# Symbolic AI: Task 2

Anna Hausberger
annaahausii@gmail.com

Bernhard Gstrein
bernhard-gstrein@mailbox.org

May 31, 2020

## 1 Preparation

We follow all instructions and get ready for the exercise. On a technical note, we rename `aleph-swi.pl` to `aleph.pl` because the default name makes problems in Prolog.

## 2 Getting started

### 2.1 (a)

Most information we can find in the background file. First we have the mode declarations; for the head of a hypothesised clause we have

```
:- modeh(1,eastbound(+train)).
```

The command `modeh` declares a mode for the head of a hypothesised clause. Here that means whatever we put in `eastbound/1` has to be a `train`. The number 1 means that we can successfully call the predicate once. There's also `modeb` for a literal in the body of a hypothesised clause. Next, the determinations. They look like this:

```
:- determination(eastbound/1,short/1).
```

Statements like these determine the predicated that can be used to construct a hypothesis. For example, Aleph wouldn't be allowed to argue that a train is eastbound because it is a train unless we allow it. However, it's allowed to argue that a train is eastbound because (among other things) it is short. The type definitions are self-explanatory, `car 11` is a car, `elipse` is a shape and `east1` is a train. Then we have the description of the trains, for example `east2` has cars 21, 22 and 23 or car 21 is an open car or car 31 has one circle as load.

In the positive examples file `train.f` we tell Aleph which trains are going east, i.e. `east1`, `east2` and so on. In the negative examples file we can see `eastbound(west6)` and so on. So the goal is to tell which trains are eastbound or not.

Trying to find a rule by looking at the trains is hard. A smaller rule would be that long cars with only one object in it have 3 wheels while long cars with 0 or more than 1 objects in it have 2 wheels.

## 2.2 (b)

We call `induce` to learn a solution. With `show(theory)` we display the learned rules:

```
[theory]

[Rule 1] [Pos cover = 5 Neg cover = 0]
eastbound(A) :-
   has_car(A,B), short(B), closed(B).

[Training set performance]
          Actual
        +          -
     +  5          0          5
Pred
     -  0          5          5

        5          5         10

Accuracy = 1
[Training set summary] [[5,0,0,5]]
```

As for us, we were really surprised by the simplicity of the solution. A train is eastbound if it has a car that is short and closed. A problem might be that we oftentimes think too complicated when trying to figure out something. The table shows the results; upper left and lower right are true positives and true negatives. Upper right and lower left are false positives and false negatives. Aleph was able to construct a theory that perfectly fits the training data with 100% accuracy.

With `show(pos)` and `show(neg)` we show the current positive and negative examples. There's also other options like `show(determinations)`, `show(modehs)`, `show(sizes)` and more.

## 2.3 (c)

With `show(bottom)` we can look at the bottom clause, i.e. the most specific clause constructed by Aleph that entails all examples and is within language restrictions provided.

```
[bottom clause]
eastbound(A) :-
   has_car(A,B), has_car(A,C), has_car(A,D), has_car(A,E),
   short(E), short(C), closed(C), long(D),
   long(B), open_car(E), open_car(D), open_car(B),
   shape(E,rectangle), shape(D,rectangle), shape(C,rectangle),
   shape(B,rectangle), wheels(E,2), wheels(D,3), wheels(C,2),
   wheels(B,2), load(E,circle,1), load(D,hexagon,1),
   load(C,triangle,1), load(B,rectangle,3).
```

The bottom clause describes actually the first train going east. If you look at the image of the first train, you can see that this train has four cars, these four cars are described in very detail by the bottom clause. For example that the car B is long, has a rectangular shape, two wheels and is loaded with three rectangles. This means that currently the bottom clause is constructed by the first example of the positive entries, how the bottom clause is constructed can be changed by setting Aleph parameters.

