Assigned: Monday 7 November 2011

Due: Monday 14 November 2011, 11:59 pm

Objectives:

- Understand decision tree learning and classification
- Learn and apply techniques for analyzing classifiers

Collaboration: This homework assignment will be completed in pairs of your choice.

Introduction

Decision trees are a very fast method of supervised learning for a classification task. Learning is at worst quadratic in the size of the training data, but very often it is much closer to $O(n \log n)$ if the splits are well-balanced.

In this assignment, you will complete a decision tree learning algorithm and perform some experiments on its behavior for a real task. The book gives the pseudo code for DECISION-TREE-LEARNING (Fig 18.5), from which you will craft a Scheme implementation. You will also use the resulting decision tree to *classify* new, unlabeled instances. You can then use these methods in combination to measure the accuracy of the learned decision tree on previously unseen instances for various amounts of training data.

Our application is to determine, based on some simple tests of physical characteristics, whether a mushroom is edible or poisonous according to the Audubon Society Field Guide. There are over 8000 training examples using 21 features, which have been downloaded from the UCI Machine Learning repository.[1]

Background

Code

You may obtain the starter code for this assignment on the MathLAN from the directory

```
~weinman/courses/CSC261/code/dtree
```

Here is an overview of the files it contains.

`restaurant.scm` Contains a sample training set–the restaurant data from Figure 18.3. Two elements will be needed for creating a decision tree, the list of examples (a set of attributes and values paired with the label for each example), as well as the set of attributes and the list of values they may take on. The latter will be useful for the process of constructing the tree.

`dtree.scm` Contains several base methods that you will use to implement the decision tree learner and classifier. While you are welcome to study the implementations, the examples below, in conjunction with the documentation, should tell you what you will need.

`mushroom.scm` Contains routines for loading the mushroom examples and attributes, which reside in `mushrooms.txt` and `mushroom-attribs.txt`. Plain-english explanations of the terse attributes in `mushroom-attribs.txt` are given in `mushroom-info.txt`.

`analysis.scm` Contains routines for helping you perform some analysis. Namely, splitting the example data into train/test subsets and selecting a (smaller) random subset of examples for training.

`assignment.scm` Contains the documentation for the procedures you will write.

The remaining files are mostly ancillary and loaded by the ones mentioned above when needed.

[1] http://archive.ics.uci.edu/ml/datasets/Mushroom

Examples

Training Examples

How are is our training data structured? An **instance** is an association list whose keys are attributes, and whose values (the `cdr` of each item in the alist) are the value taken on by that attribute in the instance. For example, the first instance (labelled X_1 in AIMA) takes following the Scheme representation. (Note that each entry in the association list is a single cons cell, rather than a list.)

```
> (load "restauraunt.scm")
> (cdar restaurant-examples)
(("Alt" . #t) ("Bar" . #f) ("Fri" . #t) ("Hun" . #t) ("Pat" . "Some")
 ("Price" . "$$$") ("Rain" . #f) ("Res" . #t) ("Type" . "French")
 ("Est" . "0-10"))
```

This is just one instance. How might we know what the other possible values of the attributes might have been? For this we keep track of all the attributes in another association list. An example of this may be seen in the `restaurant-attributes` variable from `restaurant.scm`. In the attributes alist, the key is the attribute name, and the value is a list of all the values that attribute may take on. Note that you can get a list of all the attribute names quite easily with `map`

```
> (define candidates (map car restaurant-attributes))
> candidates
("Alt" "Bar" "Fri" "Hun" "Pat" "Price" "Rain" "Res" "Type" "Est")
```

and you can find the list of possible attribute values using `cdr` and `assoc`

```
> (cdr (assoc "Type" restaurant-attributes))
("French" "Thai" "Burger" "Italian")
```

If we are to train a classifier, we shall need labels for some instances. Thus, an **example** is a pair, whose `car` is the training label and whose `cdr` is an instance. If the example instance above were defined as `X1`, then a corresponding example for it would be constructed as

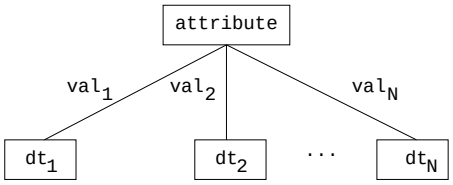
```
(cons #t X1)
```

to indicate that the label is `#t` (i.e., will wait). Note that because an instance is a list, an example is a list, too.

