7 Machine Learning Algorithms in Prolog

Chapter Objectives

Two different machine learning algorithms
- Version space search
  - Specific-to-general
  - Candidate elimination
- Explanation-based learning
  - Learning from examples
  - Generalization

Prolog meta-predicates and interpreters for learning
- Version space search
- Explanation-based learning

Chapter Contents

7.1 Machine Learning: Version Space Search

In this section and the next, we implement two machine learning algorithms: *version space search* and *explanation-based learning*. The algorithms themselves are presented in detail in Luger (2009, Chapter 10). In this chapter, we first briefly summarize them and then implement them in Prolog. Prolog is used for machine learning because, as these implementations illustrate, in addition to the flexibility to respond to novel data elements provided by its powerful built-in pattern matching, its meta-level reasoning capabilities simplify the construction and manipulation of new representations.

The Version Space Search Algorithm

Version space search (Mitchell 1978, 1979, 1982) illustrates the implementation of inductive learning as search through a concept space. A concept space is a state space representation of all possible generalizations from data in a problem domain. Version space search takes advantage of the fact that generalization operations impose an ordering on the concepts in a space, and then uses this ordering to guide the search.

Generalization and *specialization* are the most common types of operations for defining a concept space. The primary generalization operations used in machine learning and expressed in the predicate calculus (Luger 2009, Chapter 2) are:

Replacing constants with variables. For example:

```
color(ball,red)
```

generalizes to

```
color(X,red)
```

Dropping conditions from a conjunctive expression.

```
shape(X, round) ^ size(X, small) ^ color(X, red)
```

generalizes to

```
shape(X,round) ^ color(X,red)
```

Adding a disjunct to an expression.

```
shape(X,round) ^ size(X,small) ^ color(X,red)
```

generalizes to

```
shape(X,round) ^ size(X,small) ^ (color(X,red) v
    color(X,blue))
```

Replacing a property with its parent in a class hierarchy. If `primary_color` is a superclass of `red`, then

```
color(X,red)
```

generalizes to

```
color(X, primary_color)
```

We may think of generalization in set theoretic terms: let `P` and `Q` be the sets of sentences matching the predicate calculus expressions `p` and `q`, respectively. Expression `p` is more general than `q` iff $Q \subseteq P$. In the above examples, the set of sentences that match `color(X, red)` contains the set of elements that match `color(ball, red)`. Similarly, in example 2, we may think of the set of round, red things as a superset of the set of small, red, round things. Note that the "more general than" relationship defines a partial ordering on the space of logical sentences. We express this using the "≥" symbol, where `p` ≥ `q` means that `p` is more general than `q`. This ordering is a powerful source of constraints on the search performed by a learning algorithm.

We formalize this relationship through the notion of *covering*. If concept `p` is more general than concept `q`, we say that *p covers q*. We define the covers relation: let `p(x)` and `q(x)` be descriptions that classify objects as being positive examples of a concept. In other words, for an object `x`, `p(x) → positive(x)` and `q(x) → positive(x)`. `p` covers `q` iff `q(x)→ positive(x)` is a logical consequence of `p(x) → positive(x)`.

For example, `color(X, Y)` covers `color(ball, Z)`, which in turn covers `color(ball, red)`. As a simple example, consider a domain of objects that have properties and values:

```
Sizes = {large, small}
Colors = {red, white, blue}
Shapes = {ball, brick, cube}
```

These objects can be represented using the predicate `obj(Sizes, Color, Shapes)`. The generalization operation of replacing constants with variables defines the space of Figure 7.1. We may view inductive learning as searching this space for a concept that is consistent with all the training examples.

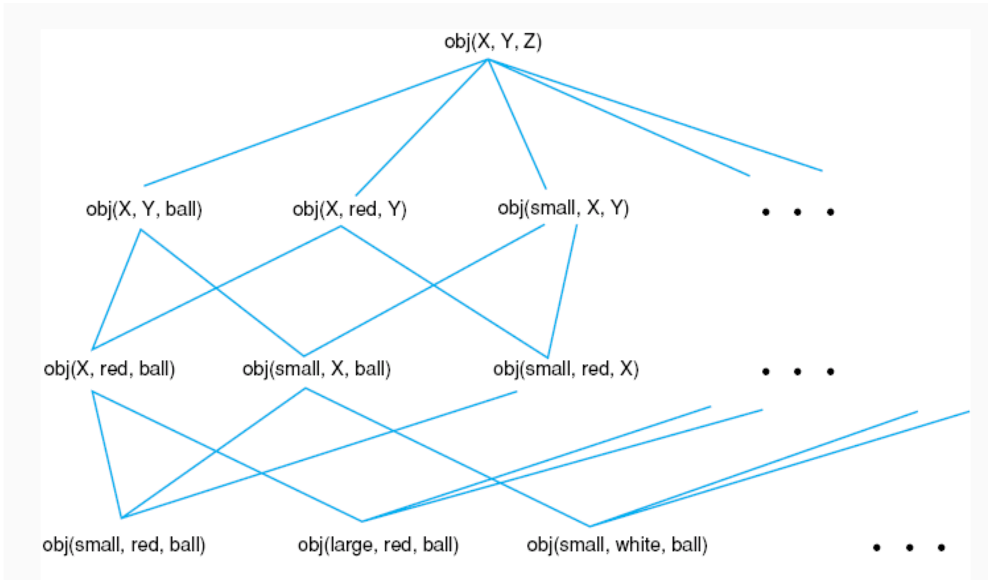

Figure 7.1. An example concept space.

We next present the *candidate elimination algorithm* (Mitchell 1982) for searching the concept space. This algorithm relies on the notion of a *version space*, which is the set of all concept descriptions consistent with the training examples. This algorithm works by reducing the size of the version space as more examples become available. The first two versions of this algorithm reduce the version space in a *specific to general* direction and a *general to specific* direction, respectively. The third version, called *candidate elimination*, combines these approaches into a bi-directional search. These versions of the candidate elimination algorithm are data driven; they generalize based on regularities found in the training data. Also, in using training data of known classification, these algorithms perform a variety of *supervised learning*.

