6/5/2021

Luka Leko, Paul Knoll

Symbolic AI: Task 2

Group 01:

Contents

[1. Getting Started 2](#_Toc73885992)

[Basic Aleph File Structure 2](#_Toc73885993)

[2. Recursion 5](#_Toc73885994)

[(a) Learn the member predicate: 5](#_Toc73885995)

# Getting Started

We first tried to understand what Aleph does by testing it out. For this we installed it as instructed in the task specification and then executed Aleph on the Train example from Ryszard Michalski. This is what we have learned:

## Basic Aleph File Structure

The whole Aleph Logic is contained within a single .pl file, which we will call aleph-swi.pl. For Aleph to be able to induce a set of rules (also called a theory), it needs the following three files contained within the same folder:

1. The .f file: It contains the set of positive examples. The examples are formatted as simple prolog facts. Aleph will consider those as correct during his learning procedure:

.f file for the Train example:

**eastbound(east1).**

**eastbound(east2).**

**eastbound(east3).**

**eastbound(east4).**

**eastbound(east5).**

1. The .n file: It contains the set of negative examples. It looks the same as .f file, except that Aleph will consider those facts as wrong during his learning procedure.
2. The .b file: This file is the most interesting out of the three. The Background knowledge is in the form of Prolog clauses that encode information relevant to the domain. This file also contains language and search restrictions for Aleph. Amongst other things, this file mostly contains:
   1. Modes. These declare the mode of call for predicates that can appear in any clause hypothesised by Aleph.
   2. Determinations. Determination statements declare the predicates that can be used to construct a hypothesis.
   3. Type definitions.
   4. You can also set certain variables used by Aleph with set(variableName, value).

When one wants to run Aleph to learn rules to distinguish between positive and negative examples, one has to:

1. Open swipl,
2. Consult the aleph-swi.pl file.
3. Read\_all(filename). This will make Aleph load the example.
4. Call Induce. This will initiate the learning step. Based on the configuration within the Background Knowledge file, this will produce a certain set of rules. During the learning, Aleph will also print out the theory that he has learned, including a matrix stating how many of the positive and negative examples one can correctly identify given that theory. Used on the trains example, Aleph is able to find a theory consisting of only a single rule which is able to correctly predict every positive example and exclude every negative example:

[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

The solution that we have found by hand to distinguish between eastbound trains and westbound trains looks as follows:

**eastbound(A) :-**

**has\_car(A, B), has\_car(A, C), has\_car(A, D), shape(C, triangle, 1).**

With this we can correctly predict all positive examples as correct, and only wrongly predict one negative example as correct.

We can also look at the bottom clauses, i.e. the clauses that contain all literals that we have allowed through the background knowledge. You can do so by calling **show(bottom)**. The bottom clause for the first example looks as follows:

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).

We can see that it does indeed contain all literals that were also included in the rule that Aleph has found during learning.

# Recursion

## 2.1. Learn the member predicate:

As always, the files needed for Aleph are the Background Knowledge file (mem.b), the positive examples (mem.f) and the negative examples (mem.n). The example files obviously contain only examples, so only the .b file is really interesting. It will now be described in more detail.

### The mem/2 Background Knowledge File

The Background Knowledge File for the mem/2 predicate looks as follows:

% Simple illustration of the learning of recursive predicates

% in Aleph

% To run do the following:

% a. Load Aleph

% b. read\_all(mem).

% c. induce.

:- modeh(\*,mem(+any,+list)).

:- modeb(\*,mem(+any,+list)).

:- modeb(1,((+list) = ([-any|-list]))).

:- set(i,3).

:- set(noise,0).

:- determination(mem/2,mem/2).

:- determination(mem/2,'='/2).

:- modeh(\*,mem(+any,+list))

says that in the head of a clause you only allow the mem() predicate with 2 parameters, the first parameter of type any and the second parameter of type list.

:- modeb(\*,mem(+any,+list))

allows that same structure to occur as a literal in the body of a clause.

:- modeb(1,((+list) = ([-any|-list])))

Allows another literal format to appear in the body of a clause, namely that a list equals another list where the head is of type any and the tail is another list.

With

:- set(variableName, value)

One can set variables for Aleph, which can steer how Aleph learns during Induction. For example,

:- set(noise, 0)

sets an upper bound on the number of negative examples allowed to be covered by an acceptable clause.

