CS 3304 Project 2: Decision Tree Learning Spring 2002

Due: 12 April, 11:59:59 p.m.

In this assignment you will write a Prolog program to implement an inductive learning algorithm for decision trees.

Inductive learning is a process of learning from classification examples. The input is a set of examples, each of which consists of values for some set of attributes, and the output is a way of classifying the examples by their attributes. For instance, we might have as input the following table

Size	Color	Spots	Sex	Classification
small	white	none	male	Bichon Frise
big	black	none	male	German Shephard
small	brown	none	female	Chihuahua
big	black	white	female	Great Dane
medium	white	black	male	Dalmation

The result would be a hypothesis that allows us to classify these examples by dog breed.

There are a number of representations that can be used for classification, and we will use a very common one which is the decision tree. The internal nodes of a decision tree are labeled by attributes, and the branches out of each node are labeled by a value of the attribute. A leaf is labeled by a class, which could be null. A path in the decision tree from the root to a leaf provides a classification based on the attribute values.

Attributes and Examples We will use the following notation to define the attributes to decribe objects or concepts.

```
attribute(AttributeName, ValueList)
```

This will be given as atoms such as

```
attribute( size, [ small, medium, big]).
attribute( color, [ while, yellow, brown, black]).
attribute( spots, [ black, brown, white, none]).
attribute( sex, [ male, female ]).
attribute( eye, [ brown, blue, grey ]).
```

Examples are defined by giving a classification for a particular set of attribute values:

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

These also will be given as atoms, and continuing the example above we have

```
example( bichon, [ size = small, color = white, spots = none, sex = male]).
example( germanshep, [ size = big, color = black, spots = none, sex = female]).
example( chihuahua, [ size = small, color = white, spots = none, sex = female]).
example( greatdane, [ size = big, color = black, spots = white, sex = female]).
example( dalmation, [ size = medium, color = white, spots = black, sex = male]).
```

Decision Trees You are to use the following notation for the decision trees:

- `null` — an empty tree, indicating no examples.
- `leaf(Class)` — a single leaf labeled with a class, indicating all examples on path are of the same class.
- `tree(Attribute, [ Val1:Subtree1, Val2:Subtree2,...])` — an internal node labeled with `Attribute`, the list identifies the branches and their labels (e.g., `Val1`). The examples in this tree belong to several classes.

Algorithm It will be useful to you to think of the algorithm recursively and by cases:

1. If there are no examples then the tree is empty.
2. If all examples belong to the same class, then the tree is a leaf labeled by that class.
3. Otherwise:
   (a) Select the attribute that is the most informative (see below) to label an internal node.
   (b) For each value of the attribute:
      i. Find all examples that have the attribute value
      ii. Build a tree with this subset of examples
      iii. Add tree to new internal node

Selecting Attributes The idea behind selecting attributes is to choose one which will divide the examples well by classes. Most approaches are based on information theory. The classical measure of information content is entropy, which is the amount of information needed to classify an object:

$$I = -\sum_c p(c) \log_2 p(c)$$

where c ranges over all classes of objects, and $p(c)$ is the probability that an object in the training set S is in class c. Of course, we are interested in choosing attributes, so we want an information measure relative to an attribute A. We can do this by measuring the information content of the partitioning of S by the values v of A:

$$I_{res}(A) = -\sum_v p(A = v) \sum_c p(c|A = v) \log_2 p(c|A = v)$$

where $p(c|A = v)$ is the conditional probability that an object is in class c if attribute A has value v in the set S. The amount of information needed to classify the values of attribute A is given by

$$I(A) = -\sum_v p(A = v) \log_2 p(A = v)$$

where the v ranges over the values of A.

Now, the measure we will use is called the information gain ratio, which is defined as

$$\text{GainRatio}(A) = \frac{I - I_{res}(A)}{I(A)}.$$

We want the attribute with the highest gain ratio.

Implementation You are to implement the decision tree learning algorithm in Prolog. The following steps indicate what is required and guide you through the process.

1. Define a predicate `attribute_impurity(Examples,Attribute,Impurity)` such that `Impurity` is the information gain ratio for `Attribute` with respect to `Examples`. You will need to compute the probabilities used in the formulas above. Since you have to use these values multiple times it may be useful to construct a mechanism by which you store the values in a list and then can retrieve them as needed (or you may use the `assert` predicates described below).