Version space search uses both positive and negative examples of the target concept. Although it is possible to generalize from positive examples only, negative examples are important in preventing the algorithm from over generalizing. Not only must the learned concept be general enough to cover all positive examples; it also must be specific enough to exclude all negative examples. In the space of Figure 7.1, one concept that would cover all sets of exclusively positive instances would simply be `obj(X, Y, Z)`. However, this concept is probably too general, because it implies that all instances belong to the target concept. One way to avoid overgeneralization is to generalize as little as possible to cover positive examples; another is to use negative instances to eliminate overly general concepts. As Figure 7.2 illustrates, negative instances prevent overgeneralization by forcing the learner to specialize concepts in order to exclude negative instances. The algorithms of this section use both of these techniques.

We define *specific to general* search, for hypothesis set S, as:

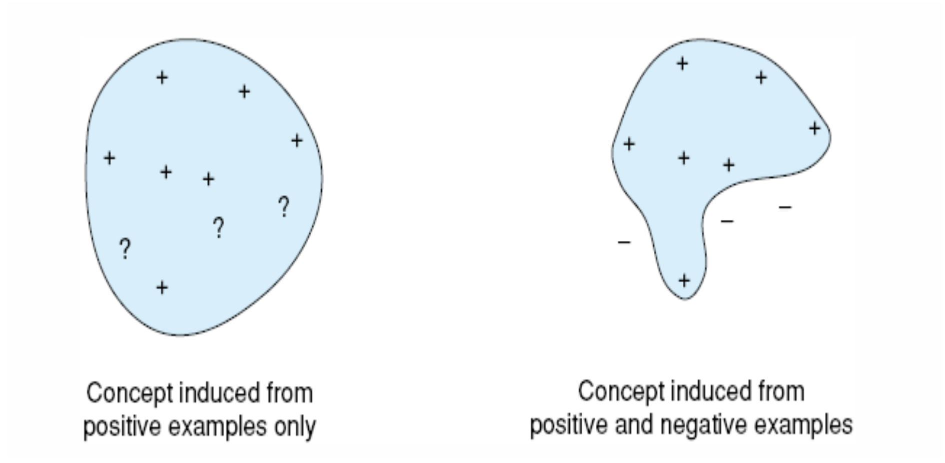
Concept induced from positive examples only
Concept induced from positive and negative examples

Figure 7.2. The role of negative examples in preventing overgeneralization.

```
Begin
Initialize S to first positive training instance;
N is the set of all negative instances seen so far;
For each positive instance p
    Begin
     For every s in S, if s does not match p,
     Replace s with its most specific
          generalization that matchs p;
     Delete from S all hypotheses more general than
          some other hypothesis in S;
     Delete from S all hypotheses that match a prev-
          iously observed negative instance in N;
    End;
For every negative instance n
    Begin
      Delete all members of S that match n;
      Add n to N to check future hypotheses
           for overgeneralization;
     End;
End
```

Specific to general search maintains a set, S, of *hypotheses*, or candidate concept definitions. To avoid overgeneralization, these candidate definitions are the *maximally specific generalizations* from the training data. A concept, `c`, is maximally specific if it covers all positive examples, none of the negative examples, and for any other concept, `c'`, that covers the positive examples, $c \leq c'$. Figure 7.3 shows an example of applying this algorithm to the version space of Figure 7.1. The specific to general version space search algorithm is built in Prolog in Section 7.1.2.

We may also search the space in a general to specific direction. This algorithm maintains a set, G, of *maximally general concepts* that cover all of the positive and none of the negative instances. A concept, `c`, is maximally general if it covers none of the negative training instances, and for any other concept, `c'`, that covers no negative training instance, `c` $\geq$ `c'`. In this algorithm, which we leave as an exercise, negative instances will lead to the specialization of candidate concepts while the algorithm uses positive instances to eliminate overly specialized concepts.

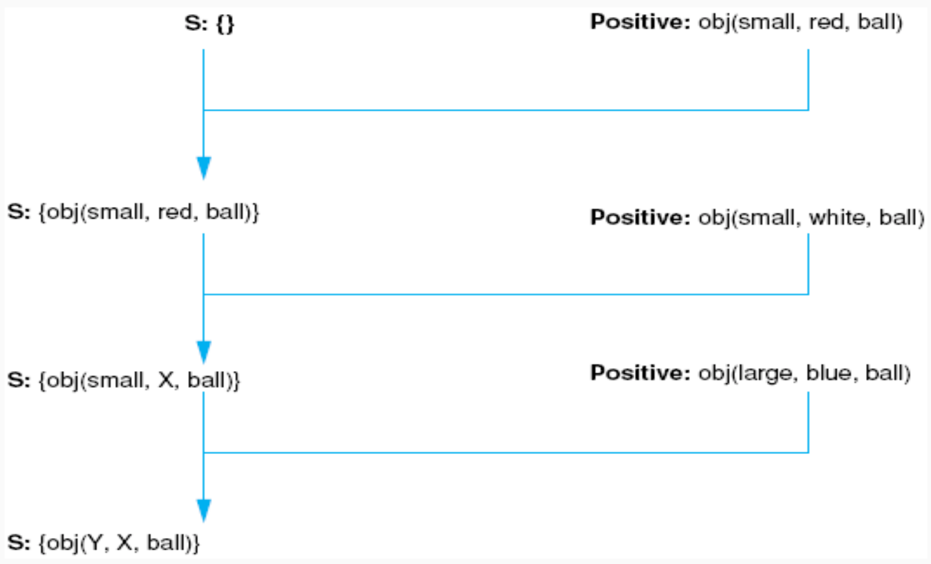
S: {}
Positive: obj(small, red, ball)
S: {obj(small, red, ball)}
Positive: obj(small, white, ball)
S: {obj(small, X, ball)}
Positive: obj(large, blue, ball)
S: {obj(Y, X, ball)}

Figure 7.3. Specific to general version space search learning the concept "ball."

