Machine learning: Inductive Logic Programming

This is a way of learning new code from existing code together with positive and negative examples of what the new code is supposed to do.

What is Inductive Logic Programming?

• Approach to machine learning where definitions of relations are induced from positive and negative examples.

• Logic is used as a hypothesis language.

• Result is a Prolog program.

• Initial hypothesis is refined to find a target hypothesis.

Example – a definition of bull/1?

Given some predicates:

- male(joe).
- sheep(joe).
- male(buster).
- cattle(buster).
- female(daisy).
- cattle(daisy).
- female(dolly).
- sheep(dolly).

and positive and negative examples:

- positive(bull(buster)).
- negative(bull(joe)).
- negative(bull(daisy)).

More formally ...

Given
- a set of positive examples (Pos_Ex)
- a set of negative examples (Neg_Ex)
- background knowledge (BK) – i.e. the predicates that are given by the programmer

Find an hypothesis \( H \) where
- all examples in Pos_Ex can be derived from BK and \( H \)
- no example in Neg_Ex can be derived from BK and \( H \)

Qualities of the hypothesis

A hypothesis is a Prolog clause (i.e. a rule) – for instance:

\[
\text{bull(Arg)} :- \text{male(Arg), cattle(Arg)}.\]

The hypothesis must be
- Complete – covering all positive examples;
- Consistent – not covering any negative examples.
Another example - 1

The rest of the lecture builds on the Mini-Hyper code in Bratko.

Task: Find a definition of a predicate has_daughter/1 (the target predicate)

using the background knowledge predicates:
parent/2, male/1, female/1

that is complete and consistent.

Another example - 2

The example is based on the Tudor family tree.

We already have some facts in the program database:
parent(ann_boleyn, elizabeth).
parent(catherine_of_aragon, mary).
parent(elizabeth_york, henry8).
parent(henry7, henry8).
parent(henry8, edward).
parent(henry8, elizabeth).
parent(henry8, mary).
parent(jane, edward).

Another example - 3

We already have some facts in the program database:
female(ann_boleyn).
female(catherine_of_aragon).
female(elizabeth).
female(elizabeth_york).
female(jane).
female(mary).

male(edward).
male(henry7).
male(henry8).

Another example - 4

The positive examples are:
ex(has_daughter(henry8)).
ex(has_daughter(catherine_of_aragon)).

The negative examples are:
 nex(has_daughter(henry7)).
 nex(has_daughter(elizabeth_york)).
 nex(has_daughter(jane)).

Another example - 5

We are allowed to use these background predicates:
backliteral(male(Arg), [Arg]).

backliteral(female(Arg), [Arg]).
backliteral(parent(Arg1, Arg2), [Arg1, Arg2]).

This is a list of the variables in the arguments.

Another example - 6

Using this knowledge and the initial hypothesis:
has_daughter(Arg)

we will find the rule:
has_daughter(Arg1) :-
parent(Arg1, Arg2), female(Arg2).

but stored in the form [Head|Subgoals]:
[has_daughter(_A), parent(_A, _B),
female(_B)],[_A,_B]
How do we get from hypothesis to rule? (Refinement)

The hypothesis is refined in small stages.

We start with an overly general hypothesis and gradually make it more specific.

Each new hypothesis must be more specific than its predecessor – so the new hypothesis covers a subset of what the previous hypothesis covers.

Methods of refinement - 1

Unifying two variables in the clause

Example:
The hypothesis:

\[
[\text{has
daughter}(X),\text{parent}(Y, Z)] \land [X, Y, Z]
\]

or

\[
\text{has
daughter}(X) :\neg \text{parent}(Y, Z)
\]

is refined to:

\[
\text{has
daughter}(X) :\neg \text{parent}(X, Z)
\]

Methods of refinement - 2

Adding a sub-goal (background literal) to the body of the clause

Example:
The hypothesis:

\[
[\text{has
daughter}(X),\text{parent}(Y, Z)] \land [X, Y, Z]
\]

or

\[
\text{has
daughter}(X) :\neg \text{parent}(Y, Z)
\]

is refined to:

\[
\text{has
daughter}(X) :\neg \text{parent}(Y, Z),\text{female}(V)
\]

Overview of the process

Testing completeness and consistency

Recall:
The hypothesis must be

– Complete – covering all positive examples;
– Consistent – not covering any negative examples.

We need a way of testing (proving) our hypothesis against the positive and negative examples.
Proving – the idea

We have a goal and a hypothesis.

We simulate a Prolog proof mechanism – but with two differences:
- depth-limit: search cannot go into infinite search
- three outcomes:
  1. true (yes);
  2. false (no);
  3. maybe (i.e. reached maximum depth before deciding true or false)

Top-level rule:
prove(Goal, Hypothesis, Answer) :-
  max_proof_length(Max_Depth),
  prove(Goal, Hypothesis, Max_Depth, Remaining_Depth),
  evaluate_learning(Remaining_Depth, Answer).
prove(_, _, no).

Evaluating the outcome:
% 1 - hypothesis proved
evaluate_learning(Depth_of_Search, yes) :-
  Depth_of_Search >= 0,
  !.     % green cut
% 2 - hypothesis not proved nor disproved
evaluate_learning(Depth_of_Search, maybe) :-
  Depth_of_Search < 0.

Proving - terminations:
% 1 - maximum depth of proof exceeded
prove(Goal, Hypothesis, Depth, Depth) :-
  Depth < 0,
  !.
% 2 - no more sub-goals in body
prove([], Hypothesis, Depth, Depth) :-
  !.

