

1 Learning decision trees (written)

(35 points) Given the following training data:

Example No.	Outlook	Temperature	Humidity	Wind	Play tennis?
1	Sunny	Hot	High	Weak	No
2	Sunny	Hot	High	Strong	No
3	Overcast	Hot	High	Weak	Yes
4	Rain	Mild	High	Weak	Yes
5	Rain	Cool	Normal	Weak	Yes
6	Rain	Cool	Normal	Strong	No
7	Overcast	Cool	Normal	Strong	Yes
8	Sunny	Mild	High	Weak	No
9	Sunny	Cool	Normal	Weak	Yes
10	Rain	Mild	Normal	Weak	Yes
11	Sunny	Mild	Normal	Strong	Yes
12	Overcast	Mild	High	Strong	Yes
13	Overcast	Hot	Normal	Weak	Yes
14	Rain	Mild	High	Strong	No

where the goal predicate is *Play tennis?*, calculate the information gains that would result as a result of splitting the data for each of the four attributes. Show and explain your calculations.

2 Learning decision trees (programming)

(35 points) For this problem, you are to write the function:

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

which should return a decision tree. The representations I have chosen for decision trees and training data are 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.

You will turn in your code from this problem to the web tester. Please see the Assignment 5 information page (off the course home page) for details.

Scheme representations

Training data and examples

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. We will also require a list of *attribute names*.

As an example, here are the first four training examples from the first problem made into a training data set.

```
(define tennis-names
  '(Outlook Temperature Humidity Wind))
(define tennis-data-small
  '((No (Sunny Hot High Weak))
    (No (Sunny Hot High Strong))
```

```
(Yes (Overcast Hot High Weak))
(Yes (Rain Mild High Weak)))
```

Decision trees

A decision tree is either 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 tennis example, a valid decision tree is:

```
(define tennis-dtree-example '(humidity (high yes)
                                       (normal (outlook (sunny no)
                                                       (rain yes)))))
```

Support code

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

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, for the training data above:

```
(split-tdata tennis-data-small tennis-names 'humidity)
;Value: ((mild ((yes (rain mild high weak)))
          (hot ((yes (overcast hot high weak))
                (no (sunny hot high strong))
                (no (sunny hot high weak)))))
```

Generally speaking, 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 subset of the training data which all have that value for the given attribute.

- `(tally-tdata training-data)`

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-tdata tennis-data-small)
;Value: ((yes 2) (no 2))
```

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

- `(pick-majority tally)`

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

```
(pick-majority (tally-tdata restaurant-data))
;Value: no
```

Testing your decision trees

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

This function returns the classification for the example; if it encounters an attribute value that is not in the decision tree, then it will return the default-value. For example:

```
(classify (second (fourth tennis-data-small))
          tennis-dtree-example tennis-names 'attribute-value-not-found)
;Value: yes
```

```
(classify '(overcast hot normal weak)
          tennis-dtree-example tennis-names 'attribute-value-not-found)
;Value: attribute-value-not-found
```

Note that an example is just a list of attribute values.

- `(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 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.

Miscellaneous utility functions

- `(remove-element e alist)`

Removes the first occurrence of `e` in `alist`, returning a newly created list.

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. For this assignment you should simply handle the cases that are present in the training data. If any unknown attribute value comes up, then the `classify` procedure will return the default value.

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".)

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 training data.

Data sets

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

- `tennis-data`, `tennis-names`

The example from Problem 1.

- `restaurant-data`, `restaurant-names`

The restaurant example from our text.

- `mushroom-data1`, `mushroom-data2`, `mushroom-names`

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). Use one data set for training and the other for testing.

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. Also, 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!