The *candidate elimination algorithm* combines these two approaches into a bi-directional search. This bi-directional approach has a number of benefits for learning. The algorithm maintains two sets of candidate concepts: G, the set of maximally general candidate concepts, and S, the set of maximally specific candidates. The algorithm specializes G and generalizes S until they converge on the target concept. The algorithm is described:

```
Begin
Initialize G to the most general concept in space;
Initialize S to first positive training instance;
For each new positive instance p
    Begin
     Delete all members of G that fail to match p;
     For every s in S, if s does not match p,
         replace s with its most specific
         generalizations that match p;
     Delete from S any hypothesis more general than
         some other hypothesis in S;
     Delete from S any hypothesis more general than
         some hypothesis in G;
    End;
```

```
For each new negative instance n
    Begin
     Delete all members of S that match n;
     For each g in G that matches n, replace g with
          its most general specializations that do
          not match n;
     Delete from G any hypothesis more specific than
          some other hypothesis in G;
     Delete from G any hypothesis more specific than
          some hypothesis in S;
    End;
If G = S and both are singletons, then the algorithm
    has found a single concept that is consistent
    with all the data;
If G and S become empty, then there is no concept
    that covers all positive instances and none of
    the negative instances;
End
```

Figure 7.4 illustrates the behavior of the candidate elimination algorithm in searching the version space of Figure 7.1. Note that the figure does not show those concepts that were produced through generalization or specialization but eliminated as overly general or specific. We leave the elaboration of this part of the algorithm as an exercise and show a partial implementation in the next section.

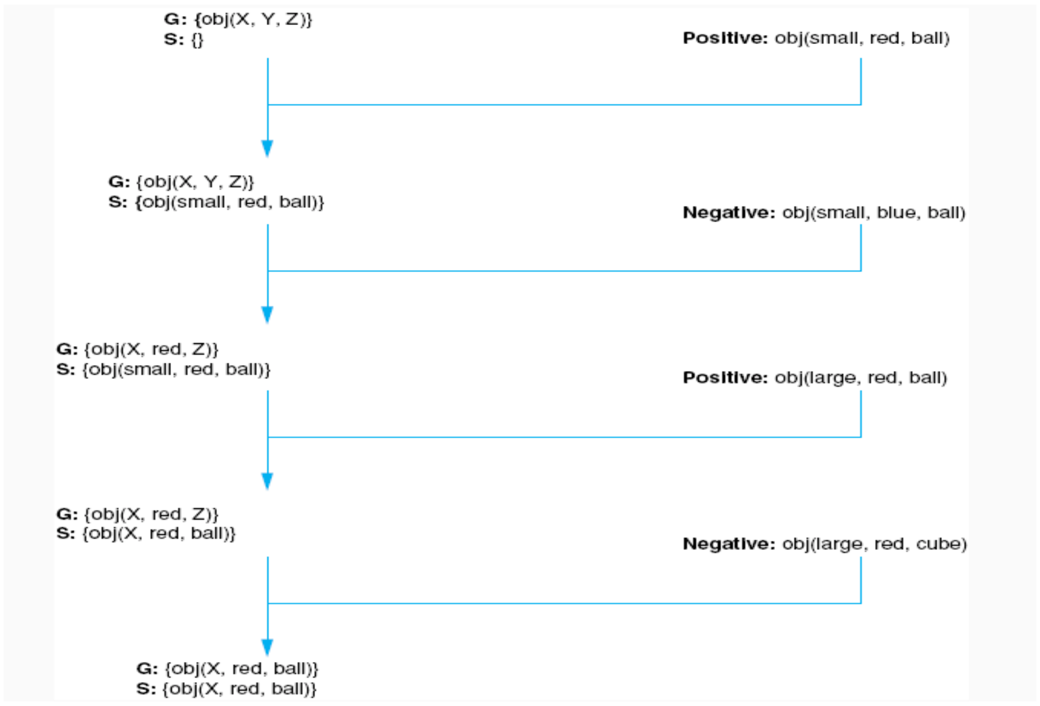
G: {obj(X, Y, Z)}
S: {}
Positive: obj(small, red, ball)
G: {obj(X, Y, Z)}
S: {obj(small, red, ball)}
Negative: obj(small, blue, ball)
G: {obj(X, red, Z)}
S: {obj(small, red, ball)}
Positive: obj(large, red, ball)
G: {obj(X, red, Z)}
S: {obj(X, red, ball)}
Negative: obj(large, red, cube)
G: {obj(X, red, ball)}
S: {obj(X, red, ball)}

Figure 7.4. The *candidate elimination* algorithm learning the concept "red ball."

A Simple Prolog Program

We first implement the specific to general search and then the full bi-directional candidate elimination algorithm. We also give hints on how to construct the general to specific version space search. These search algorithms are independent of the representation used for concepts, as long as that representation supports appropriate generalization and specialization operations. We use a representation of objects as lists of features. For example, we describe a small, red, ball with the list:

```
[small, red, ball]
```

We represent the concept of all small, red things by including a variable in the list:

```
[small, red, X]
```

This representation we call a *feature vector,* It is less expressive than full logic, e.g., it cannot represent the class "all red or green balls." However, it simplifies generalization, and provides a strong *inductive bias* (Luger 2009, Section 10.4). We generalize a feature vector by substituting a variable for a constant, for example, the most specific common generalization of `[small, red, ball]` and `[small, green, ball]` is `[small, X, ball]`. This vector will cover both of the specializations and is the most specific vector to do so.