Proving – conjunctions of subgoals:
% 3 - Goal is a conjunction (i.e. from a % body) so prove each in turn
prove([Goal|Goals], Hypothesis, Depth0, Depth) :-
  !,
  prove(Goal, Hypothesis, Depth0, Depth1),
  prove(Goals, Hypothesis, Depth1, Depth).

Proving – predicate or hit maximum depth:
prove(Goal, Hypothesis, Depth, Depth) :-
  prolog_predicate(Goal),
  call(Goal).
prove(Goal, Hypothesis, Max_Depth, New_Max_Depth) :-
  Max_Depth =:= 0,
  !,
  New_Max_Depth is Max_Depth - 1.
Proving - 6

Proving – move deeper:
prove(Goal, Hypothesis, Max_Depth, New_Max_Depth) :-
    Max_Depth > 0,
    Max_Depth1 is Max_Depth - 1,
    member(Clause*Arguments, Hypothesis),
    copy_term(Clause, [Head | Body]),
    Goal = Head,
    prove(Body, Hypothesis, Max_Depth1, New_Max_Depth).

Overview of the process

Testing completeness

complete(Hypothesis) :-
    \+ ( ex(Positive_Example),
        once(prove(Positive_Example, Hypothesis, Answer)),
        % the answer should be "no"
        % or "maybe"
        Answer \== yes
    ).

Or to put it another way: if there is any query that isn’t true that should be true, we’ve got a problem.

Testing consistency

consistent(Hypothesis) :-
    \+ ( nlex(Negative_Example),
        once(prove(Negative_Example, Hypothesis, Answer)),
        % the answer should be "yes"
        % or "maybe"
        Answer \== no
    ).

Or to put it another way: if there is any query that is or may be true that should be false, we’ve got a problem.

Generate and test - again

This program has a very familiar structure.

Like the Sudoku solver, it:
- generates a potential solution (by refining a hypothesis)
- tests the solution for completeness and consistency.

The rest of the program - 1

Start things going by finding the maximum depth and search …

induce(Target_Hypothesis) :-
    max_proof_length(Depth_Threshold),
    iter_deep(Target_Hypothesis, 0, Depth_Threshold).
The rest of the program - 2

Search at this level ...

\[
\text{iter\_deep}(\text{Hypothesis}, \text{Maximum\_Depth}, \\
\text{Depth\_Threshold}) : - \\
\text{Maximum\_Depth} \leq \text{Depth\_Threshold}, \\
\text{initial\_hypothesis}(\text{Hypothesis0}), \\
\text{complete}(\text{Hypothesis0}), \\
\text{depth\_first}(\text{Hypothesis0}, \text{Hypothesis}, \\
\text{Depth\_Threshold}).
\]

The rest of the program - 3

or go deeper...

\[
\text{iter\_deep}(\text{Hypothesis}, \text{Maximum\_Depth}, \\
\text{Depth\_Threshold}) : - \\
\text{Maximum\_Depth} < \text{Depth\_Threshold}, \\
\text{New\_Maximum\_Depth} \text{ is } \\
\text{Maximum\_Depth} + 1, \\
\text{iter\_deep}(\text{Hypothesis}, \\
\text{New\_Maximum\_Depth}, \\
\text{Depth\_Threshold}).
\]

The rest of the program - 4

And search for a refinement...

% 1 - terminating
\[
\text{depth\_first}(\text{Hypothesis0}, \text{Hypothesis}, \\
\text{Maximum\_Depth}) : - \\
\text{Maximum\_Depth} \geq 0, \\
\text{consistent}(\text{Hypothesis}).
\]

Refining by unifying variables

\[
\text{refine}(\text{Clause}, \text{Args}, \text{Clause}, \text{New\_Args}) : - \\
\% \text{ non-deterministically select arg} \\
\text{append}(\text{Args1}, [\text{Argument}|\text{Args2}], \\
\text{Clause}, \text{New\_Args}). \\
\% \text{ unify Argument with another} \\
\text{member(Argument, Args2)}, \\
\% \text{ recombine the arguments without} \\
\% \text{ duplicate args} \\
\text{append}(\text{Args1}, \text{Arguments2}, \\
\text{New\_Arguments}).
\]
Refining by adding a subgoal

\[
\text{refine}(	ext{Clause}, \text{Args}, \text{New-Clause}, \text{New-Args}) : - \\
\text{length}(	ext{Clause}, \text{No-of-Lits}), \\
\text{max_clause_length}(	ext{Max-No-of-Lits}), \\
\text{No-of-Lits} < \text{Max-No-of-Lits}, \\
% \text{select a background literal} \\
\text{backliteral}(	ext{Lit}, \text{Args-of-Lit}), \\
% \text{add this literal to body of clause} \\
\text{append}(	ext{Clause}, \text{[Lit]}, \text{New-Clause}), \\
% \text{add arguments of the literal to} \\
\text{the arguments of the clause} \\
\text{append}(	ext{Args}, \text{Args-of-Lit}, \text{New-Args}).
\]

How good is this system?

The ability to learn new code is very impressive.

This ILP system is very basic:
- it won’t learn more complex arguments (e.g. \{Head|Tail\});
- it generates many hypothesis when searching for code for more complicated problems.

But there are ways of making it more efficient – for instance, by adding types to arguments and using best-first search with an heuristic.