In this chapter we look at two aspects of machine learning (ML). In order to implement the first example of ML, *explanation based generalisation*, we first need to consider meta-programming in Prolog. The second example of ML is learning simple if-then rules.

### 5.1 Meta Interpreters

Meta-programs are programs written to manipulate other programs. Prolog is ideal for writing meta programs as Prolog programs are just Prolog terms themselves. A meta-interpreter is an interpreter written in its own language. Prolog and Lisp are languages in which it is easy to write meta-interpreters — writing them for conventional procedural languages is much more difficult. The simplest meta-interpreter for Prolog is:

```
meta(Goal) :- Goal.
```

This isn’t that useful, but see an example in *Art of Prolog* for a shell and logging program. We now look at some more useful ones. The first is `solve` which simulates the normal Prolog interpreter (at least for pure Prolog). It makes use of the built-in meta-predicate `clause(Head,Body)` which, assuming `Head` is instantiated, finds a clause matching the head `Head` and then instantiates `Body` to the body of that clause.

```
solve(true).
solve((A,B)) :-
    solve(A),
    solve(B).
solve(A) :-
    clause(A,B), % find clause with head matching A
    solve(B).
```

Now this basic interpreter isn’t that much use as it stands, but it can be used as a starting point to develop much more useful meta-interpreters. What we normally want to do is solve a goal and also perform some other task or modify the basic search strategy.
We will give two simple examples of this. But first let us see how we can easily extend \texttt{solve} so that it will deal correctly with system predicates e.g. \texttt{nl}.

\begin{verbatim}
solve_with_sys(true).
solve_with_sys((A,B)) :-
    solve_with_sys(A),
    solve_with_sys(B).
solve_with_sys(A) :- system(A),A.
solve_with_sys(A) :-
    clause(A,B),
    solve_with_sys(B).
\end{verbatim}

We see that all that is necessary is to add the extra clause testing to see if the goal is a built in predicate. This uses \texttt{system(A)} which is true if \texttt{A} is instantiated to a system predicate.

Now suppose we want to solve a goal, but at the same time construct a proof of the goal if it succeeds. This is \texttt{solve_with_proof(Goal,Proof)} which given a goal \texttt{Goal} which has a solution instantiates \texttt{Proof} to a proof of this. As an example, if we have in the database:

\begin{verbatim}
son_of(Par,Son) :-
    child_of(Par,Son),
    boy(Son).
child_of(mary,bill).
boy(bill).
\end{verbatim}

and we launch the goal:

\begin{verbatim}
?- solve_with_proof(son_of(mary,Son),Proof).
\end{verbatim}

then we obtain

\begin{verbatim}
Proof = ( son_of(mary,bill) :-
            ( ( child_of(mary,bill) :- true),
              ( boy(bill) :- true)) )
\end{verbatim}

Here is the code for \texttt{solve_with_proof}
solve_with_proof(true,true).

solve_with_proof((A,B),(PfA,PfB)) :-
    solve_with_proof(A,PfA),
    solve_with_proof(B,PfB).

solve_with_proof(A,(A:-PfB) :-
    clause(A,B),
    solve_with_proof(B,PfB).

We could alter solve_with_proof so that we don’t return the true’s at the leaves of the tree but have assertions there, by just inserting as the third clause

solve_with_proof(A,A) :-
    clause(A,true).

As a last example, we introduce solve_with_d_bound(Goal,Depth,Res) which is used to see if there is a proof of Goal with tree depth \( \leq \) Depth if so Res is true else overflow.

solv_d_bound(true,Depth,true):-
    !,Depth>0.

solve_d_bound(Goal,0,overflow):-!.

solve_d_bound((A,B),Depth,Res) :- solve_d_bound(A,Depth,ResA),
    (ResA == true ->
     solve_d_bound(B,Depth,Res);
     Res = overflow).

solve_d_bound(A,Depth,Res) :-
    clause(A,B),
    D1 is Depth-1,
    solve_d_bound(B,D1,Res).

As some trees for a proof may be longer than Depth to see if there exists at least one such tree we can use solve_any(G,D,R) which will continue calling solve_with_d_bound if overflow is obtained until either Res becomes true or no such proof exists in which case overflow is returned.

solve_any(G,D,R) :-
    solve_d_bound(G,D,R),
    R\== overflow.

solve_any(G,D,overflow).

