CSCI–4150 Introduction to Artificial Intelligence, Fall 2005
Assignment 6 (134 points), out Thursday November 3, see below for due dates

Notes on this assignment

This assignment includes one problem on Logic (since we skipped Assignment 5), but the main focus is on learning decision trees.

- Problem 1 is due on Thursday November 10. *No late submissions accepted!*
- The rest of the assignment is due on Thursday November 17 and is subject to the regular late policy.
- As usual, there will be a web page with support code, as well as a discussion group on WebCT.

A. Problems

1. (4 points) Take the survey on WebCT. Your responses will be used to form a data set that you will use for Problem 6. (Your identity will not be part of your response in the data set, of course.)
2. (16 points, written) This problem explores using FOL to express uniqueness of objects.
   (a) Write FOL sentences to express the following general facts:[1]
      - A parent is either a mother or a father.
      - A person has only one father.
      - A person has only one mother.

      Use the predicates `Parent(x,y)`, `Father(x,y)`, and `Mother(x,y)` (where the meaning of these predicates is, for example, `x` is the parent of `y`). See the "Equality" section on page 253 for some idea on how to write these sentences.

   (b) Form a knowledge base by translate these sentences into conjunctive normal form (CNF) and adding the following specific facts:
      - Bob is Carl's father.
      - Alice is Carl's mother.

   (c) Do a refutation proof using the resolution inference rule and the set of support strategy to show:
      - Carl has no other parent besides Alice and Bob.

      You should be able to do this without the demodulation and paramodulation rules (p. 304).
3. (30 points) Write the procedure:

   ```
   (learn-dtree training-data attribute-names)
   ```

   which returns a decision tree learned from the training data using the algorithm covered in class (also in the text) with the "greatest information gain" heuristic. Scheme representations for decision trees and training data are described in the remainder of this assignment handout. Support code will be provided to do some of the more mundane data manipulation tasks.
4. (20 points) Write the procedure:

   ```
   (missing-learn-dtree training-data attribute-names)
   ```

   which learns a decision tree from training data with missing attribute values. Your procedure should "split" examples with missing attribute values and send the appropriate fractions of each example down all branches of the tree.

[1] To keep things simple, we will often use a "kinship" domain that assumes that everyone forms nice heterosexual families, that no one ever gets divorced, and that no one ever has children out of wedlock.

5. (20 points) Write the procedure:

   ```
   (create-binary-discretization c-attribute values training-data attribute-names)
   ```

   which should return a procedure that takes a single example and discretizes a single continuous attribute into a binary attribute with the two values in the list `values`. If the attribute value is missing (i.e., is the symbol `?`), it should not be changed.

6. (44 points, written) This problem asks you to learn some decision trees and analyze your results.

   (a) Use your `learn-dtree` procedure to learn a decision tree using `mushroom-data1`, and test the resulting decision tree using `mushroom-data2`. Report the number and percentage of examples correctly classified when you test your decision tree, and also give the learned decision tree.

   Has your decision tree overfit the training data? Justify your answer.

   (b) Run your `missing-learn-dtree` procedure using the voting data set. Learn a decision tree using voting-data1 and test it using voting-data2. Report the number and percent of examples correctly classified by your decision tree, and also give the learned decision tree.

   Has your decision tree overfit the training data? Justify your answer.

   (c) Take the class survey data set and turn it into training data with ”OS” as the goal predicate. Use your `create-binary-discretization` procedure to discretize the continuous-valued attributes of the data set. Report the threshold and symbols used for discretizing each attribute.

   Learn a decision tree (using your `missing-learn-dtree` procedure) from the discretized version of the data set. Include the resulting decision tree in your writeup.

   (d) Try to learn something interesting from the class survey data set using other fields as the goal predicate. You may want to discretize a continuous-valued field (into two or more discrete values) in order to use it as a goal predicate. (You can manually choose threshold(s) for this.) Give the details of what you did, show the resulting decision tree, and explain why your result is interesting.