The next step is to find a more general clause than the bottom clause by searching for a subset of the bottom clause, which has the "best" score. A subset of this bottom clause is also the theory clause, which can be seen in the bottom clause with the following statements:

```
[theory]

eastbound(A) :-
   has_car(A,B), short(B), closed(B).

[bottom clause]

eastbound(A) :-
    [...], has_car(A,C), [..]. , short(C), closed(C), [...].
```

# 3 Recursion

## 3.1 (a)

In Prolog, `member/2` is defined as

```
member(X,[X|Y]).
member(X,[_|Y]) :-
    member(X,Y).
```

We load in the files from `examples/recursion`. Noteworthy in the background knowledge file is

```
:- set(i,3).
:- set(noise,0).
```

The statement `set(i,3)` sets the upper bound on layers of new variables to 3 and the statement `set(noise,0)` sets the upper bound on the number of negative examples allowed to be covered by an acceptable clause to 0 (it's 0 by default anyway). Upon calling `induce`, Aleph yields

```
[theory]

[Rule 1] [Pos cover = 12 Neg cover = 0]
mem(A,B) :-
   B=[A|C].

[Rule 2] [Pos cover = 10 Neg cover = 0]
mem(A,B) :-
   B=[C|D], mem(A,D).

[Training set performance]
          Actual
        +          -
     + 19         0         19
Pred
     -  0          6          6

       19         6         25

Accuracy = 1
[Training set summary] [[19,0,0,6]]
```

Aleph found two rules that match the definition of `member/2` exactly. Let's take a look at the bottom clause:

```
[bottom clause]
mem(A,B) :-
   B=[C|D], mem(C,B), mem(A,D), D=[A|E].
```

## 3.2 (b)

Let's try out Aleph with `append/3`. It's defined as

```
append([],Xs,Xs).
append([X|Xs],Ys,[X|Zs]) :-
    append(Xs,Ys,Zs).
```

The append algorithm has three inputs, the first input is the element (`+any`), which should be added to the list (`+list`), which results in the third input (`+result`), a list with the element added. This is how we defined the background knowledge file:

```
:- mode(*,app(+any,+list,+result)).
% split both lists up in a new element and a new list.
:- mode(1, ((+list) = ([-any|-list]))).
:- mode(1, ((+result) = ([-any|-result]))).

:- set(i,3).
:- set(noise,0).

:- determination(app/3,app/3).
:- determination(app/3,'='/2).
```

The two lists `+list` and `+result` will be splitted into a new list and result and one head element. Then we defined the following positive examples:

```
app(0,[1],[1,0]).
app(1,[0],[0,1]).
app(2,[1],[1,2]).
app(3,[4],[4,3]).
app(2,[4],[4,2]).
app(3,[4],[4,3]).
app(4,[5],[5,4]).
app(1,[3,4],[3,4,1]).
app(2,[3,4],[3,4,2]).
app(3,[4,5],[4,5,3]).
app(4,[5,6],[5,6,4]).
```

And added a few negative examples:

```
app(2,[4],[4,6]).
app(3,[4],[4,5]).
app(4,[5],[3,4]).
app(1,[3,4],[3,4,3]).
app(4,[1],[1,2,3]).
```

Running this script we get the following bottom clause and learned theory:

```
[bottom clause]
app(A,B,C) :-
   B=[D|E], C=[D|F], app(A,E,F), app(A,B,F),
   E=[G|H], F=[G|I], app(G,H,C), app(G,E,C),
   app(G,B,C), app(A,H,F), I=[A|J].
```

```
[theory]

[Rule 1] [Pos cover = 7 Neg cover = 0]
app(A,B,C) :-
   B=[D|E], C=[D|F], F=[A|G].

[Rule 2] [Pos cover = 4 Neg cover = 0]
app(A,B,C) :-
   B=[D|E], E=[F|G], C=[D|H], app(A,E,H).

[Training set performance]
          Actual
        +          -
     + 11         1         12
Pred
     -  0          4          4

       11         5         16

Accuracy = 0.9375
```

Looking at the theory, it learned, that the last element of the resulting list (`C`) is the element, we want to add (`A`), and that the resulting list has also the elements from the second list (`B`).