We define one feature vector as *covering* another if the first is either identical to or more general than the second. Note that unlike unification, `covers` is asymmetrical: values exist for which `X` covers `Y`, but `Y` does not cover `X`. For example, `[X, red, ball]` covers `[large, red, ball]` but the reverse is not true. We next define the predicate `covers` for feature vectors as:

```
covers([ ], [ ]).
covers([H1 | T1], [H2 | T2]) :-
     var(H1), var(H2), covers(T1, T2).
     % variables cover each other
covers([H1 | T1], [H2 | T2]) :-
     var(H1), atom(H2), covers(T1, T2).
     % a variable covers a constant
covers([H1 | T1], [H2 | T2]) :-
     atom(H1), atom(H2), H1 = H2,
     covers(T1, T2).
     % matching constants
```

We next need to determine whether one feature vector is strictly more general than another; i.e., the vectors are not identical. We define the `more_general/2` predicate as:

```
more_general(X, Y) :- not(covers(Y,X)),covers(X,Y).
```

We implement generalization of feature vectors as a predicate, `generalize` with three arguments, where the first argument is a feature vector representing an hypothesis (this vector may contain variables), the second argument is an instance, containing no variables. `generalize` binds its third argument to the most specific generalization of the hypothesis that covers the instance. `generalize` recursively scans the feature vectors, comparing corresponding elements. If two elements match, the result contains the value of the hypotheses vector in that position; if two elements do not match, it places a variable in the corresponding position of the generalized feature vector. Note the use of the expression `not(Feature \= Inst_prop)`, in the second definition of `generalize`; this double negative enables us to test if two atoms will unify without actually performing the unification and forming any unwanted variable bindings. We define generalize:

```
generalize([ ], [ ], [ ]).
generalize([Feature | Rest],[Inst_prop | Rest_inst],
        [Feature | Rest_gen]) :-
    not(Feature \= Inst_prop),
    generalize(Rest, Rest_inst, Rest_gen).
generalize([Feature | Rest],[Inst_prop | Rest_inst],
        [_ | Rest_gen]) :-
    Feature \= Inst_prop,
    generalize(Rest, Rest_inst, Rest_gen).
```

These predicates define the essential operations on feature vector representations. The remainder of the implementation that follows is independent of any specific representation, and may be adapted to a variety of representations and generalization operators.

As discussed in Section 7.1, we may search a concept space in a specific to general direction by maintaining a list `H` of candidate hypotheses. The hypotheses in `H` are the most specific concepts that cover all the positive examples and none of the negative examples seen so far. The heart of the algorithm is `process` with five arguments. The first argument to `process` is a training instance, `positive(X)` or `negative(X)`, indicating that `X` is a positive or negative example. The second and third arguments are the current list of hypotheses and the list of negative instances. On completion, `process` binds its fourth and fifth arguments to the updated lists of hypotheses and to the negative examples, respectively.

The first clause in the definition below initializes an empty hypothesis set to the first positive instance. The second handles positive training instances by generalizing candidate hypotheses to cover the instance. It then deletes all over-generalizations by removing those that are more general than some other hypothesis and eliminating any hypothesis that covers some negative instance. The third clause in the definition handles negative examples by deleting any hypothesis that covers those instances.

```
process(positive(Instance), [ ], N, [Instance], N).
process(positive(Instance), H, N, Updated_H, N) :-
    generalize_set(H, Gen_H, Instance),
    delete(X, Gen_H,(member(Y, Gen_H),
        more_general(X, Y)), Pruned_H),
    delete(X, Pruned_H, (member(Y, N),
    covers(X, Y)), Updated_H).
```

```
process(negative(Instance), H, N, Updated_H,
        [InstanceN]) :-
     delete(X, H, covers(X, Instance), Updated_H).
process(Input, H, N, H, N):-      %Catches bad input
     Input \= positive(_),
     Input \= negative(_),
     write('Enter either positive(Instance) or
          negative(Instance) '), nl.
```

An interesting aspect of this implementation is the `delete` predicate, a generalization of the usual process of deleting all matches of an element from a list. One of the arguments to `delete` is a test that determines which elements to remove from the list. Using `bagof`, `delete` matches its first argument (usually a variable) with each element of its second argument (this must be a list). For each such binding, it then executes the test specified in argument three: this test is any sequence of callable Prolog goals. If a list element causes this test to fail, `delete` includes that element in the resulting list. It returns the result in its final argument. The `delete` predicate is an excellent example of the power of meta reasoning in Prolog: by letting us pass in a specification of the elements we want to remove from a list, `delete` gives us a general tool for implementing a range of list operations. Thus, `delete` lets us define the various filters used in `process/5` in an extremely compact fashion. We define `delete`:

```
delete(X, L, Goal, New_L) :-
     (bagof(X, (member(X, L), not(Goal)), New_L);
          New_L = [ ]).
```

`Generalize_set` is a straightforward predicate that recursively scans a list of hypotheses and generalizes each one against a training instance. Note that this assumes that we may have multiple candidate generalizations at one time. In fact, the feature vector representation of Section 7.1.1 only allows a single most specific generalization. However, this is not true in general and we have defined the algorithm for the general case.

```
generalize_set([ ], [ ], _).
generalize_set([Hypothesis | Rest],
        Updated_H, Instance):-
     not(covers(Hypothesis, Instance)),
     (bagof(X, generalize(Hypothesis, Instance, X),
          Updated_head); Updated_head = [ ]),
     generalize_set(Rest, Updated_rest, Instance),
     append(Updated_head, Updated_rest, Updated_H).

generalize_set([Hypothesis | Rest],
        [Hypothesis | Updated_rest], Instance) :-
     covers(Hypothesis, Instance),
     generalize_set(Rest, Updated_rest, Instance).
```

specific_to_general implements a loop that reads and processes training instances:

```
specific_to_general(H, N) :-
     write('H = '), write(H), nl, write('N = '),
     write(N), nl,
     write('Enter Instance: '), read(Instance),
     process(Instance, H, N, Updated_H, Updated_N),
     specific_to_general(Updated_H, Updated_N).
```

The following transcript illustrates the execution of this algorithm.

```
?- specific_to_general([], []).
H = [ ]
N = [ ]
Enter Instance: positive([small, red, ball]).
H = [[small, red, ball]]
N = [ ]
Enter Instance: negative([large, green, cube]).
H = [[small, red, ball]]
N = [[large, green, cube]]
Enter Instance: negative([small, blue, brick]).
H = [[small, red, ball]]
N = [[small, blue, brick], [large, green, cube]]
Enter Instance: positive([small, green, ball]).
H = [[small, _66, ball]]
N = [[small, blue, brick], [large, green, cube]]
Enter Instance: positive([large, blue, ball]).
H = [[_116, _66, ball]]
N = [[small, blue, brick], [large, green, cube]]
```

The second version of the algorithm searches in a general to specific direction, as described in Section 7.1.1. In this version, the set of candidate hypotheses are initialized to the most general possible concept. In the case of the feature vector representation, this is a list of variables. It then specializes candidate concepts to prevent them from covering negative instances. In the feature vector representation, this involves replacing variables with constants. When given a new positive instance, it eliminates any candidate hypothesis that fails to cover that instance.

