CSC 261 **Lab 9: Decision Tree Learning** **Spring 2020**

Assigned: Monday 6 April

Due: Monday 13 April, 10:30 pm

Objective: Understand decision tree learning

Collaboration: This homework assignment will be completed in pairs assigned by the instructor.

Introduction

Decision trees quickly learn supervised classification tasks. The time is at worst quadratic in the number of training examples, but learning time is closer to $O(n \log n)$ when the splits are well-balanced. In this assignment, you will complete a Scheme-based decision tree learner using AIMA's DECISION-TREE-LEARNING (Fig 18.5) pseudo-code algorithm. Based on some simple tests of physical characteristics, our application will determine whether a mushroom is edible or poisonous according to the Audubon Society Field Guide. We have over 8,000 training examples using 21 features.[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.rkt` 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 values each may take. The latter will be useful for the tree construction process.

`dtree.rkt` Several base methods you will use to implement the decision tree learner. While you are welcome to study the implementations, the documentation and examples below should tell you all you need.

`mushroom.rkt` 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`.

`learning.rkt` Documentation for the procedures you will write.

Examples

Training Examples

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

[1] Bache, K. & Lichman, M. (2013). *UCI machine learning repository: Mushroom data set.* Irvine, CA: University of California, School of Information and Computer Science. http://archive.ics.uci.edu/ml/datasets/Mushroom

```
> (require "restauraunt.rkt")
> (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 do we know what the *other* possible values of the attributes might have been? For that we keep track of all the attributes in separate association list, as in the `restaurant-attributes` variable from `restaurant.scm`. In the attributes alist, the key is the attribute name, and the associated value is a list of all the values that attribute may take on. 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 an attribute's possible values using `cdr` and `assoc`

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

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 `X1` would be constructed as

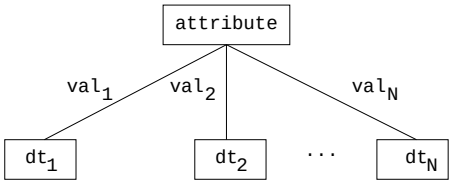
```
(cons #t X1)
```

to indicate that the label is `#t` (i.e., will wait).

Decision Tree Representation

How can we represent a decision tree in Scheme? It is 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. For lists, the base case 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 recursive case specifies the attribute and a decision sub-tree for each value of that attribute. In Scheme, this would look like the following

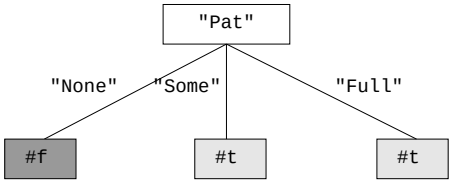
```
(cons attribute
      (list  (cons val1 dt1)
             ...
             (cons valN dtN)))
```



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 decision tree as follows.

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

What if we need to apply more than one test? Then we would have a recursive case. Instead of a simple classification (e.g., `#t` or `#f`), a decision sub-tree $\texttt{dt}_\texttt{i}$ would use the same type of structure, indicating an attribute to test and yet another decision tree to apply for each value of the attribute. For example, after discovering that the restaurant is full, we could then ask whether we are hungry, rather than immediately deciding to wait. Instead of using the simple decision `#t` in the pair `("Full" . #t)`, we would need yet another decision *tree*. Electing not to wait when hungry, our decision sub-tree (used only when the restaurant is full) would be built as follows.

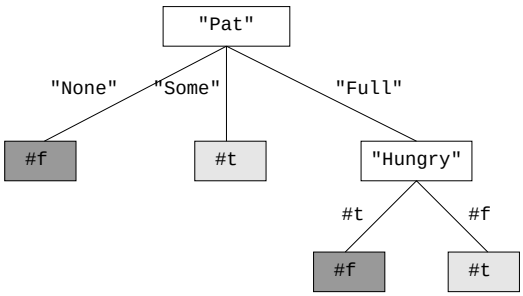
```
(define subtree-when-full
  (cons "Hun"
        (list (cons #t #f)
              (cons #f #t))))
```

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 tree as follows.

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



We can repeat the recursive nesting until we run out of attributes to test along the path. Rather than belabor the construction further, let's look 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 (giving the range 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 range (possible values). We 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 provided predicate `(all-same-label? examples)` will tell you whether all the examples have the same classification.

In the third case, a simple test reveals whether that candidate list is empty, assuming we have kept (somewhere) a list of *remaining* (i.e., untested) candidate attribute keys. In that case, the procedure `(plurality-value examples)` will return the class label that occurs most frequently in `examples`.

If none of these three cases occur, we must recursively build a decision tree. We begin by selecting an attribute to test 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 range of the `candidates`. (We'll have more to say about this in the next section.)

Once we know `best-attrib`, the value returned by `choose-attribute`, we use this attribute as the first element in the list forming our decision tree. To build the rest of the list, we have to loop over the all the values in the range 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 add branches forming 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 (recursively) build the decision sub-tree *only* on those examples. Fortunately, the procedure `(filter-examples-by-attribute-value examples attribute value)` does just that. 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 ranges, 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 local helper (i.e., a simple named `let`) to iteratively/recursively bind the smaller sets of examples and candidate attributes. In this way, you can avoid passing along the full attribute/range association list with every recursive call.

Choosing an Attribute

So far we have side-stepped how to choose an attribute. The textbook describes the **information gain** as the difference between an existing information content (due to Claude Shannon[2]) and the information resulting from applying some test. Our learner will use this metric, implemented for you as `(information-gain examples candidate attributes)` where `candidate` is some attribute and `attributes` is the association list giving the ranges 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 over half a bit, which probably explains why this is the leading attribute test.

[2]Shannon, C.E. (July 1948), A Mathematical Theory of Communication, *Bell System Technical Journal, 27*, 379–423. http://cm.bell-labs.com/cm/ms/what/shannonday/shannon1948.pdf

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"
       (#t
         "Type"
           ("French" . #f)
           ("Thai"
              "Rain"
                (#t . #t)
                (#f . #f))
           ("Burger" . #t)
           ("Italian" . #f))
       (#f . #f)))
```

Note that this tree is substantially the same as the one given in the text (Figure 18.6). The only differences are in the subtree for French and Thai restaurants. 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. There are no instances of being hungry at a full French restauraunt. The four examples of being hungry at a full restauraunt are evenly split, and our learning algorithm chose one label, while the textbook uses the other.

Assignment

Problem 1: `choose-attribute`

Using the procedure `information-gain` described above (and documented in `dtree.rkt`), write the `choose-attribute` procedure documented in `learning.rkt`; the algorithm's pseudo-code is 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 `learning.rkt`. 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 candidates best)`

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

Problem 3: A Mushroom Tree

Display the Scheme value for and draw a graphical representation of the tree learned by your algorithm on *all* of the mushroom data. How does the size of the tree compare with what you expected?

What to turn in

Your submission should include the following

- Your completed `learning.rkt`
- A short `driver.rkt` program
  - demonstrating the correctness of your procedures, and
  - containing the code for Problem 3 (learning/displaying the mushroom tree Scheme value)
- A transcript `output.txt` of your driver program's output
- A single PDF `results.pdf` containing the rendered tree and (brief) commentary from Problem 3