CS 6604: Data Mining Fall 2007

Lecture 15 — Oct 10, 2007

Lecture: Naren Ramakrishnan *Scribe: Yifei Ma*

1 Overview

In the last lecture we introduced the relational data mining software PROGOL. We went through the syntax/format of its input, output files, and obtained an idea of its usage.

In this lecture we continue with PROGOL and describe its algorithms and functioning in more detail.

2 Mode declarations

2.1 Mode head and Mode body

We learned from the last lecture that mode describes the legal predicates that can be used in the head (modeh) and body (modeb) of any hypothesized clause. Because the concept being learnt might require multiple rules, each with different predicates, we might need multiple modeb declarations. For instance the Grammar learning problem from the handout requires as background knowledge predicates that define determiners, verb phrases, and noun phrases. Even the head predicate might require multiple modeh declarations, depending on the different ways in which the predicate may be instantiated.

2.2 '+'/'-' signs in mode declarations

As we know, the + sign indicates that an argument is an input variable, and the - sign indicates an output variable. Let's learn more about the + and - signs from the following examples.

- **Grandfather examples:** Assume that we wish to learn the following rule:

  ```
  gfather(x,y) :- father(x,z), father(z,y).
  ```

  The introduction of the first `father(x,z)` predicate can be supported by a body mode (modeb) declaration such as:

  ```
  :- modeb(*,father(+person, -person))?
  ```

  This means that the second argument (i.e., `z`) can be 'cooked up' by the predicate and need not be instantiated at the time the predicate is introduced in the rule. The second predicate could be introduced by a declaration such as:

  ```
  :- modeb(*,father(+person, +person))?
  ```

since at this time, both of its arguments (i.e., `z` and `y`) have been defined previously.

Here is another example of a rule to represent grandfather.

```
gfather(x,y) :- father(z,y), father(x,z).
```

The first predicate `father(z,y)` could be introduced by a mody mode declaration such as:

```
:- modeb(*,father(-person, +person)))?
```

since `z` is the output variable.

- **child relation:** The rule

  ```
  child(x,y) :- parent(y,z).
  ```

  can be learnt using a mode declaration such as:

  ```
  :- modeb(*,parent(+person,-person))?
  ```

  Observe that output variables essentially help introduce new parameters. Although they are necessary to model certain clauses, indiscriminate use of them would require more search.

2.3 Recall in mode declarations

The recall (the `'*'` in the above examples) is used to bound the number of alternative solutions for instantiating the atom. A recall of '*' means any number of possible solutions. A constant n means at most n different instantiations.

Parent relation: If we write a body mode declaration for the parent relation as:

```
:- modeb(*, parent(-person, +person))?
```

where the first argument is the parent of the second argument, we can replace the the recall `'*'` by constant 2. That is because every person at most has two parents. This will be a hint to the ILP system to not search for more solutions after two have been found.

On the other hand, if the declaration was such that the child argument was the output:

```
:- modeb(*, parent(+person, -person))?
```

In this case, we **cannot** replace the '*' by any constant, since parents may have any number of children.

Here's a final example, the aunt relation, to help us understand the use of recall in mode declarations. Consider:

```
:- modeb(1, aunt(+person, +person))?
```

This declaration assumes both arguments as input. Obviously, since the predicate is either satisfied or not, the recall is set to 1.

3 Construction of the most specific clause

The construction of the most specific clause is conducted by taking into account the background knowledge. For example, from the positive example, $s([boy, eats, food], [])$, we get the specific clause $\perp$

$$s(A, B) : -sub(A, C), verb(C, D), obj(D, B).$$

Similarly, from 'A tall boy hungerly eats dorm food' we might get:

$$s(A, B) : -np(A, C), vp(C, D), np(D, B)$$

4 Searching the space of clauses

To search the subsumption lattice ROGOL applies an A*-like algorithm. It searches the lattice $C, \Box \preceq C \preceq \perp$, and employs f_s (for a given clause s) as the heuristic estimate to choose the best branch to visit in the lattice. Checkout the PROGOL manual for a definition of the f_s statistic:

$$f_s = p_s - (n_s + c_s + h_s)$$

where

- p_s is the number of positive examples correctly deducible from s
- n_s is the number of negative examples incorrectly deducible from s
- c_s is the length of clause s
- h_s is the number of further atoms to complete the clause

The last value is determined by estimating how many predicates need to be introduced in order to supply instantiations of all arguments. In this regard, it is helpful to note that sometimes we might want a rule where some arguments are 'don't care' arguments. For instance, assume that `p1(A,B)` means person `A` is sent to jail `B`, and `p2(A,C)` means that person `A` is guilty of crime `C`. Then the rule

```
p1(A,B) :- p2(A,C).
```

essentially means the predicate logic expression:

$$\forall a\ \forall b\ ((\exists c\ p_2(a, c)) \Rightarrow p_1(a, b))$$

5 Covering algorithms

PROGOL uses a simple set cover algorithm to deal with multiple examples. It learns from a single example and 'runs' this generalization through the dataset to verify it. The generalization is added to the background theory. Any examples which are redundant w.r.t. this generalization are now removed. The next example is picked up and generalized, and so on.