2. Define a predicate `choose_attribute(Attributes, Examples, BestAttribute)` such that `BestAttribute` is the attribute with the highest information gain ratio. You may find it useful to use the builtin `setof` predicate discussed below.

3. Define a predicate `induce_tree(Attributes,Examples,Tree)` Such that `Tree` is a decision tree constructed from `Examples` and `Attributes` using the algorithm described above. The value bound to `Tree` must use the notation given above.

4. You can execute your program using queries to the following predicate

```
% induce_tree(Tree)
%
% true if Tree is the decision tree formed from the existing example
% and attribute atoms, and by using the information gain ratio for
% selecting attributes.
%
induce_tree(Tree) :-
   findall( example(Class, Obj), example(Class, Obj), Examples),
   findall( Att, attributes(Att, _), Attributes),
   induce_tree(Attributes,Examples,Tree).
```

   The findall predicate is described below.

Remember that the only real organizational structure you have is the predicate. So, if you have something that requires recursion within another computation (e.g., finding the subtrees of an internal node), you need to define another predicate to handle it.

Also, remember that the predicates can be used to test whether something is true or to compute something so that the predicate is true.

Builtin Predicates The following might be useful.

- `setof(X,P,L)` — `L` is the list of unique objects `X` that satify predicate `P`. For instance, we might have a database of atoms like

```
companion(nate,ronaldanne).
companion(sam,troika).
companion(tess,troika).
companion(joel,suki).
```

The query

```
setof(Person,companion(Person,troika),L)
```

would return with `L = [sam, tess]`. The values in the list are ordered in increasing order by a default ordering on the terms, and duplicates are removed. You can also construct pairs in the set. For instance suppose we want to pair the person with their age, and had the following atoms in addition to those above:

```
age(nate,1).
age(sam,1).
age(tess,2).
age(joel,42).
```

Then the query

```
setof(Person/Age,(companion(Person,troika),age(Person,Age)),L).
```

would return the binding `L = [sam/1, tess/2]`. The predicate in this case is a conjunction and the parenthesis are necessary.

- `findall(X,P,L)` — `L` is the list of all objects `X` that satisfy `L`. This is similar to `setof`, but returns everything that matches even if there are duplicates. It is used in the code given above.
- `member(X,L)` — `X` occurs in the list `L`.
- `delete(L,X,R)` — `R` is the list that results by deleting all objects that match `X` from `L`.
- `assert(C)` — add clause C to the fact/rule database. This predicate is used to modify the program during execution — usually to save computed information. You might find this useful for saving probabilities, which could be saved as facts. If you want the fact or rule to be at the beginning of the corresponding facts use `asserta(C)`. The predicate `assertz(C)` places the clause at the end of the set of clauses with the same name (in SWI-Prolog this is also what `assert(C)` does).
- `retract(C)` — removes all clauses that match with `C`. Allows you to undo an `assert`, but could do more than that if you aren't careful.

You might also want to look at the SWI-prolog documentation for information on predicates. If you do look in the documentation you might see the notation `setof/3`. This refers to the `setof` predicate with three arguments. There could be other predicates with the same name and a different number of arguments.

Implementation Constraints. You are to use the tree structure given above, and to implement the predicates as specified in Prolog to run on SWI-Prolog. There is no input and output in this program.

Commenting and Style. Your code should be neatly formatted with good indentation style.

You program should have a header block comment that minimally identifies you as the programmer, lists your pid, and lists the predicates that are implemented. Each predicate should have a block comment like that for the single argument `induce_tree` given above. Each separate clause should have at least a single comment line to describe the condition it is testing. For instance, you might have something like:

```
% knows(Knower, Knowee)
%
% true if Knower knows Knowee

% Case: everyone knows sam.
knows(_,sam).

% Case: everyone knows themself
knows(X,X).

% General case: if someone knows someone, who someone else knows, then the
% know that someone else.
knows(X,Y) :- knows(X,Z), knows(Y,Z).
```

What to Submit. You are to submit a single (text) pl (Prolog) file to the curator containing your function definitions. Do not include any attribute or example atoms in the file.

Evaluation. Your program will hopefully be evaluated for correctness using automatic grading in the curator. The GTA will then look at your programs for programming style.