## 3.3 (c)

For this exercise we decided to implement the sorting algorithm `sort/3`, which allows us to sort in one element (`+any`) into an already sorted list (`+list`) and results in a sorted list with the new element (`+sorted`). This was our implementation of the knowledge file:

```
:- mode(*,sort(+any, +list, +sorted)).
% split list into a new element and new list.
:- mode(1,((+list) = ([-any|-list]))).
% split list up into an element and a new list,
% but do not create a new element (+any).
:- mode(1, ((+sorted) = ([+any|-sorted]))).
% compare all elements with each other.
:- mode(1, ((+any) > (+any))).
:- mode(1, ((+any) < (+any))).

:- set(i,3).
:- set(noise,0).
```

```
:- determination(sort/3,sort/3).
:- determination(sort/3,'='/2).
:- determination(sort/3, '>'/2).
:- determination(sort/3, '<'/2).
```

We added the following positive examples:

```
sort(1, [0], [0,1]).
sort(1, [2], [1,2]).
sort(5, [0], [0,5]).
sort(1, [2], [1,2]).
sort(1, [4], [1,4]).
sort(2, [1], [1,2]).
sort(4, [3], [3,4]).
sort(5, [2,3], [2,3,5]).
sort(4, [1,2], [1,2,4]).
sort(4, [1,5], [1,4,5]).
sort(4, [5,6], [4,5,6]).
sort(1, [2,3], [1,2,3]).
sort(0, [1,2], [0,1,2]).
sort(1, [2,3,4], [1,2,3,4]).
sort(2, [1,3,4], [1,2,3,4]).
sort(4, [5,6,7], [4,5,6,7]).
```

And defined the following negative examples:

```
sort(1,[2],[2,1]).
sort(1,[2,3],[3,2,1]).
sort(2,[1,3],[1,3,2]).
sort(2,[1,3],[2,1,3]).
sort(3,[2,1],[4,5,6]).
sort(2,[3,5],[3,4,5]).
sort(5,[1,2],[1,3,4]).
sort(5,[1,2],[1,3,5]).
sort(1,[2],[4,3,5]).
sort(1,[2,3],[1,4,5]).
sort(3,[1,4],[1,4,3]).
sort(1,[3,4],[3,1,4]).
```

Executing this will result in the following bottom clause and learned theory:

```
[bottom clause]
sort(A,B,C) :-
   B=[D|E], sort(D,E,C), E=[F|G], C=[D|H],
   A>D, D<A, sort(F,B,C), sort(A,E,H),
   H=[F|I], F>D, A>F, F<A,
   D<F.
```

```
[theory]

[Rule 1] [Pos cover = 6 Neg cover = 0]
sort(A,B,C) :-
   B=[D|E], C=[D|F], F=[A|G], A>D.

[Rule 2] [Pos cover = 8 Neg cover = 0]
sort(A,B,C) :-
   B=[D|E], D>A, C=[A|F], F=[D|G].

[Rule 3] [Pos cover = 4 Neg cover = 0]
sort(A,B,C) :-
   B=[D|E], C=[D|F], sort(A,E,F), A>D.

[Training set performance]
          Actual
        +          -
     + 16         2         18
Pred
     -  0         10        10

       16        12        28

Accuracy = 0.9285714285714286
```

With this rules an element can be successfully sorted into a list of 2 elements, resulting in a sorted list of 3 elements. It will also work for more elements, if the element, which should be sorted in, is the largest or second largest number.

# 4 Generalization and Noise Handling

## 4.1 (a)

Let's take a look at the KRK illegality files. An illegal movement is encoded as follows:

```
illegal(5,1,7,5,7,5).
```

The first two entries are for the white king, then the white rook and black king. An illustration can be seen in Figure 1.

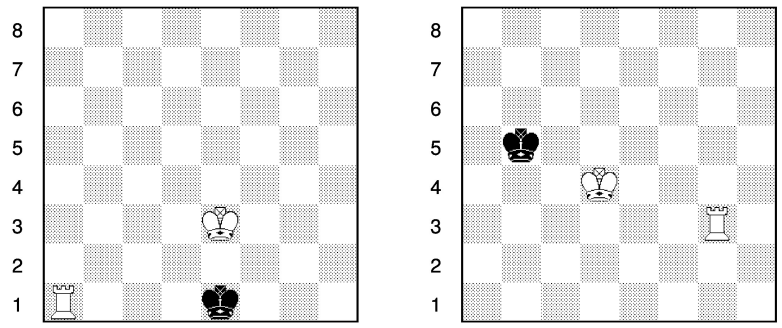


Figure 1: Examples of illegality and corresponding encoding. Figures taken from [1].