The last part in the member Background Knowledge are the determinations, with which one selects what predicates are allowed in what clauses in general. For example,

:- determination(mem/2,mem/2)

allows mem/2 predicates to appear in clauses where mem/2 is in the head.

### Executing Aleph on the mem files

With this very restricted search, Aleph is able to find the following implementation of the mem/2 predicate in Prolog:

[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

The mem/2 predicate it has found is indeed a correct implementation of the member predicate. Obviously, because of that, it is also able to predict all example correctly.

## Learn the last/2 predicate

In order to have Aleph learn the last/2 predicate, one has to provide Aleph with the three files as always. The two example files are trivial to construct, as one only has to think of a couple of examples for where the predicate is positive and where it is negative. One only has to watch out to also cover the base case in both files at least once.

To know how the Background Knowledge file should look like, we first defined the last/2 predicate ourselves, and then we looked at what kinds of predicates we needed there. The idea is that if we know one possible definition of the predicate, then we only have to make sure that Aleph is able to construct that particular clause, and then it will be able to find it. This was our recursive definition of the last/2 predicate:

mem(A, [A|\_]).

mem(A, [H|T]) :- mem(A, T).

Fortunately for us, this recursive definition is already very close to the recursive definition of the mem/2 predicate, for which we already have the Background Knowledge File. With that in mind, we created the following Background Knowledge File:

:- modeh(\*,last(+any,+list)).

:- modeb(\*,last(+any,+list)).

:- modeb(1,((+list) = ([-any]))).

:- modeb(1,((+list) = ([-any\_2|-list]))).

:- set(i,3).

:- set(noise,0).

:- determination(last/2,last/2).

:- determination(last/2,'='/2).

With this slight modification, Aleph was able to find the following clauses:

[theory]

[Rule 1] [Pos cover = 5 Neg cover = 0]

last(A,B) :-

B=[A].

[Rule 2] [Pos cover = 11 Neg cover = 0]

last(A,B) :-

B=[C|D], last(A,D).

[Training set performance]

Actual

+ -

+ 16 0 16

Pred

- 0 13 13

16 13 29

It has found our definition of the last/2 predicate. It does not look exactly the same, but when evaluated by the Prolog Interpreter it is equivalent to our definition.

# Generalization

This exercise was about inducing a set of clauses which can distinguish between good and bad training examples in the dataset that we were provided with. The dataset has the following shape:

First 10 rows of the training\_dataset.csv:

A13,18,A32,A42,A61,A73,2,A93,A101,A121,A143,A152,A172,2,A191,A201,Bad

A11,12,A32,A40,A61,A72,4,A93,A101,A121,A143,A152,A172,2,A191,A201,Bad

A14,18,A32,A42,A61,A72,4,A94,A102,A122,A143,A152,A172,2,A191,A201,Bad

A12,18,A34,A42,A61,A72,2,A93,A101,A121,A143,A152,A172,1,A191,A201,Bad

A11,18,A32,A46,A61,A71,4,A92,A101,A121,A143,A152,A171,1,A191,A201,Bad

As one can see, this is not yet in the correct format for Aleph to be able to process those examples. With the script Task4\_Script.ipynb we converted this data into the correct format, spread out over the three files generalization.b, generalization.f and generalization.n. Because the script is not relevant for Aleph, it is omitted at this point.

First 5 rows of generalization.f:

generalization(a13,21,a32,a40,a62,a73,1,a92,a101,a123,a141,a152,a174,1,a192,a201).

generalization(a11,12,a34,a40,a61,a73,3,a93,a101,a121,a143,a151,a172,2,a191,a202).

generalization(a14,9,a34,a43,a65,a73,1,a93,a101,a121,a143,a152,a173,2,a191,a201).

generalization(a14,24,a34,a42,a64,a73,4,a93,a101,a121,a143,a152,a173,1,a192,a201).

generalization(a14,24,a32,a42,a61,a74,2,a92,a101,a122,a143,a151,a173,1,a192,a201).

We can see that one example consists of 16 features, three of which are numerical features, and the other 13 are categorical features. For the categorical features, only the equality predicate really makes sense to compare two values that belong to the same feature. For the numerical features, the predicates “≤”, “≥”, “<”, “>” are also plausible. Comparing between two different features obviously does not make sense. Because of that, we defined the Background Knowledge the following way: