**Assignment 6: Decision Trees**

**Assigned:** Monday 23 November 2009

**Due:** Friday 4 December 2009

**Topics:** Supervised learning, decision trees,

**Objective:** Understand decision tree learning and classification, learn how to analyze classifiers

**Expected Time:** 4-8 hours

**Collaboration:** This homework assignment may be completed individually or in pairs.

**Submission:** Follow the instructions for submitting programs via P-Web and handing in a printed copy of source code. Your non-programming components should be typed, nicely formatted, and feature a logical organization, complete sentences, and proper grammar, spelling, and punctuation. Only one submission (both paper and digital) per group is necessary. Do not turn in any code that was provided for you. Only submit code you have written.

**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), which you will craft a Scheme implementation of. 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.

¹[http://archive.ics.uci.edu/ml/datasets/Mushroom](http://archive.ics.uci.edu/ml/datasets/Mushroom)
mushroom.scm Contains routines for loading the mushroom examples and attributes, which reside in 
mushrooms.txt and mushroom-attrs.txt.

analysis.scm Contains two 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. You may try to do your work in any Scheme environment you like, but DrScheme using the PrettyBig language is the only one I have used and will support.

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.)

\[
\begin{align*}
\text{'} & (\text{load} \ "\text{restaurant.scm}" ) \\
& (\text{cdr} \ \text{restaurant-examples}) \\
& ("\text{Alt}" . \#t) \ ("\text{Bar}" . \#f) \ ("\text{Fri}" . \#t) \ ("\text{Hun}" . \#t) \ ("\text{Pat}" . \ "\text{Some}"") \\
& ("\text{Price}" . \"$$\$$") \ ("\text{Rain}" . \#f) \ ("\text{Res}" . \#t) \ ("\text{Type}" . \ "\text{French}"") \\
& ("\text{Est}" . \ "0-10")
\end{align*}
\]

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 list, 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

\[
\begin{align*}
& (\text{define} \ \text{candidates} \ (\text{map} \ \text{car} \ \text{restaurant-attributes})) \\
& \text{candidates} \\
& ("\text{Alt}" \ "\text{Bar}" \ "\text{Fri}" \ "\text{Hun}" \ "\text{Pat}" \ "\text{Price}" \ "\text{Rain}" \ "\text{Res}" \ "\text{Type}" \ "\text{Est}")
\end{align*}
\]

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

\[
\begin{align*}
& (\text{cdr} \ (\text{assoc} \ "\text{Type}" \ \text{restaurant-attributes})) \\
& ("\text{French}" \ "\text{Thai}" \ "\text{Burger}" \ "\text{Italian}"")
\end{align*}
\]

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 X_1, 
then a corresponding example for it would be constructed as

\[
\begin{align*}
& (\text{cons} \ \#t \ X_1)
\end{align*}
\]

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

Training Data

To load the attributes and examples for the mushroom classification task, you may use the routines in 
mushroom.scm for the files mushrooms.txt and mushroom-attrs.txt.

\[
\begin{align*}
& (\text{load} \ "\text{mushroom.scm}" ) \\
& (\text{define} \ \text{mushroom-attributes} \\
& \ (\text{load-mushroom-attributes} \ "\text{mushroom-attrs.txt}")) \\
& (\text{define} \ \text{mushroom-examples} \\
& \ (\text{load-mushroom-examples} \ "\text{mushrooms.txt}" \ \text{mushroom-attributes}))
\end{align*}
\]
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

\[
(\text{cons \ attribute} \\
\text{\hspace{1cm} (list} \ (\text{cons \ val}_1 \ dt_1) \\
\hspace{2cm} \ldots \\
\hspace{2cm} \text{\hspace{1cm} (cons \ val}_N \ dt_N)))
\]

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

Let us take an abridged version of the decision tree found in Figure 18.6 (p. 658). In our restaurant representation, the attribute corresponding to the query Patrons? is “Pat”. This has three possible values, each with yet another decision tree. We represent this in Scheme as a cons cell (or pair) whose car is the attribute, and whose cdr is an association list. This list then has the attribute domain elements as its keys, and decision trees as the values. If we decided instead to always wait when the restaurant is full, we might have the following as our decision tree.

\[
(\text{cons “Pat”} \\
\hspace{1cm} (\text{list} \ (\text{cons “None”} \ #f) \\
\hspace{2cm} \text{\hspace{1cm} (cons “Some” \ #t) \\
\hspace{3cm} \text{\hspace{1cm} (cons “Full” \ #t))}))
\]

Scheme would display this as the following:

\[
(“Pat” \\
\hspace{1cm} (“None” . #f) \\
\hspace{2cm} (“Some” . #t) \\
\hspace{3cm} (“Full” . #t))
\]

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 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.

\[
(\text{define subtree-when-full} \\
\hspace{1cm} (\text{cons “Hun”} \\
\hspace{2cm} \text{\hspace{1cm} (list} \ (\text{cons #t #f) \\
\hspace{3cm} \text{\hspace{1cm} (cons #f #t))))))
\]

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, t is easy to tell whether the list of examples is empty. The predicate `(all-same-label? examples)` will tell you whether all the examples have the same classification.