We implement this algorithm in a way that closely parallels the specific to general search just described, including the use of the general delete predicate to define the various filters of the list of candidate concepts. In defining a general to specific search, process will have six arguments. The first five reflect the specific to general version: the first a training instance of the form `positive(Instance)` or `negative(Instance)`; the second is a list of candidate hypotheses; these are the most general hypotheses that cover no negative instances. The third argument is the list of positive examples, used to delete any overly specialized candidate hypothesis. The fourth and fifth arguments are the updated lists of hypotheses and positive examples, respectively. The sixth argument is a list of allowable variable substitutions for specializing concepts.

Specialization by substituting a constant for a variable requires the algorithm to know the allowable constant values for each field of the feature vector. These values will have to be passed in as the sixth argument of process. In our example of `[Size, Color, Shape]` vectors, a sample list of types might be: `[[small, medium, large], [red, white, blue], [ball, brick, cube]]`. Note that the position of each sublist determines the position in a feature vector where those values are used; for example, the first sublist defines allowable values for the first position of a feature vector. We leave construction of this algorithm as an exercise. For guidance we include a run of our implementation:

```
?- general_to_specific([[_, _, _]], [ ],
          [[small, medium, large],
           [red, blue, green],
           [ball, brick, cube]]).
H = [[_0, _1, _2]]
P = [ ]
Enter Instance: positive([small, red, ball]).
H = [[_0, _1, _2]]
P = [[small, red, ball]]
Enter Instance; negative([large, green, cube]).
H = [[small, _89, _90], [_79, red, _80],
          [_69, _70, ball]]
P = [[small, red, ball]]
Enter Instance: negative([small, blue, brick]).
H = [[_79, red, _80],[_69, _70, ball]]
P = [[small, red, ball]]
Enter Instance: positive([small, green, ball]).
H = [[_69,_70,ball]]
P = [[small, green, ball], [small, red, ball]]
```

The full candidate elimination algorithm, as described in Section 7.1.1, is a combination of the two single direction searches. As before, the heart of the algorithm is the definition of `process`, with six arguments. The first argument to `process` is a training instance. Arguments two and three are `G` and `S`, the sets of maximally general and maximally specific hypotheses respectively. The fourth and fifth arguments are bound to the updated versions of these sets. The sixth argument of `process` lists allowable variable substitutions for specializing feature vectors.

On positive instances, `process` generalizes `S`, the set of most specific generalizations, to cover the training instance. It then eliminates any elements of `S` that have been over generalized. It also eliminates any elements of `G` that fail to cover the training instance. It is interesting to note that an element of `S` is overly general if there is no element of `G` that covers it; this is true because `G` contains those candidate hypotheses that are both maximally general and cover no negative instances. `process` uses `delete` to eliminate these hypotheses.

On a negative training instance, `process` specializes all hypotheses in `G` to exclude that instance. It also eliminates any candidates in `S` that cover the negative instance. As discussed above, specialization of feature vectors requires replacing variables with constants. This requires that we pass a list of allowable substitutions as the sixth argument to `process`. We define `process`:

```
process(negative(Instance), G, S, Updated_G,
        Updated_S, Types) :-
    delete(X, S, covers(X, Instance), Updated_S),
    specialize_set(G, Spec_G, Instance, Types),
    delete(X, Spec_G, (member(Y, Spec_G),
        more_general(Y, X)), Pruned_G),
    delete(X, Pruned_G, (member(Y, Updated_S),
        not(covers(X, Y))), Updated_G).
process(positive(Instance), G, [ ],
        Updated_G, [Instance],_) :-  %Initialize S
    delete(X, G, not(covers(X, Instance)),
        Updated_G).
process(positive(Instance), G, S,
        Updated_G, Updated_S,_) :-
    delete(X, G, not(covers(X, Instance)),
        Updated_G),
    generalize_set(S, Gen_S, Instance),
    delete(X, Gen_S, (member(Y, Gen_S),
        more_general(X, Y)), Pruned_S),
    delete(X, Pruned_S, not((member(Y, Updated_G),
        covers(Y, X)))), Updated_S).
process(Input, G, P, G, P,_) :-
    Input \= positive(_), Input \= negative(_),
    write(`Enter a positive(Instance) or
        negative(Instance): '), nl.
```

`generalize_set` generalizes all members of a set of candidate hypotheses to cover a training instance. It is identical to the version defined for the specific to general search. `specialize_set` takes a set of candidate hypotheses and computes all maximally general specializations of those hypotheses that exclude (do not cover) a training instance. Note the use of `bagof` to get all specializations.

```
specialize_set([ ], [ ], _, _).
specialize_set([HypothesisRest],
        Updated_H, Instance, Types) :-
    covers(Hypothesis, Instance),
    (bagof(Hypothesis, specialize(Hypothesis,
        Instance,Types), Updated_head) ;
        Updated_head = [ ]),
    specialize_set(Rest, Updated_rest, Instance,
        Types),
    append(Updated_head, Updated_rest, Updated_H).
```

```
specialize_set([HypothesisRest],
          [HypothesisUpdated_rest],Instance,Types):-
     not (covers(Hypothesis, Instance)),
     specialize_set(Rest, Updated_rest,
          Instance, Types).
```

specialize finds an element of a feature vector that is a variable. It binds that variable to a constant value that it selects from the list of allowable values, and which does not match the training instance. Recall that specialize_set called specialize with bagof to get all specializations. If we call specialize once, it will substitute a constant into the first variable; bagof causes it to produce all specializations.

```
specialize([Prop_], [Inst_prop_],
          [Instance_values_]) :-
     var(Prop), member(Prop, Instance_values),
     Prop \= Inst_prop.
specialize([_Tail], [_Inst_tail], [_Types]) :-
     specialize(Tail, Inst_tail, Types).
```