Now there are many more uses of meta–interpreters. See Art of Prolog for a good selection. Some uses are:

a) Obtaining traces of proofs as the solution of a goal is sought.
b) Asking the user to supply information if a goal cannot be proved.

c) Generating explanations for expert system shells written in Prolog.

d) Interactive shells with why explanations.

e) Reasoning with uncertainty.

f) Explanation based generalisation. A proof is generated for a training example and at the same time a more general proof of the more general concept is created.

g) Shapiro in his thesis used meta-interpreters for algorithmic debugging of Prolog programs.

5.2 Explanation based generalisation

Explanation based generalisation (EBG) is a technique to obtain general concepts from a training example. When we reach a conclusion, if it is correct, then by tracing through the reasoning process we may be able to identify what exactly contributes to the conclusion and so ignore superfluous information. From the particular case we may then be able to generalise the conclusion (and reasoning).

If the conclusion turns out to be false, the reasoning may provide information as to where the problem lies and possibly how to correct it. This technique is called failure-based learning and was used extensively in Shapiro’s work on algorithmic debugging. We however shall not be deal with this.

EBG consists of 2 stages:

i) Analyse a single training example in terms of some domain together with a goal concept and produce an explanation structure separating relevant features from irrelevant — why example is an instance of the goal.

ii) analyse explanation structure to determine constraints sufficient for explanation to apply in general — generalise the explanation.

We can consider the EBG input to comprise of:

gold concept The concept to be learned — realised as a predicate.

domain theory Set of rules — used in explaining why the example is instance of goal concept.

training example Set of facts describing a particular situation and an instance of the goal concept (the training goal).

operational criteria The predicates in which learned concept must be expressed — easily evaluated predicates from domain theory.

We first consider an example to clarify the concepts.
5.2.1 Example

We are trying to determine conditions under which it is safe to place one object on top
of another in a particular domain of objects.

**goal concept**
safe_to_stack(X,Y)

**training example**

on(box1,table1).
vol(box1,10).
box(box1).
table(table1).
colour(box1,red).
colour(table1,blue).
density(box1,10).

domain theory

safe_to_stack(X,Y) :-
    lighter(X,Y).

lighter(X,Y) :-
    wt(X,W1),
    wt(Y,W2),
    less_than(W1,W2).

wt(X,500) :-
    table(X).
wt(X,Y) :-
    vol(X,V),
    density(X,D),
    times(V,D,Y).

**operational criterion**

explanation to be given using concepts:

[times, less_than, on, vol, box, table, colour, density]

By constructing a proof for the training goal safe_to_stack(box1,table1) (Figure 5.1)
and generalising we are lead to the new ‘rule’
safe_to_stack(X,Y) :-
    vol(X,VX), density(X,DX),
    times(VX,DX,MX), table(Y),
    less_than(MX,500).

Figure 5.1: Proof Tree of Training Goal

with the obvious interpretation — it is safe to stack X on Y if Y is a table and the volume of X times its density is less than 500.

Alternatively from the proof tree for the training goal we can extract the full proof tree, Figure 5.2, giving an explanation for the goal concept:

Figure 5.2: Proof Tree of Generalised Goal
In Prolog we can produce the generalised proof by writing a meta-interpreter. We first recall the meta-interpreter for building a proof tree where built in predicates are evaluated

\[
\text{prove}(A,[A]) :- \\
\text{built-in}(A), A.
\]

\[
\text{prove}(A,[A]) :- \\
\text{clause}(A,\text{true}).
\]

\[
\text{prove}((A,B),Pf) :- \\
\text{prove}(A,PfA), \\
\text{prove}(B,PfB), \\
\text{append}(PfA,PfB,Pf).
\]

\[
\text{prove}(A,[(A :- PfB)]) :- \\
\text{clause}(A,B), \\
\text{prove}(B,PfB).
\]

Now for \texttt{ebg} we generalise the goal and proof by using a modified meta-interpreter [Kedar-Cabelli & McCarty].