B. Scheme representations

B.1 Decision trees

A *decision tree* will be represented in Scheme as either:

- a symbol which is a value for the goal predicate, or
- a list of the following form:

```
(<attribute-name> (<attribute-value-1> <decision-tree-1>)
                  ...
                  (<attribute-value-n> <decision-tree-n>))
```

For the “loan” example in the support code, a valid (but not necessarily good) decision tree is:

```
(define loan-dtree-example
  '(income (high no)
           (low (house (rent yes)
                       (own  no)))))
```

B.2 Training data

Training data consist of a list of training examples, and a *training example* is a list where:

- the first element is the value of the goal predicate (which can be any symbol, not just `yes` or `no`), and
- the second element is a list of the attribute values.

Any missing attribute value will be indicated by the symbol `?` We will also require a list of *attribute names* so we can refer to attributes by name.

Here is an example of training data (a subset of the "loan" example):

```
(define loan-names '(House Bills Income Credit))
(define loan-data-small
  '((No  (Own Late    High   Good))
    (Yes (Own On-time Medium Good))
    (No  (Own On-time High   Good))
    (No  (Own On-time High   Good))))
```

Note that the goal predicate is not explicitly named.

B.3 Weighted training data

You will also make use of *weighted training data* for learning on data sets with missing attribute values. The only difference between weighted training data and regular training data is that each training example has a number (the weight) inserted as the first element. For example:

```
(define weighted-loan-data-small
  '((1.0 No  (Own Late    High   Good))
    (1.5 Yes (Own On-time Medium Good))
    (1.2 No  (Own On-time High   Good))
    (0.7 No  (Own On-time High   Good))))
```

There is one caveat: all training examples must be weighted. You cannot mix weighted and unweighted training examples in training data.

C. Support code

C.1 Procedures for handling training data

- `(split-tdata training-data attribute-names attribute)`

  This function divides the training data into groups according to the specified attribute. For example, using the training data above:

  ```
  (split-tdata loan-data-small loan-names 'income)
  ;Value: ((medium ((yes (own on-time medium good))))
  ;        (high ((no (own on-time high good))
  ;               (no (own on-time high good))
  ;               (no (own late high good)))))
  ```

  This procedure returns a list of what I refer to as *splits*. Each split is a list whose first element is a value of the attribute and whose second element is a list containing a subset of the training data which all have that value for the given attribute.

Note that in the example above, there is no split generated for the income attribute value "low" because the training data do not have an example with this value. See the "Implementation notes" section for discussion of this issue.

This procedure also works with weighted training data. For example

```
(split-tdata weighted-loan-data-small loan-names 'bills)
;Value: ((on-time ((.7 no (own on-time high good))
;                  (1.2 no (own on-time high good))
;                  (1.5 yes (own on-time medium good))))
;        (late ((1. no (own late high good)))))
```

- `(tally-tdata training-data)`

  This procedure counts the number of examples for each value of the goal predicate. It returns a list of clauses. Each clause is a list where the first element each is the value of the goal predicate, and the second element is the number of examples with that value.

  For example,

  ```
  (tally-tdata loan-data-small)
  ;Value: ((yes 1) (no 3))
  ```

  Do not assume that the goal predicate will always have the values "yes" and "no"! Also do not assume that there will always be only two values of the goal predicate!

  Like the `split-tdata` procedure, if there are no examples with a given goal predicate value, that value will not appear in the tally.

  This procedure will also handle weighted training data. For example:

  ```
  (tally-tdata weighted-loan-data-small)
  ;Value: ((yes 1.5) (no 2.9))
  ```

- `(pick-majority tally)`

  Given a tally (as returned by the `tally-tdata` procedure), this procedure returns the majority value. If there is a tie, it returns the first instance it finds. For example:

  ```
  (pick-majority '((yes 1) (no 3)))
  ;Value: no
  ```

C.2 Procedures for testing your decision trees

- `(classify example decision-tree attribute-names default-value)`

  This function returns a classification for the example determined by the given decision tree. Note that an "example" is just a list of attribute values. If an attribute value not in the decision tree is encountered, then it returns the default-value.