The definitions of generalize, more_general, covers, and delete are the same as in the specific to general algorithm defined above. candidate_elim implements a top-level read-process loop, printing out the current G set, the S set, and calls process on the input:

```
candidate_elim([G],[S],_) :-
     covers(G,S),covers(S,G),
     write('target concept is: '), write(G),nl.
candidate_elim(G, S, Types) :-
     write('G= '), write(G), nl, write('S= '),
     write(S), nl, write('Enter Instance: '),
     read(Instance),
     process(Instance, G, S, Updated_G,
          Updated_S, Types),
     candidate_elim(Updated_G, Updated_S, Types).
```

To conclude this section we present a trace of the candidate elimination algorithm. Note initializations of G, S, and the list of allowable substitutions:

```
?- candidate_elim([[_, _, _]], [ ],
          [[small, medium, large],
           [red, blue, green],
           [ball, brick, cube]]).
G= [[_0, _1, _2]]
S= [ ]
Enter Instance: positive([small, red, ball]).
G= [[_0, _1, _2]]
S= [[small, red, ball]]
Enter Instance: negative([large, green, cube]).
G= [[small, _96, _97], [_86, red, _87],
          [_76, _77, ball]]
```

```
S= [[small, red, ball]]
Enter Instance: negative([small, blue, brick]).
G= [[_86, red, _87], [_76, _77, ball]]
S= [[small, red, ball]]
Enter Instance: positive([small, green, ball]).
G= [[_76, _77, ball]]
S= [[small, _351, ball]]
Enter Instance: positive([large, red, ball]).
target concept is: [_76, _77, ball] yes
```

7.2 Explanation Based Learning in Prolog

The Explanation Based Learning Algorithm

In this section, we describe briefly the algorithms for explanation-based learning, Section 7.2.1 and then present a Prolog implementation of the explanation-based learning in Section 7.2.2. Our implementation is based upon Kedar-Cabelli and McCarty's formulation (Kedar-Cabelli and McCarty 1987; Luger 2009, Section 10.5.2), called `prolog_ebg`, and illustrates the power of unification in Prolog. Even though it is quite difficult to implement explanation-based learning in many languages, the Prolog version is fairly simple.

Explanation-based learning uses an explicitly represented domain theory to construct an explanation of a training example, usually a proof that the example logically follows from the theory. By generalizing from the explanation of the instance, rather than from the instance itself, explanation-based learning filters noise, selects relevant aspects of experience, and organizes training data into a coherent structure.

There are several alternative formulations of this idea. For example, the STRIPS program for representing general operators for planning (see Section 6.3) has exerted a powerful influence on this research (Fikes et al. 1972). Meta-DENDRAL established the power of theory-based interpretation of training instances (Luger 2009, Section 10.5.1). A number of authors (DeJong and Mooney 1986, Minton 1988) have proposed alternative formulations of this idea. The *Explanation-Based Generalization* algorithm of Mitchell et al. (1986) is also typical of the genre. In this section, we examine a variation of the explanation-based learning (EBL) algorithm developed by DeJong and Mooney (1986).

EBL begins with:

1. *A target concept*. The learner's task is to determine an effective definition of this concept. Depending upon the specific application, the target concept may be a classification, a theorem to be proven, a plan for achieving a goal, or a heuristic for a problem solver.
2. *A training example*, an instance of the target.
3. *A domain theory*, a set of rules and facts that are used to explain how the training example is an instance of the goal concept.
4. *Operationality criteria*, some means of describing the form concept definitions may take.

To illustrate EBL, we present an example of learning about when an object is a cup. This is a variation of a problem explored by Winston et al. (1983) and adapted to explanation-based learning by Mitchell et al. (1986). The target concept is a rule that may be used to infer whether an object is a cup; again, we adopt a predicate calculus representation:

```
premise(X) → cup(X)
```

where `premise` is a conjunctive expression containing the variable `X`.

Assume a domain theory that includes the following rules about cups:

```
liftable(X) ^ holds_liquid(X) → cup(X)

part(Z, W) ^ concave(W) ^ points_up(W) →
     holds_liquid(Z)

light(Y) ^ part(Y, handle) → liftable(Y)

small(A) → light(A)

made_of(A, feathers) → light(A)
```

The training example is an instance of the goal concept. That is, we are given:

```
cup(obj1)
small(obj1)
part(obj1, handle)
owns(bob, obj1)
part(obj1, bottom)
part(obj1, bowl)
points_up(bowl)
concave(bowl)
color(obj1, red)
```

Finally, assume the operationality criteria require that target concepts be defined in terms of observable, structural properties of objects, such as `part` and `points_up`. We may provide domain rules that enable the learner to infer whether a description is operational, or we may simply list operational predicates.

A theorem prover constructs an explanation of why the example is an instance of the training concept: a proof that the target concept logically follows from the example, as in Figure 7.5. Note that this explanation eliminates such irrelevant concepts as `color(obj1, red)` from the training data and captures (only) those aspects of the example known to be relevant to the goal.