As mentioned in the discussion on May 26, we do not want to compare rows with columns and vice-versa; only rows with rows and columns with columns. We thus change the training examples with a quick python script so that they have letters for rows:

```
illegal(b,2,a,5,a,6)
```

Additional background knowledge has to be provided then for the comparisons, for example

```
lt_(a,b).
adj_(e,f).
```

We introduced mode declarations like this:

```
:- modeh(1,illegal(+let,+int,+let,+int,+let,+int)).

:- modeb(1,lt(+int,+int)).
:- modeb(1,adj(+int,+int)).

:- modeb(1,lt_(+let,+let)).
:- modeb(1,adj_(+let,+let)).
```

The head clause should only contain `illegal/6`, while to body clauses can contain the comparisons. The determinations are also standard; Aleph is allowed to argue with the comparisons we provide:

```
:- determination(illegal/6,adj/2).
:- determination(illegal/6,lt/2).
:- determination(illegal/6,adj_/2).
:- determination(illegal/6,lt_/2).
```

## 4.2 (b)

When we run Aleph, we get following results:

```
[Rule 1] [Pos cover = 140 Neg cover = 0]
illegal(A,B,C,D,C,D).

[Rule 2] [Pos cover = 1145 Neg cover = 0]
illegal(A,B,C,D,E,F) :-
   adj(F,B), adj_(E,A).

[Rule 3] [Pos cover = 50 Neg cover = 0]
illegal(A,B,C,B,D,B) :-
   lt_(A,D), lt_(A,C).

[Rule 4] [Pos cover = 524 Neg cover = 0]
illegal(A,B,C,D,C,E) :-
   lt_(A,C).

[Rule 5] [Pos cover = 525 Neg cover = 0]
illegal(A,B,C,D,C,E) :-
   lt_(C,A).

[Rule 6] [Pos cover = 165 Neg cover = 0]
illegal(A,B,A,B,C,D).

[Rule 7] [Pos cover = 514 Neg cover = 0]
illegal(A,B,C,D,E,D) :-
   lt(B,D).

[Rule 8] [Pos cover = 27 Neg cover = 0]
illegal(A,B,A,C,A,D) :-
   lt(B,D), lt(B,C).

[Rule 9] [Pos cover = 549 Neg cover = 0]
illegal(A,B,C,D,E,D) :-
   lt(D,B).

[Rule 10] [Pos cover = 49 Neg cover = 0]
illegal(A,B,C,B,D,B) :-
   lt_(D,A), lt_(C,A).

[Rule 11] [Pos cover = 32 Neg cover = 0]
illegal(A,B,A,C,A,D) :-
   lt(D,B), lt(C,B).
```

Aleph found 11 rules with a training accuracy of 1. On a side note, previously

we weren't distinguishing between rows and columns. The algorithm took longer and when it finished, we ended up with 23 rules.

## 4.3 (c)

In the background knowledge file, we add following lines:

```
:- set(test_pos,'test_new.f').
:- set(test_neg,'test_new.n').
```

Now Aleph knows where the test files are and reports the test error automatically when it's finished with training. For the test error, we get

```
[Test set performance]
             Actual
         +            -
     +  3361           0          3361
Pred
     -   0           6639          6639

        3361          6639         10000

Accuracy = 1
```

## 4.4 (d)

We write a small script in Python to randomly flip the labels.

```
import numpy as np

with open("our_examples/krk_illegal/krk.f") as f:
    pos_examples = [line.rstrip() for line in f]

with open("our_examples/krk_illegal/krk.n") as f:
    neg_examples = [line.rstrip() for line in f]

total_examples = len(pos_examples) + len(neg_examples)

flip_idx = np.random.choice(
    list(range(total_examples)), size=int(total_examples * 0.05)
)

examples = [[x, 1] for x in pos_examples]
for x in neg_examples:
    examples.append([x, 0])
```

```
for idx in flip_idx:
    label = examples[idx][1]
    flipped_label = 1 if label == 0 else 0
    examples[idx][1] = flipped_label

pos_new_examples = [x[0] for x in examples if x[1] == 1]
neg_new_examples = [x[0] for x in examples if x[1] == 0]

with open("our_examples/krk_illegal/krk_flipped.f", "w") as f:
    for line in pos_new_examples:
        f.write(line + "\n")

with open("our_examples/krk_illegal/krk_flipped.n", "w") as f:
    for line in neg_new_examples:
        f.write(line + "\n")
```

