), mother(Y, X)

daughter(X, Y) ← female(X), father(Y, X)

The search

- Even if we know what we're looking for (say, the smallest H satisfying our constraints),
- actually finding it is the difficult bit.
- We have to use a heuristic search in the space \mathcal{H} .
- Top-down v bottom-up is a basic dichotomy.

Clause-at-a-time learning

- Almost always ILP systems induce a multi-clause logic program by finding one clause at a time.
- Not a bad compromise . . .
- . . . especially if H is not recursive (Think about it!)

Clause-at-a-time learning with Aleph

- Let's see some clauses induced using the Aleph system.
- Example from *Morphosyntactic tagging of Slovene using Progol* by Cussens, Džeroski and Erjavec, ILP99
- Aleph used to be called P-Progol.

Understanding the demo-ed application

```
rmv(Left,Right,Focus) :-  
    case(Left,_Case,_), case(Focus,a,_),  
    gender(Left,m,_), disongender(Left,Focus).
```

Left is all tags to the left of Focus.

Right is all tags to the left of Focus.

“Can’t have the sequence [Left,Focus] if Left has some case, Focus has accusative case, Left has masculine gender and Focus has some other gender.”

Top-down search

- Start with most general clause e.g. `rmv(A,B,C)`
- and specialise by adding literals
- (or instantiating variables)
- This is called *refinement*

Which is the best clause?

- Aleph's default clause evaluation is coverage
- coverage defines clause utility to be $P - N$, where P , N are the number of positive and negative examples covered by the clause.
- Aleph has another 9 evaluation functions to choose from!

Defining the hypothesis space \mathcal{H}

- Probably the most crucial step in using ILP!
- Let's go through the Aleph method for doing this.
- There are many others.

Choosing background predicates

- The most basic step.
- Which predicates can appear in clauses?

```
:- determination(rmv/3,noun/2).
```

```
:- determination(rmv/3,verb/2).
```

```
:- determination(rmv/3,adjective/2).
```

```
...
```

User-defined constraints on acceptable clauses

Just write what's allowed in Prolog!

```
prune((_:-Body)) :-  
    has_redundant_literals(Body).
```

```
false :-  
    hypothesis(Head,Body,_PosNegCoverage), %Aleph built-in  
    Body=(_,-,_), %only false out at clauselength  
    arg(3,Head,Focus),  
    \+ uses_focus(Body,Focus).
```

`prune/1` disallows a clause *and all its refinements*

Limiting the length of clauses

- Each clause search can be exponential in the maximum allowed length of a clause.
- So limit it!

```
:- set(clauselength,5).
```

Beware: the Aleph default is `clauselength=4`
This **includes** the head literal.

Saturation - a bit of bottom up

- *Saturating* an example produces the most specific clause which (together with B) entails the example.
- It's called the *bottom clause*
- Use `sat/1` to do this in Aleph. Like this ...
- Let's see how Aleph did that.

Mode declarations

- Aleph uses *mode declarations* to build the bottom clause
- Modes declare the types of all arguments of all predicates;
- and whether the argument is
 1. an input variable (+)
 2. an output variable (-)
 3. or a constant term

Types

- Types do not have to be defined
- Unless you're doing positive-only learning...
- ...in which case use a monadic predicate to define them

Using mode declarations

```
:- modeh(1,rmv(+msdlist1,+msdlist,+msdlist2)).
```

So each argument of `rmv/3` is an input variable, and they all have different types.

(We'll come back to the number 1.)

Attaching type labels

```
:- modeh(1,rmv(+msdlist1,+msdlist,+msdlist2)).
```

So when saturating example 21:

```
[rmv([spsg,afpms1..],[spsg,afpsi,..],[mcnsg1])
```

1. [spsg,afpms1..] is an input of type msdlist1
2. [spsg,afpsi..] is an input of type msdlist
3. [mcnsg1] is an input of type msdlist2

Finding body literals for the bottom clause

`[spsg,afpms1..]` is an input of type `msdlist1`

We have `:- modeb(1,noun(+msdlist1,-msdlist1))`.

So Aleph first calls:

```
noun([spsg,afpms1..],X)
```

which fails

Finding body literals for the bottom clause

Because of `:- modeb(1,adposition(+msdlist1,-msdlist1)).`, Aleph eventually calls:

```
adposition([spsg,afpms1..],X)
```

which succeeds with `X = [afpms1,...]`.

This is how the `adposition(A,D)` got into the bottom clause.

Variables and constants in the bottom clause

Aleph internally builds up a **ground** clause in this way:

```
rmv([spsg,afpms1,...],[spsg,...],[mcmsal_--y]) :-  
    adposition([spsg,afpms1,...],[afpms1,...]),  
    case([spsg,afpms1,...],g,[afpms1,...]),  
    ...
```

But eventually variablises some of these constant terms according to the mode declarations.

Variable depth

- Variables in the head have depth 0
- (Roughly) an output variable generated by an input variable of depth i has depth $i + 1$.
- `:- set(i,5).` limits variables in the bottom clause to depth 5.
- Default is $i = 2$. Pretty small!

The recall number - according to Aleph manual

```
mode(RecallNumber, PredicateMode)
```

RecallNumber bounds the non-determinacy of a form of predicate call [when building up the bottom-clause]

RecallNumber can be either (a) a number specifying the number of successful calls to the predicate; or (b) * specifying that the predicate has bounded non-determinacy. It is usually easiest to specify RecallNumber as *.

Why build a bottom clause?

- Pure top-down search can waste time constructing clauses *which cover no positive examples!*
- In each Aleph search *only clauses which are generalisations of the bottom clause are constructed.*
- It follows that they all cover the saturated example

The basic Aleph algorithm

1. Select example (default at random)
2. Saturate it
3. Do top-down search for 'best' clause
4. Remove covered positive examples
5. If there remain positive examples, go to 1

Advanced use of Aleph

- Each step of Aleph's basic algorithm can be altered.
- Let's just look at:
 1. Noise-handling
 2. Evaluation functions
 3. User-defined refinement

Noise

- In real applications, we have to accept clauses which cover negative examples.
- Use `set(minacc,+Float)` and/or
- `set(noise,+PositiveInt)`
- Default is noise-free learning

Evaluating clauses

Here's one way to change the clause evaluation function:

```
:- set(evalfn,mestimate).  
:- set(m,1).
```

This uses a Bayesian smoothed measure of accuracy to evaluate clauses.

User-defined refinement

```
:- set(refine,user).
```

```
refine(false, eastbound(_)).
```

```
refine(eastbound(X), (eastbound(X):-has_car(X,_))).
```

```
refine(eastbound(X), (eastbound(X):-has_car(X,Y), short(Y))).
```

```
refine((eastbound(X):-has_car(X,Y), short(Y)), Clause):-
```

```
    Clause = (eastbound(X):-has_car(X,Y), short(Y), closed(Y)).
```

The Aleph clause search is non-greedy

- Aleph keeps searching for a clause, *until it is sure it has found a maximally good one*
- *This* is what can make it slow
- It is not because of some inherent slowness of ILP or Prolog (as is commonly thought).
- `set(openlist,0)` effects greedy search

On background knowledge

- It is crucial (for reasonable size applications) that your background predicates are written in efficient Prolog.
- *Always* exploit first-argument indexing.

On the hypothesis space

- Reduce your hypothesis space as much as possible
- You can use Prolog to define it precisely
- This is good (i) computationally and (ii) statistically
- Tip: `set(minpos,+V)` is a good way to avoid overfitting and can give big speed-ups.