CSCI 4150 Introduction to Artificial Intelligence, Fall 1999
Assignment 7: out Tuesday November 9, due midnight, evening of Friday November 19

In this assignment you will write a program to learn decision trees using (a slight variation of) the algorithm described in class (and in our text). You will not have to deal with missing data in the examples, although for the final part of this assignment, you will need to discretize some continuous valued attributes.

1 Learning decision trees

You are to write the function:

```
(learn-dt training-data)
```

which should return a decision tree. The representation I have chosen for decision trees and training data is described in subsequent sections. I am providing support code to do some of the more mundane data manipulation tasks. You may structure the `learn-dt` procedure however you like, but I will make a few suggestions. I am also providing a few data sets for you to test your code on.

There is an additional problem which deals with a domain that has continuously valued attributes. For this problem, you will write an additional procedure that discretizes these attributes.

Please see the web page for this assignment for the support code and any errata/updates that may come up.

Scheme representations

Training data and examples

An "example" consists of a list of attribute values. For a given domain, all the examples are the same length (same number of attributes), the attributes are in the same order, and there are no missing attribute values. An attribute value can be a symbol or a number.

"Training data" is a list of training examples. A "training example" is a list whose first element is the value of the *goal predicate* for the training example, and the second element is the "example". The value of the goal predicate can be any symbol! (Not just "yes" or "no", and there can be more than two values.)

For example, a small set of training data based on the restaurant example in our text is:

```
(define small-td
  '((yes (yes no  no yes some $$$ no yes french upto10))
    (no  (yes no  no yes full   $ no no  thai   upto60))
    (yes (no  yes no no  some   $ no no  burger upto10))))
```

The first element of each training example is the value of the goal predicate "Will wait"; the second element is the "example" which is the list of values for the attributes: alternate, bar, friday, hungry, patrons, etc.

Decision trees

A decision tree is either:

- a value of the goal predicate
- a list of the following form:

```
(<attribute-number> (<attribute-value-1> <decision-tree-1>)
                    (<attribute-value-2> <decision-tree-2>)
                    ...
                    (<attribute-value-n> <decision-tree-n>))
```

where attribute-number is an integer specifying that attribute's position within an "example" (0 indexed).

For example, for the restaurant example, a valid decision tree is:

```
(4 (none no)
   (some yes)
   (full (3 (no yes)
            (yes no))))
```

where attribute 4 refers to "patrons" and attribute 3 refers to "hungry".

Support code

I have provided a few function that you may find useful for writing your `learn-dt` procedure.

Handling training data

- `(split-td td attribute-no)`

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

  ```
  (split-td small-td 0)
  ;Value: ((no  ((yes (no yes no no some $ no no burger upto10))))
           (yes ((no  (yes no no yes full $ no no thai upto60))
                 (yes (yes no no yes some $$$ no yes french upto10)))))
  ```

  To generalize, this function returns a list of "splits" where each split is a list whose first element is a value of the attribute and whose second element is a set of training data which all has that value for the given attribute.

- `(tally-td td)`

  This function tallies up how many instances there are of each value of the goal predicate, returning a list of lists; the first element of each sublist is the value of the goal predicate, and the second element is the number of training examples with that value.

  For example,

  ```
  (tally-td small-td)
  ;Value: ((no 1) (yes 2))
  ```

  Do not assume that the goal predicate will always have the values "yes" and "no"!

Testing your decision trees

- `(classify example dtree . default-value)`

  This function takes an example and a decision tree and returns the classification for the example. An optional third argument specifies a default value to return in case `classify` encounters an attribute value it has not seen before.

- `(classify-list example-list dt)`

  This function takes a list of examples and a decision tree and classifies all the examples, returning a list of the goal predicate values.

- `(test-dt dt test-td)`

  This function takes a decision tree and a set of training data. From the training data, it creates a list of examples 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.

- `(make-readable-dt dt attribute-names)`

  This function replaces the attribute numbers by their names by looking them up in a list of names. See below for an example.

Miscellaneous utility functions

- `(remove-element e alist)`

  Removes the first occurence of `e` in `alist`, returning a (mostly) newly created list.

- `(zero-to-n-1 n)`

  Creates a list of integers from 0 to n-1 inclusive.

Differences from the text's algorithm

The main difference in the algorithm you should implement for this assignment versus the algorithm in our text is the handling of cases where there are no examples left in a recursive call.

The algorithm in the text handles this while creating the decision tree, but for this assignment, you should ignore this case; this situation is handled when evaluating examples using the `classify` function. The reason for this modification is simplicity — in order to implement the text's algorithm, you would have to know all the values of each attribute at every recursive call. What I am suggesting is that you simply handle the cases that are present in the training data. If any unknown attribute value comes up, then the `classify` procedure will return a default value.

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

```
(make-readable-dt (learn-dt restaurant-td) restaurant-anames)
;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".)

Like the text's algorithm, you should choose the "majority" goal attribute value when there are no attributes left to split a mixed set of trainind data.

Data sets

I have provided three data sets for you to test your procedure:

- `play-tennis-tdata`,`play-tennis-examples`,`play-tennis-answers`,`play-tennis-anames`

  Deciding whether to play tennis based on the (weather) outlook, humidity, and the wind. There is also `play-tennis-dtree` which is a decision tree for this domain; however, it is not the same as the one you will generate from the training data.

- `restaurant-tdata`, `restaurant-anames`

  The restaurant example from our text.

- `mushroom-anames`, `mushroom-tdata`, `mushroom-test`

  A database of mushrooms. All the attribute values are abbreviated to a single letter; the goal predicate is whether the mushroom is poisonous (p) or edible (e). You have both training data and test (training) data for this data set.

2 Discretizing continuous-valued atttributes

For this part, you will write a function:

```
(discretize training-data)
```

which takes a training data set from the `census-tdata` training data and discretizes the continuous-valued attributes. The object of this problem is to determine whether someone makes more or less than $50,000 per year based on certain census data.

The goal predicate is the person's income, either `>50K` or `<=50K`. The attributes in this data set are:

- age: continuous.
- workclass: discrete
- fnlwgt: continuous.
- education: discrete
- education-num: continuous
- marital-status: discrete
- occupation: discrete
- relationship: discrete
- race: discrete
- sex: discrete
- capital-gain: continuous
- capital-loss: continuous
- hours-per-week: continuous
- native-country: discrete

More information on these attributes will be avalable online.

You may do anything you wish in your `discretize` function so long as it returns a training example that can be used by my implementation of `learn-dt`. The most straightforward approach would be to discretize the continuous-valued attributes somehow, but you might also omit certain attributes.

Your objective in this problem is to achieve the highest possible number of correct classifications. I will provide a few test (training) data sets, but ultimately, I will use your `discretize` procedure to process the given training data set, run my `learn-dt` on the resulting training data, and then test the resulting decision tree on unreleased test data.

Most of the points of this question will be based on a working discretize function, but there will also be some performance component.