  For example:

  ```
  (classify '(rent late high good) loan-dtree-example loan-names 'No)
  ;Value: yes
  ```

- `(test-dtree decision-tree training-data attribute-names)`

  This function takes a decision tree and a set of training data. From the training data, it creates a list of examples (i.e. just the attribute values) and a list of correct classifications. It classifies all the examples using the decision tree, compares the results to the correct classifications, and reports the results.

C.3 Other support code procedures

- `(log2 x)`

  Takes the logarithm to base 2. Signals an error if you try to take the logarithm of a nonpositive number.

- `(print . args)`

  Calls `display` on each argument. (Takes 0 or more arguments.)

C.4 Useful built-in Scheme procedures

The following are either standard Scheme procedures or procedures tha come with MIT/GNU Scheme.

- `(sort Lst proc)`

  Sorts the given list. The second argument must be a procedure of two arguments that returns true if the first argument should come before the second argument. This procedure should not return `#t` if the arguments are "equal". (See the MIT/GNU Scheme reference manual, Section 7.9, for more information.)

  For example:

  ```
  (sort '(43 8 2 88 9) <)
  ;Value: (2 8 9 43 88)
  ```

- `(assoc key a-list)`

  The second argument a-list must be an *association list*. The elements of an association list are themselves lists where the first element is the key. The `assoc` procedure takes the given key and finds the first association in a-list that matches (using `equal?`) the key. If no matching key is found, it returns the empty list.

  For example:

  ```
  (assoc 'red '((blue 7) (red 6) (green 5)))
  ;Value: (red 6)
  ```

  Note that the splits returned by split-tdata have the form of an associaton list.

- `(delete el Lst)`

  Deletes all copies of the given element `el` from the list `Lst`. The `equal?` predicate is used to determine if the element matches `el`. Actually, this doesn't delete elements from the list — it makes a copy of the list without any matching elements.

  For example:

  ```
  (delete 4 '(1 3 5 4 2.5 4.0 9 4))
  ;Value: (1 3 5 2.5 4. 9)
  ```

  Note that the value 4.0 was not removed because it is not `equal?` to 4.

D. Implementation notes

D.1 Differences from the text's algorithm

The basic decision tree learning algorithm you should implement for this assignment is slightly different than the algorithm in our text. The difference is in how the decision tree will handle examples that have attribute values not seen in the training data.

The algorithm in the text handles this by making a recursive call to the `learn-dtree` procedure with zero examples. This returns a decision (sub)tree that consists of a leaf node: the default classification.

The way the support code for this assignment is structured, you should never make a recursive call to to your `learn-dtree` procedure when there are no examples left in a given branch. Instead, the `classify` procedure returns the default value if it encounters an attribute value not in the decision tree.

The reason for this difference is to simplify your code. In order to implement the text's algorithm, you would have to know all the values of each attribute, and they would have to be passed down from one recursive call to the next. Leaving this situation to be handled by the `classify` procedure means that only the attribute names need to be passed.

As an example, consider the decision tree in Figure 18.8 of the text. My solutions, run on the same training data produce the decision tree:

```
(learn-dtree restaurant-data restaurant-names)
;Value: (patrons (none no)
                (full (hungry (no no)
                              (yes (type (burger yes)
                                         (italian no)
                                         (thai (fri (no no)
                                                    (yes yes)))))))
                (some yes))
```

Notice that there is no `french` value handled under type. This is because the french restaurants in the training data were classified under other cases of the decision tree. (One training example had `some` patrons; in the other, patrons was `full`, and `hungry` was `no`.)

D.2 Suggestions

- Because the decision tree representations, tallies, and splits can be confusing, I strongly suggest using simple accessor functions to access information from these data structures.
- Take advantage of the fact that Scheme is interpreted and test your procedures from the bottom up — make sure your lower level functions are doing the right thing before you go on to the higher level functions!

D.3 Data sets

There will be several data sets available for you to test your procedures, including the data set we are collecting from the class. See the Assignment 6 web page for details.