\[
\text{ebg}(\text{Goal},\text{GenGoal},\text{Proof},\text{GenProof}) \text{ takes training goal } \text{Goal} \text{ and generalised goal } \text{GenGoal} \text{ and produces both a proof of } \text{Goal} \text{ and a generalised proof of } \text{GenGoal}.
\]

\[
\text{ebg}(A,\text{GenA},[A],[\text{GenA}]) :- \% \text{ to prevent going below leaves}
\text{leaf}(A),!,A.
\]

\[
\text{ebg}((A,B),(\text{GenA},\text{GenB}),Pf,\text{GenPf}) :- \\
!,\text{ebg}(A,\text{GenA},PfA,\text{GenPfA}), \\
\text{ebg}(B,\text{GenB},PfB,\text{GenPfB}), \\
\text{append}(PfA,PfB,Pf), \\
\text{append}(\text{GenPfA},\text{GenPfB},\text{GenPf}).
\]

\[
\text{ebg}(A,\text{GenA},[A],[\text{GenA}]) :- \\
\text{clause}(A,\text{true}),!.
\]

\[
\text{ebg}(A,\text{GenA},[(A :- PfB)],[\text{GenA} :- \text{GenPfB}]) :- \\
\text{clause}(\text{GenA},\text{GenB}), \\
\text{copy}((\text{GenA} :- \text{GenB}),(A :- B)), \\
\text{ebg}(B,\text{GenB},PfB,\text{GenPfB}).
\]

\text{copy}(\text{Tm},\text{CTm}) \text{ produces an exact copy of } \text{Tm} \text{ but with the variables all replaced by new variable names. A simple way of implementing this if it is not supplied is:}
copy(Tm,CTm) :-
    assert(dummy(Tm)),
    retract(dummy(CTm)).

The purpose of copy is so that the generalised tree exactly mirrors the specific tree by ensuring that (A:-B) is an exact copy of (GenA:-GenB). Matching is done on the copy so that variables in the general goal did not become incorrectly instantiated.

Now consider our previous example:

?- ebg(safe_to_stack(box1,table1),safe_to_stack(X,Y),Pf,GenPf).

will yield the following proofs:

Pf = [(safe_to_stack(box1,table1) :-
    [(lighter(box1,table1) :-
        [(wt(box1,W1) :-
            [vol(box1,10),density(box1,10),
              times(10,10,100)],
          (wt(table1,500) :- [table(table1)]),
          less_than(100,500)])])]

and the generalised proof:

GenPf = [(safe_to_stack(B,T) :-
    [(lighter(B,T) :-
        [(wt(B,W1) :-
            [vol(B,V),density(B,D),times(V,D,W1)],
          (wt(T,500) :- [table(T)]),
          less_than(W1,500)])])]

In the above times and the predicate less_than will be defined as leaves and will be evaluated by introducing their obvious definitions using is and <.
5.2.2 Example

goal concept: kills(john,john) training goal

generalised goal: kills(X,Y)

domain theory:
kills(A,B) :-
    hate(A,B),
    possess(A,W),
    weapon(W).

hate(M,M) :- depressed(M).

possess(U,V) :- buy(U,V).

weapon(X) :- gun(X).

training example:
depressed(john).
buy(john,gun1).
gun(gun1).

?- ebg(kills(john,john),kills(X,Y),Pf,GenPf).

Pf :
[(kills(john,john) :-
    [(hate(john,john) :- [depressed(john)])]
)]

GenPf :
[(kills(X,X) :-
    [(hate(X,X) :- [depressed(X)])]
)]
Now we may only want the conditions for the goal to be provable, in the above two examples:

```prolog
safe_to_stack(X,Y) :-
    vol(X,VX),density(X,DX),times(XV,DX,MX),
    table(Y),less_than(MX,500).
```

and

```prolog
kills(X,X) :-
    depressed(X),buy(X,G),gun(G).
```

“Some one will kill themselves if they have bought a gun and are depressed”. this can be extracted by the simpler program:

```prolog
ebg2(Leaf,Genleaf,GenLeaf) :- leaf(Leaf),!,Leaf.

ebg2((G1.G2),(GenG1,GenG2),(Leaves1,Leaves2)) :-
    !,ebg2(G1,GenG1,Leaves1),
    ebg2(G2,GenG2,Leaves2).

ebg2(G,GenG,GenG):-
    clause(G,true),!.

ebg2(G,GenG,Leaves):-
    clause(GenG,GenBody),
    copy((GenG:-GenBody),(G:-B)),
    ebg(B,GenBody,Leaves).
```

`ebg2` is similar to `ebg` but only saves the leaves of the generalised proof tree as a conjunct of goals.