## 4.5 (e)

When we run Aleph on the noisy dataset, we get following results:

```
[Training set performance]
             Actual
          +            -
     +  3376           27           3403
Pred
     -   0           6597          6597

         3376          6624         10000

Accuracy = 0.9973
```

The number of rules found were 2563, most of the rules were just a single training example. The algorithm took about 15 minutes to run. For the test set:

```
[Test set performance]
             Actual
          +            -
     +  1049           15           1064
Pred
     -  2312           6624          8936

         3361          6639         10000

Accuracy = 0.7673
```

With the current settings, Aleph was trying too hard to fit to the training set. The test accuracy of 76% could be a lot worse, though. We attribute that to the

multiple generalized rules (i.e. rules with variables in them, not single training examples) that were indeed found.

## 4.6 (f)

Table 1 shows all the configurations that we tried. In the first row there's the configuration from the previous exercise. Apart from the settings mentioned in the homework sheet (`evalfn`, `noise`, `search`), we also consider `clauselength` and `minpos` now.

The next configuration we tried was to allow negative examples to be covered by a clause. So we set `noise` to 200. After about 4 minutes training time we got 311 rules, 6 of which were general ones with variables in them. One rule for example we recognize from before:

```
illegal(A,B,C,D,E,F) :-
   adj(F,B), adj_(E,A).
```

Other rules are a little changed, for example

```
illegal(A,B,C,D,C,E)).
```

Previously that rule also appeared, but `A` had to be less than `C` or `C` less than `A`.

Let's also try to reduce the training time by setting `clauselength` to 3 as opposed to the default of 4. Training time is now 1.4 minutes, so a 75% reduction over the previous training time of 4 minutes. In (b) we obtained literals with a maximum clause length of 3. Aleph was allowed then to construct clauses with a length of 4, but didn't do so. We use that as justification to restrict the clause length to 3 from now on to save time.

Now we should try to reduce the amount of rules because one training example as rule is not very helpful. The setting `minpos` comes in handy there. It specifies a lower bound on the number of positive examples to be covered by an acceptable clause. So if we set it to 2, we should get no more rules that are just one training example. After training we obtained 6 rules, the same ones as before (that had variables in them). The accuracies didn't change at all though! Running the algorithm again in a separate terminal yielded the correct accuracies, they were slightly reduced of course. From now on we will run Aleph always in a new terminal when changing a setting.

The result is already not too bad, but we still tried around some more. Setting the search to `scs` triggered another search algorithm that didn't obey other rules we set, most importantly the clause length. It was trying to look for clauses with 5 or 6 literals. With a search space that big, it took too long and we terminated after 15 minutes. Setting the search to `heuristic` made no difference to the default. So we stick to the default.

For the `evalfn` we tried `mestimate` because it comes from a paper called "Handling Noise in Inductive Logic Programming" [2]. After 1.4 minutes we obtained 19 rules with a test accuracy of 1.

| | Params | Training accuracy | Test accuracy | Num rules | Training time |
|---|---|---|---|---|---|
| 1 | evalfn: coverage, noise: 0, search: bf | 0.9973 | 0.7673 | 2563 | 15 min |
| 2 | Same as (1), but noise: 200 | 0.9774 | 0.9953 | 311 | 4 min |
| 3 | Same as (2), but clauselength: 3 | 0.9774 | 0.9953 | 311 | 1.4 min |
| 4 | Same as (3), but minpos: 2 | 0.9488 | 0.9968 | 6 | 1.4 min |
| 5 | Same as (4), but search: scs | - | - | - | >15 min |
| 6 | Same as (4), but search: heuristic | 0.9488 | 0.9968 | 6 | 1.4 min |
| 7 | Same as (4), but evalfn: mestimate | 0.9524 | 1 | 19 | 1.4 min |

Table 1: Hyperparameter search for the noisy data set.

# References


[1] Stephen H. Muggleton. Learning rules from chess databases. `https://www.doc.ic.ac.uk/~shm/chess.html`.

[2] Saso Dzeroski and Ivan Bratko. Handling noise in inductive logic programming. In *Proceedings of the International Workshop on Inductive Logic Programming*, volume 91, 1992.