The next stage of explanation-based learning generalizes the explanation to produce a concept definition that may be used to recognize other cups. EBL accomplishes this by substituting variables for those constants in the proof tree that depend solely on the particular training instance, as may be seen in Figure 7.5 (bottom). Based on the generalized tree, EBL defines a new rule whose conclusion is the root of the tree and whose premise is the conjunction of the leaves:

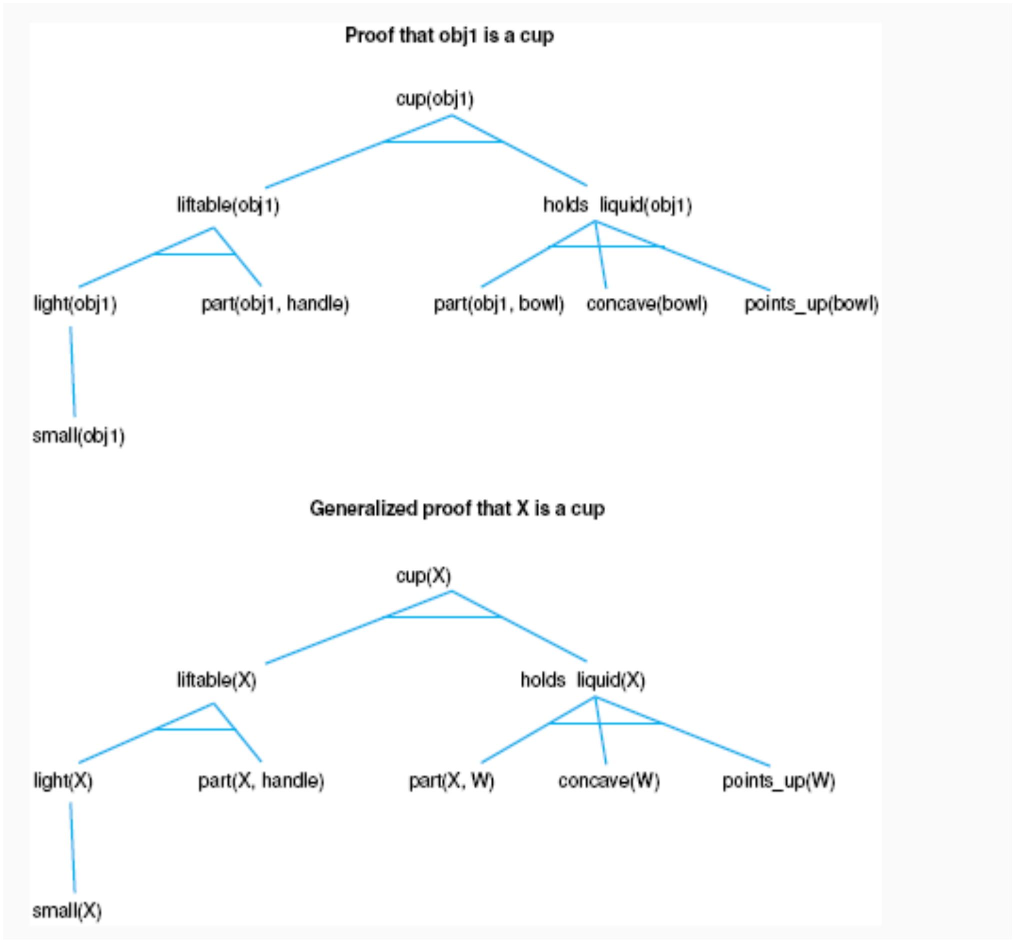

Figure 7.5. A specific (top) and generalized (bottom) proof that an object, X, is a cup.

```
small(X) ∧ part(X, handle) ∧ part(X, W) ∧ concave(W)
  ∧ points_up(W) → cup(X).
```

In constructing a generalized proof tree, our goal is to substitute variables for those constants that are part of the training instance while at the same time retaining those constants and constraints that are part of the domain theory. In this example, the constant handle originated in the domain theory rather than the training instance. We have retained it as an essential constraint in the acquired rule.

We may construct a generalized proof tree in a number of ways using a training instance as a guide. Mitchell et al. (1986) accomplish this by first constructing a proof tree that is specific to the training example and subsequently generalizing the proof through a process called *goal regression*. Goal regression matches the generalized goal (in our example, `cup(X)`) with the root of the proof tree, replacing constants with variables as required for the match. The algorithm applies these substitutions recursively through the tree until all appropriate constants have been generalized. See Mitchell et al. (1986) for a detailed description of this process. We next implement the explanation based learning algorithm in Prolog.

Prolog Implementation of EBL

Instead of building an explanation structure and maintaining separate sets of specific and general substitutions as done in Section 7.2.1, our algorithm will build both the proof of the training instance and the generalized proof tree concurrently.

For this example, we represent proof trees as we did in `exshell` (Section 6.2). When `prolog_ebg` discovers a fact, it returns this fact as the leaf of a proof tree. The proof of conjunctive goals is the conjunction of the proof of the conjuncts. The proof of a goal that requires rule chaining is represented as `(Goal :- Proof)`, where `Proof` becomes bound to the proof tree for the rule premise.

The heart of the algorithm is `prolog_ebg`. This predicate takes four arguments: the first is the goal being proved in the training example, the second is the generalization of that goal. If the domain theory enables a proof of the specific goal, it binds the third and fourth arguments to a proof tree for the goal and the generalization of that proof. For instance, implementing the cup example from Section 7.2.1, we would call `prolog_ebg` with the arguments:

```
prolog_ebg(cup(obj1), cup(X), Proof, Gen_proof).
```

We assume that Prolog has the domain theory and training instance of Section 7.2.1. When `prolog_ebg` succeeds; `Proof` and `Gen_proof` are the proof trees of Figure 7.5.