So `?- ebg2(kills(john,john),kills(X,Y),L)` will return

```prolog
X = Y,
L = (depressed(X),buy(X,G),gun(G))
```

and for `?- ebg2(safe_to_stack(box1,table1),safe_to_stack(X,Y),L)` will return the rule obtained earlier informally.
5.2.3 Example

training goal:  \texttt{prescribes(spock,john,asprin)}

generalised goal:  \texttt{prescribes(Dr,Pat,Drug)}

daomain theory:

\texttt{prescribes(D,P,Drug) :-}
\hspace{1cm}  \texttt{doctor(D),}
\hspace{1cm}  \texttt{symptom(P,S),}
\hspace{1cm}  \texttt{cures(Drug,S).}

\texttt{symptom(P,headache) :- illness(P,flu).}
\texttt{symptom(P,sick) :- illness(P,flu).}

\texttt{cures(Drug,headache) :- analgesic(Drug).}

training example:

\texttt{doctor(spock).}
\texttt{analgesic(asprin).}
\texttt{illness(john,flu).}

operational criteria:

\texttt{leaf(doctor(_)).}
\texttt{leaf(analgesic(_)).}
\texttt{leaf(illness(_,_)).}

This will produce the generalised rule

\texttt{prescribes(Dr,Pat,Drug) :-}
\hspace{1cm}  \texttt{doctor(Dr),}
\hspace{1cm}  \texttt{illness(Pat,flu),}
\hspace{1cm}  \texttt{analgesic(Drug).}

If the operational criteria included \texttt{cures} instead of \texttt{analgesic} then we would obtain:
prescribes(Dr,Pat,Drug) :-
    doctor(Dr),
    illness(Pat,flu),
    cures(Drug, headache).

5.2.4 Remark

van Harmelen and Bundy have shown that ebfg is essentially a form of partial evaluation or unfolding (replace procedure calls by their bodies). The training example guides the unfolding and the operational criteria determine how far the unfolding should go. For details see AI journal.

5.3 Learning if-then Rules

Consider the silhouettes shapes shown in Figure 5.3. There are different classes of shapes: bolt, nut, keys etc. We would like to learn rules from the example to describe
these classes of shapes. We consider that the classes are described by values of the following attributes:

size: small, large

shape: long, compact, other

holes: none, 1, 2, 3, many.

Each example from the sample can be described as a Prolog fact of the form

`example(Class, [Attribute1 = Val1, Attribute2 = Val2, ...])`

where the complete set of twelve objects is:

```prolog
attribute(size, [small, large])
attribute(shape, [long, compact, other])
attribute(holes, [none, 1, 2, 3, many])
exa(mple(nut, [size = small, shape = compact, holes = 1])
exa(mple(bolt, [size = small, shape = long, holes = none])
exa(mple(key, [size = small, shape = long, holes = 1])
exa(mple(nut, [size = small, shape = compact, holes = 1])
exa(mple(key, [size = large, shape = long, holes = 1])
exa(mple(bolt, [size = small, shape = compact, holes = none])
exa(mple(nut, [size = small, shape = compact, holes = 1])
exa(mple(pencil, [size = large, shape = compact, holes = none])
exa(mple(scissors, [size = large, shape = compact, holes = 2])
exa(mple(pencil, [size = large, shape = long, holes = none])
exa(mple(scissors, [size = large, shape = other, holes = 2])
exa(mple(key, [size = large, shape = other, holes = 2])
```

We can describe different classes by rules as illustrated below for `nut` and `key`:

```
nut <= [ [size = small, holes = 1]]
key <= [ [shape = long, holes = 1], [shape = other, holes = 2] ]
```

The intended meaning of these rules are

An object is a nut if
its size is small AND has one hole

An object is a key if
shape is long AND has 1 hole OR
shape is other AND has 2 holes

The general format for a rule is
Class \( \Leftrightarrow [\text{Conj}_1, \text{Conj}_2, \ldots \text{Conj}_n] \)

where \( \text{Conj}_i \) are lists of attributes of the form:

\[
[\text{Att}_1 = \text{Val}_1, \text{Att}_2 = \text{Val}_2, \ldots].
\]

An object will be in a class if it satisfies one of the \( \text{Conj}_i \) and the conjunctions are satisfied if all the common attributes have the indicated values.