Decision Tree Representation

How can we represent a decision tree in Scheme? It is actually quite simple, not much more complicated than thinking about how a list is defined. Both are recursive data structures. That means, when giving a definition, we have a recursive case and a base case. The base case for a list is simply the empty list (`null`), while the recursive case is a value followed by another list. A decision tree has a simple base case: we need to make no tests and simply emit a classification decision. In this case, a decision tree may be a valid class label (e.g., `#t` for "will wait" in the restaurant problem, or `#\p` for "poisonous" in the mushroom problem). The basic construction, therefore, looks like the following

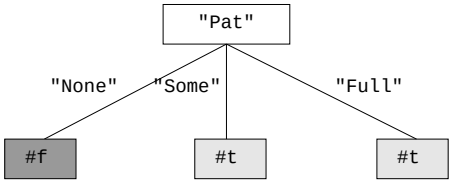
```
(cons attribute
      (list  (cons val1 dt1)
             ...
             (cons valN dtN)))
```



where $\texttt{val}_1$ through $\texttt{val}_N$ are the domain values for `attribute` and $\texttt{dt}_1$ through $\texttt{dt}_N$ are the corresponding (recursively built) decision trees to apply to the case when `attribute` has value $\texttt{val}_i$.

Let us take an abridged version of the decision tree found in Figure 18.6 (p. 702). In our restaurant representation, the attribute corresponding to the query *Patrons?* is `"Pat"`. This attribute has three possible values, each paired with yet another decision tree. We represent all this in Scheme as a `cons` cell (or pair) whose `car` is the attribute, and whose `cdr` is an association list. This association list has the possible attribute values as its keys, and decision trees as the associated values. If we decided instead to always wait when the restaurant is full, we might have the following as our decision tree.

```
(cons “Pat”
      (list (cons “None” #f)
            (cons “Some” #t)
            (cons “Full” #t)))
```



Scheme would display this as the following.

```
(“Pat”
  (“None” . #f)
  (“Some” . #t)
  (“Full” . #t))
```

What if we need to apply more than one test? We then have a recursive case: rather than a simple classification (e.g., `#t` or `#f`), each decision tree $\texttt{dt}_i$ must be the same type of structure. That is, it also indicates an attribute to test, and then for each value in the attribute's domain, another decision tree to use. If we continued our way down the tree, after discovering that the restaurant is full, we may then wish not to decide to wait, but to ask whether we are hungry. Thus, rather than having the simple decision `#t` in the pair `(“Full” . #t)`, we would need yet another decision *tree*. If we elected to wait only if we were not hungry, our decision sub-tree (used only when the restaurant is full) would be built as follows.

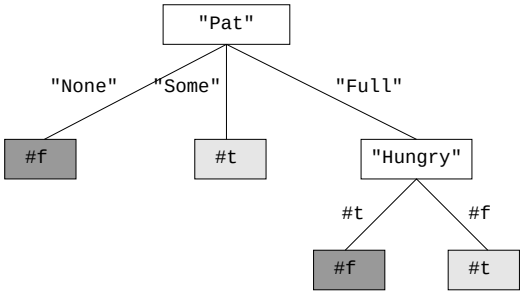
```
(define subtree-when-full
  (cons “Hun”
        (list (cons #t #f)
              (cons #f #t))))
```

Then we simply use this decision tree in place of deciding to wait when the restaurant is full.

```
(cons “Pat”
      (list (cons “None” #f)
            (cons “Some” #t)
            (cons “Full” subtree-when-full)))
```

Scheme would display this result as the following.

```
("Pat"
  ("None" . #f)
  ("Some" . #t)
  ("Full"
   "Hun"
     (#t . #f)
     (#f . #t)))
```



We can repeat the recursive nesting as many times as we wish, so long as we haven't run out of attributes to test along the path. Rather than belabor this construction further, let's continue by looking at how one actually puts such a tree together.

Building Blocks for Learning

Your decision tree learning method will take three parameters,

- the list of examples,
- the association list of attributes (also giving the domain of each attribute), and
- a default label to give examples when tests are exhausted.

In order to complete the implementation, a few other ingredients will be helpful. Let us trace through the DECISION-TREE-LEARNING algorithm.

First, note that there is a subtle difference between the type of parameters to DECISION-TREE-LEARNING and what you will implement in the Scheme procedure `decision-tree-learning`. The pseudocode takes only the set of available attributes that may be tested, while the Scheme procedure takes the association list of attributes and their domains (possible value). We shall point out the importance of this difference as we progress.