Assuming we have kept a list of remaining candidate attribute keys, it is easy to test whether this list is empty. The procedure `(majority-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 is the best to split on. We pass in the `attributes` list 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-attribute`, the value returned by `choose-attribute`, we can use this 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-attribute`, 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

\[\text{examples}
\]
so that we may recursively build the decision tree on only those examples. Fortunately, the procedure \(\text{filter-examples-by-attribute-value examples attribute value}\) does just that for you. You will also need to remove the best attribute from the list of candidate attributes. You can do this with the procedure \(\text{filter-list val lst}\) found in \texttt{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 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 side-stepped how to choose an attribute. The textbook describes the \texttt{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 \(\text{information-gain examples candidate attributes}\) where \texttt{candidate} is some attribute and \texttt{attributes} is the association list giving the domains of all our attributes. For example,

\begin{verbatim}
> (information-gain restaurant-examples "Est" restaurant-attributes)
0.20751874963942185
> (information-gain restaurant-examples "Pat" restaurant-attributes)
0.5408520829727552
\end{verbatim}

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 \texttt{cons} cell (which you can test with \texttt{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 \texttt{assoc}),
- follow the branch of the decision tree having that value to get the next decision tree (using \texttt{assoc}), and
- recursively apply the classification routine on the next decision 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"
     "Hungry"
     ( #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 (25 points)

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 algo-

rhism is shown below.
Algorithm 1 Choosing an attribute.

function \textsc{Choose-Attribute}(\textit{examples, candidates, attributes}) returns an attribute from \textit{candidates}

inputs: \textit{examples}, a set of examples
\textit{candidates}, a list of candidate attributes to test
\textit{attributes}, an association list of all attributes with their domains

\[
\begin{align*}
\text{maxgain} & \leftarrow \text{Information-Gain}(\textit{examples, First(\textit{candidates}), attributes}) \\
\text{best} & \leftarrow \text{First(\textit{candidates})} \\
\text{for each} & \ \text{attrib in Rest(\textit{candidates}) do} \\
& \text{gain} \leftarrow \text{Information-Gain}(\textit{examples, attrib, attributes}) \\
& \text{if} \ \text{gain} > \text{maxgain} \ \text{then} \\
& \text{maxgain} \leftarrow \text{gain} \\
& \text{best} \leftarrow \text{attrib}
\end{align*}
\]

return \textit{best}

Problem 2: decision-tree-learning (35 points)

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

<table>
<thead>
<tr>
<th>Pseudo-Code</th>
<th>Scheme</th>
</tr>
</thead>
<tbody>
<tr>
<td>all \textit{examples} have the same classification</td>
<td>(all-same-label? \textit{examples})</td>
</tr>
<tr>
<td>\textsc{Majority-Value}(\textit{examples})</td>
<td>(majority-value \textit{examples})</td>
</tr>
<tr>
<td>\textsc{Choose-Attribute}(\textit{attrs, examples})</td>
<td>(choose-attribute \textit{examples}</td>
</tr>
<tr>
<td></td>
<td>\textit{candidates}, \textit{attributes})</td>
</tr>
<tr>
<td>values of \textit{best}</td>
<td>(cdr (assoc \textit{best} \textit{attributes}))</td>
</tr>
<tr>
<td>elements of \textit{examples} with \textit{best} = \textit{v}</td>
<td>(filter-examples-by-attribute-value \textit{examples} \textit{best} \textit{v})</td>
</tr>
<tr>
<td>\textit{attrs} - \textit{best}</td>
<td>(filter-list \textit{best} \textit{candidates})</td>
</tr>
</tbody>
</table>

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

\[
\begin{align*}
(\text{cons} \ \text{attribute} \\
 & (\text{list} \ (\text{cons} \ \text{val}_1 \ \text{dt}_1) \\
 & \ldots \\
 & (\text{cons} \ \text{val}_n \ \text{dt}_n)))
\end{align*}
\]

where \text{val}_1 through \text{val}_n are the domain values for \textit{attribute} and \text{dt}_1 through \text{dt}_n are the corresponding (recursively built) decision trees for \textit{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 \textit{let}).

Problem 3: decision-tree-classify (20 points)

Implement the decision-tree-classify procedure documented in \textit{assignment.scm}. Note the structure of the procedure outlined in the examples above.

Problem 4: Analysis (20 points)

You now have everything you need to train and test your supervised learning classifier. Note that \textit{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 \textit{not} need to be the same ones used to train the decision tree.
Part A

Print 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 (“Assessing the performance of the learning algorithm,” AIMA pp. 660-661) using the mushroom data.

The procedure (rand-split-list lst) in analysis.scm randomly divides its argument into two lists, returning a list of two lists. For example

```scheme
> (define split (rand-split-list (iota 5)))
> split
(((3 6 0 5) (4 1 2 7))
```

You can use this to split the list of examples into a training set and a testing set. In addition, you may extract a random sample from a list (i.e., a list of training examples) using (rand-sub-list lst len). For example

```scheme
> (rand-sub-list (car split) 2)
(5 3)
```

Your task is to measure the accuracy of the decision tree using a single test set, and randomly selected training subsets of lengths 5, 10, 15, 20, 25, ... 85, 90, 95, 100. Report the average accuracy over 20 runs for each length and present your learning curve in a table or (better!) a graph.

**Hint:** You can easily create nested “loop”s in Scheme using `iota` and `map`. For instance, here is a “loop” that does 3 things 5 times.

```scheme
> (map (lambda (iter-1)
      (map (lambda (iter-2)
             (list iter-1 iter-2))
             (iota 5)))
      (iota 3))
```

**Hint:** To find the average accuracy over several runs, use `apply` to apply an operation to the list produced by the inner “loop.”