### 5.3.1 Inducing Rules

Given a set of examples, we want to construct class descriptions which match exactly the examples in the class — the description is satisfied by all the examples in the class and other examples don’t satisfy the description.

A description covers an object which matches it, so the aim is to form descriptions which cover all examples in a class and no others. In Prolog, we can test for an object matching a description as follows:

\[
\text{match}(\text{Obj}, \text{Descrip}) :- \\
\quad \text{member}(\text{Conj}, \text{Descrip}), \\
\quad \text{satisfy}(\text{Obj}, \text{Conj}).
\]

\[
\text{satisfy}(\text{Obj}, \text{Conj}) :- \\
\quad \text{forall}((\text{member}(\text{Att} = \text{Val}_1, \text{Conj}), \text{member}(\text{Att} = \text{Val}_2, \text{Obj})), \\
\quad \quad \quad \text{Val}_1 == \text{Val}_2).
\]

To find the rules we use the covering algorithm (described in pseudo-code):

\[
\begin{align*}
\text{Rules} & := []; \\
\text{E} & := S; \\
\text{while} \ \text{E} \ \text{contains} \ \text{pos examples} \ \text{do} \\
& \quad \text{[ Rule } := \text{InduceOneRule}(E); \\
& \quad \quad \text{Add Rule to Rules; \\
& \quad \quad \text{Remove from E all examples covered by Rule]}
\end{align*}
\]

Starting with an empty rule set, new rules are gradually constructed to cover some positive examples. Once a rule is added to the set, the covered positive examples are removed from the set of examples, so the next rule is induced from the reduced example set. This is repeated until all positive examples are covered.

A Prolog program based on one from Bratko is attached. The main procedure is

\[
\text{learn}(\text{Examples}, \text{Class}, \text{Description})
\]

which constructs the Description from Class and Examples. This works as follows:

- if no example in Examples is in Class then Description = [], otherwise Description = [Conj|Conj] where
1. construct Conj, a list of attribute value pairs that covers at least one positive example of Class and no example from other classes;

2. remove examples covered by Conj from Examples and recursively cover remaining examples by Cons.

Each attribute value list is constructed using

```
learn_conj(Examples,Class,Conj)
```

The main recursive call for this procedure is

```
learn_conj( Examples, Class, [Cond | Conjs]) :-
    choose_cond( Examples, Class, Cond),
    filter( Examples, [ Cond], Examples1),
    learn_conj( Examples1, Class, Conds).
```

The Cond is chosen by means of a heuristic. Cond is of the form Att=Val initially chosen from the set of attributes and appropriate values. These pairs Att=Val are restricted to suitable candidates by ensuring there is an example from a different class whose Att value is different from Val. Amongst all these candidates, a score is calculated by taking the number of negative occurrences of Cond (in a different class) from the number of positive occurrences of Cond. This is performed by the procedure score. The procedure choose_cond then selects the attribute value pair with the best score.

```
choose_cond( Examples, Class, AttVal) :-
    findall( AV/Score, score( Examples, Class, AV,Score), AVs),
    best( AVs, AttVal).
```

This heuristic is based on the intuition that a good attribute value pair should have as many positive examples as possible and as few negative ones. Once this Cond has been selected, the examples satisfying Cond are obtained, yielding Examples1 and learn_conj called on this set to produce Conds.

If we execute the goals learn(nut), learn(key), learn(scissors) on the given examples then we would obtain the following descriptions:

```
nut <= [ [shape=compact, holes=1] ]

key <= [ [shape=other, size=small],
         [holes=1, shape=long] ]

scissors <= [ [holes=2, size=large] ]

bolt <= [ [holes=none, size=small] ]
```