For the first case, it is easy to tell whether the list of examples is empty.

In the second case, the pre-written predicate `(all-same-label? examples)` will tell you whether all the examples have the same classification.

Assuming we have kept (somewhere) a list of remaining candidate attribute keys, it is easy to test whether that candidate list is empty. In that case, the procedure `(plurality-value examples)` will return the class label in `examples` that occurs with the greatest frequency.

If none of these three cases occur, then we must recursively build a decision tree. We begin by selecting an attribute to test the value of. You will do this via the `(choose-attribute examples candidates attributes)` procedure, which you will write. Using `examples`, this procedure selects among the attributes in the list `candidates` to determine which attribute is the best to split on. In otherwords, it must loop over the candidate attributes to find the one having the largest IMPORTANCE. We also pass in the `attributes` association list to `choose-attribute` so that the procedure may know the domains (possible values) of the `candidates`. More is said about this in the next section.

Once we know `best-attrib`, the value returned by `choose-attribute`, we can use this attribute as the first element in the list that forms our decision tree. To build the rest of the list, we have to loop over the all the values in the domain of `best-attrib`, adding pairs containing the value and the decision tree associated with that value. (Note that we saw above how to get the values in the domain of an attribute using `cdr` and `assoc`.)

As we're adding the branches that form our decision tree, we need two additional capabilities. First, we need to find the the subset of `examples` that have a particular value for the best attribute so that we may recursively build the decision tree on only those examples. Fortunately, the procedure `(filter-examples-by-attribute-value examples attribute value)` does just that for you. You will also need remove the best attribute from the list of candidate attributes. You can do this with the procedure `(filter-list val lst)` found in `general.scm`.

Finally, you are reminded that your `decision-tree-learning` procedure takes the association list of attributes and their domains, yet your recursive call to build a tree requires only a narrowing set of candidate attributes. Thus, in implementing the learning algorithm you are advised to use a helper procedure or (even better) a simple named `let` to iteratively/recursively bind the smaller sets of examples and candidate attributes. In this way, you can avoiding to pass along the full attribute/domain association list with every recursive call.

Choosing an Attribute

As of now, we've sideaccuracy-stepped how to choose an attribute. The textbook describes the **information gain** as the difference between an existing information content (due to Claude Shannon) and the information resulting from applying some test. We will use this as our metric, and it has already been implemented for you as `(information-gain examples candidate attributes)` where `candidate` is some attribute and `attributes` is the association list giving the domains of all our attributes. For example,

```
> (information-gain restaurant-examples "Est" restaurant-attributes)
0.20751874963942185
> (information-gain restaurant-examples "Pat" restaurant-attributes)
0.5408520829727552
```

So the estimated wait gives us about one fifth of a bit of information, while the number of patrons more than doubles that at half a bit, which probably explains why this is the leading attribute test.

Classification

How do you classify an instance? Now that you know the recursive structure of a decision tree, it is straightforward. If the decision tree starts with a `cons` cell (which you can test using the predicate `pair?`), then you know you need to apply an attribute test. You will need to

- get the attribute test specified in the decision tree,
- get the value of that attribute in the instance (using `assoc`),
- follow the branch of the decision tree having that value to get the next decision tree (again using `assoc`), and then
- recursively apply the classification routine on the next decision (sub)tree.

Otherwise, the decision tree is simply a classification value, which may be returned.

Putting it Together

Using the restaurant example, here is a a sample decision tree

```
> (decision-tree-learning restaurant-examples restaurant-attributes #t)
("Pat"
  ("None" . #f)
  ("Some" . #t)
  ("Full"
     "Hun"
       (#f . #f)
       (#t
         "Type"
           ("French" . #f)
           ("Thai"
              "Rain"
                (#t . #t)
                (#f . #f))
           ("Burger" . #t)
           ("Italian" . #f))))
```

Note that this is substantially the same as the one given in the text (Figure 18.6). The only difference is in the subtree for a Thai restaurant. There are but two instances in our example list that are full when we are hungry and at a Thai restaurant. In either of these cases, both the rain and Friday attributes classify them perfectly. Our learning algorithm chose one, while the textbook uses the other.