`prolog_ebg` is a straightforward variation of the `exshell` meta-interpreter of Section 6.2. The primary difference is that `prolog_ebg` solves the goal and the generalized goal in parallel. A further interesting aspect of the algorithm is the use of the predicate `duplicate` to create two versions of each rule: the first version is the rule as it appears in the domain theory, the second binds variables in the rule to the values in the training instance. We define `prolog_ebg`:

```
prolog_ebg(A, GenA, A, GenA) :- clause(A, true).
prolog_ebg((A, B), (GenA, GenB), (AProof, BProof),
        (GenAProof, GenBProof)) :- !,
    prolog_ebg(A, GenA, AProof, GenAProof),
    prolog_ebg(B, GenB, BProof, GenBProof).
prolog_ebg(A, GenA, (A :- Proof), (GenA :-
        GenProof)) :-
    clause(GenA, GenB),
    duplicate((GenA :- GenB), (A :- B)),
    prolog_ebg(B, GenB, Proof, GenProof).
```

`Duplicate` relies upon the behavior of `assert` and `retract` to create a copy of a Prolog expression with all new variables.

```
duplicate(Old, New) :-
    assert('$marker'(Old)),
    retract('$marker'(New)).
```

`extract_support` returns the sequence of the highest level operational nodes, as defined by the predicate `operational`. The `extract_support` predicate implements a recursive tree walk, terminating the recursion when it finds nodes in the proof tree that qualifies as `operational`.

```
extract_support(Proof, Proof) :- operational(Proof).
extract_support((A :- _), A) :- operational(A).
extract_support((AProof, BProof), (A, B)) :-
     extract_support(AProof, A),
     extract_support(BProof, B).

extract_support((_ :- Proof), B) :-
     extract_support(Proof, B).
```

The final component of the explanation based generalization algorithm constructs the learned rule, using the `prolog_ebg` and `extract_support` predicates:

```
ebg(Goal, Gen_goal, (Gen_goal :- Premise)) :-
     prolog_ebg(Goal, Gen_goal, _, Gen_proof),
     extract_support(Gen_proof, Premise).
```

We illustrate the execution of these predicates with the example of learning structural definitions of cups from Section 7.2.1, as described originally by Mitchell et al. (1986). We begin with a domain theory for cups and other physical objects. The theory includes the rules:

```
cup(X) :- liftable(X), holds_liquid(X).
holds_liquid(Z) :-
     part(Z, W), concave(W), points_up(W).
liftable(Y) :-
     light(Y), part(Y, handle).
light(A):- small(A).
light(A):- made_of(A, feathers).
```

The learner is also given the following example, in which `obj1` is known to be a `cup`:

```
small(obj1).
part(obj1, handle).
owns(bob, obj1).
part(obj1, bottom).
part(obj1, bowl).
points_up(bowl).
concave(bowl).
color(obj1, red).
```

The operationality criteria define predicates that may be used in a rule:

```
operational(small(_)).
operational(part(_, _)).
operational(owns(_, _)).

operational(points_up(_)).

operational(concave(_)).
```

A run of the algorithm on the cup example illustrates the behavior of these predicates. Of course, symbols such as "_0" and "_106" indicate specific variables in Prolog, i.e., all uses of _106 represent the same variable:

```
?- prolog_ebg(cup(obj1), cup(X), Proof, Gen_proof).

X = _0,

Proof = cup(obj1) :-
        ( (liftable(obj1) :-
            ( (light(obj1) :-
                 small(obj1)),
                 part(obj1, handle))),
          (holds_liquid(obj1) :-
               (part(obj1, bowl),
                concave(bowl),
                points_up(bowl))))

 Gen_prooof = cup(_0) :-
        ( (liftable(_0) :-
            ( (light(_0) :-
                 small(_0)),
                part(_0, handle))),
          (holds_liquid(_0) :-
              (part(_0, _106),
                concave(_106),
                points_up(_106))))
```

When we give extract_support the generalized proof from the previous execution of prolog_ebg, it returns the operational nodes of the proof, in left-to-right order:

```
?- extract_support((cup(_0) :-
        ( (liftable(_0) :-
              ( (light(_0) :-
                 small(_0)),
                 part(_0, handle))),
          (holds_liquid(_0) :-
             (part(_0,_106),
              concave(_106),
              points_up(_106))))), Premise),
_0 = _0, _106 = _1,
Premise = (small(_0), part(_0,handle)), part(_0,_1),
          concave(_1), points_up(_1)
```

Finally, ebg uses these predicates to construct a new rule from the example.

```
?- ebg(cup(obj1), cup(X), Rule).

X = _0,

Rule = cup(_0) :-
     ((small(_0), part(_0, handle)), part(_0,_110),
          concave(_110), points_up(_110))
```

In the next two chapters we address the problem of understanding natural language. We first, in Chapter 8, discuss issues in semantic (or language meaning) representations, building Prolog structures for conceptual dependencies. We then build several recursive descent parsers to capture syntactic relationships in sentences. These meta-interpreters demonstrate context free, context sensitive, deterministic, and probabilistic parsing. In Chapter 9 we present the Earley parser in Prolog, which uses techniques from dynamic programming. The Earley parser is often called a *chart* parser.

Exercises

1. The run of the candidate elimination algorithm shown in Figure 7.4 does not show candidate concepts that were produced but eliminated because they were either overly general, overly specific, or subsumed by some other concept. Re-do the execution trace, showing these concepts and the reasons each was eliminated.

2. Develop a domain theory for explanation-based learning in some problem area of your choice. Trace the behavior of an explanation-based learner in applying this theory to several training instances.

3. Implement a general to specific search of the version space using the feature vector representation of Section 7.2. We can specialize feature vectors by replacing variables with constants; since this requires telling the algorithm of allowable values for each field of the feature vector, we must pass this in as an extra argument. The following definition of `run_general`, the top-level goal, illustrates the necessary initializations for the example used in the text: objects may be `small`, `medium`, or `large`, their color may be `red`, `blue`, `green`, and their shape may be `ball`, `brick`, or `cube`.

```
run_general :-
      general_to_specific([[_, _, _]], [ ],
            [[small,medium,large], [red,blue,green],
            [ball,brick,cube]]).
```

4. Create another domain theory example, as proposed in exercise 2 above, and run it with `prolog_ebg`.

5. Extend the definition of `ebg` so that, after it constructs a new rule, it asserts it to the logic database where it may be used in future queries. Test the performance of the resulting system using a theory for a suitably rich domain. You might do this by constructing a theory for a domain of your choice, or extending the theory from the cup example to allow it to explain different types of cups such as Styrofoam cups, cups without handles, etc.