If we defined the tree above as `restaurant-tree`, we could then use it to classify an instance

```
> (decision-tree-classify restaurant-tree (cdar restaurant-examples))
#t
```

Assignment

There are four interrelated tasks on this assignment. If you get stuck on any of them, you should continue working on the others until you are able to get help with wherever you may be having problems or until you have taken a break long enough that you are able to see your errors. The worst strategy is to spend all your time on one problem and turn in nothing for the others.

Problem 1: `choose-attribute`

Using the procedure `information-gain` described above (and documented in `dtree.scm`), write the `choose-attribute` procedure documented in `assignment.scm`. A pseudo-code description of this algorithm is shown below.

Algorithm 1 Choosing an attribute.

function CHOOSE-ATTRIBUTE(*examples*, *candidates*, *attributes*) **returns** an attribute from *candidates*
  inputs: *examples*, a set of examples
    candidates, a list of candidate attributes to test
    attributes, an association list of all attributes with their domains

  maxgain ← INFORMATION-GAIN(*examples*, FIRST(*candidates*), *attributes*)
  best ← FIRST(*candidates*)
  for each *attrib* in REST(*candidates*) **do**
    gain ← INFORMATION-GAIN(*examples*, *attrib*, *attributes*)
    if *gain* > *maxgain* **then**
      maxgain ← *gain*
      best ← *attrib*
  return *best*

Problem 2: `decision-tree-learning`

Using the methods described above, implement the `decision-tree-learning` procedure documented in `assignment.scm`. To assist you, a summary of the conversion from the pseudo code (Figure 18.5) to the scheme procedures is given below.

Pseudo-Code	Scheme
all *examples* have the same classification	`(all-same-label? examples)`
PLURALITY-VALUE(*examples*)	`(plurality-value examples)`
$\arg\max_{a \in attributes}$ IMPORTANCE$(a, examples)$	`(choose-attribute examples candidates attributes)`
values of v_k *of* A	`(cdr (assoc best attributes))`
$\{e : e \in examples$ **and** $e.A = v_k\}$	`(filter-examples-by-attribute-value examples best v)`
attribs - A	`(filter-list best candidates)`

Keep in mind that when all the initial tests fail, you will be producing a value that has the following format:

```
(cons attribute
      (list  (cons val1 dtN)
             ...
             (cons valN dtN)))
```

where $\texttt{val}_1$ through $\texttt{val}_N$ are the domain values for `attribute` and $\texttt{dt}_1$ through $\texttt{dt}_N$ are the corresponding (recursively built) decision trees for examples with the attribute having that value. How you choose to go about constructing this is up to you (e.g., `map` versus looping with a named `let`).

Problem 3: `decision-tree-classify`

Implement the `decision-tree-classify` procedure documented in `assignment.scm`. Note the structure of the procedure outlined in the examples above.

Problem 4: Analysis

You now have everything you need to train and test your supervised learning classifier. Note that `dtree.scm` includes the procedure `(decision-tree-accuracy` *`decision-tree examples`*`)` that allows you to test the accuracy (a number between 0 and 1) on some examples. The examples do *not* need to be the same ones used to train the decision tree.

Part A

Display the Scheme value and draw the tree learned by your algorithm on *all* of the mushroom data. What is the accuracy on the data? How does the size of the tree compare with what you expected?

Part B

Here you will apply the methodology for assessing a learning algorithm using the mushroom data. Namely, this requires us to:[2]

1. Collect a large set of examples (which we have already done).
2. Divide the examples into two disjoint sets, one for training and another for testing.
3. Apply `decision-tree-learning` to a training set, generating a hypothesis function h.
4. Measure the accuracy on the test set made by h
5. Repeat steps 2 to 4 for different sizes of training sets and different randomly selected sets of each size.

The procedure `(run-trials` *`examples attributes default num-trials`*`)` in `analysis.scm` does this sort of analysis, measuring the error rate on a *single* test set with randomly selected training subsets of lengths 5,10,15,20,25,...,85,90,95,100. It reports the average error rate over several trials for each length. Use this procedure and display your learning curve in a table or (better!) a graph, much like Figure 18.7 (p. 703).

What to turn in

Your submission should include the following

- Your completed `assignment.scm`
- A short driver program generates the data for Problem 4
- A single PDF containing (merged)
  - Your Scheme files
  - A transcript of your driver program's output
  - Your graphical representation of the decision tree from Problem 4(A)
  - Your table or graph of Problem 4(B)

[2] Adapted from ("Assessing the performance of the learning algorithm," AIMA 2/e pp. 660-661.