19 Machine Learning in Lisp

Chapter Objectives

ID3 algorithm and inducing decision trees from lists of examples.
A basic Lisp implementation of ID3
Demonstration on a simple credit assessment example.

Chapter Contents

19.1 Learning: The ID3 Algorithm
19.2 Implementing ID3

19.1 Learning: The ID3 Algorithm

In this section, we implement the ID3 induction algorithm described in Luger (2009, Section 10.3). ID3 infers decision trees from a set of training examples, which enables classification of an object on the basis of its properties. Each internal node of the decision tree tests one of the properties of a candidate object, and uses the resulting value to select a branch of the tree. It continues through the nodes of the tree, testing various properties, until it reaches a leaf, where each leaf node denotes a classification. ID3 uses an information theoretic test selection function to order tests so as to construct a (nearly) optimal decision tree. See Table 19.1 for a sample data set and Figure 19.1 for an ID3 induced decision tree. The details for the tree induction algorithms may be found in Luger (2009, Section 10.3) and in Quinlan (1986).

The ID3 algorithm requires that we manage a number of complex data structures, including objects, properties, sets, and decision trees. The heart of our implementation is a set of structure definitions, aggregate data types similar to records in the Pascal language or structures in C. Using `defstruct`, Common Lisp allows us to define types as collections of named slots; `defstruct` constructs functions needed to create and manipulate objects of that type.

Along with the use of structures to define data types, we exploit higher order functions such as `mapcar`. As the stream-based approach to our expert system shell demonstrated, the use of maps and filters to apply functions to lists of objects can often capture the intuition behind an algorithm with greater clarity than less expressive programming styles. The ability to treat functions as data, to bind function closures to symbols and process them using other functions, is a cornerstone of Lisp programming style.

A Credit History Example

This chapter will demonstrate the ID3 implementation using a simple credit assessment example. Suppose we want to determine a person's credit risk (high, moderate, low) based on data recorded from past loans. We can represent this as a decision tree, where each node examines one aspect of a person's credit profile. For example, if one of the factors we care about is

A Credit History Example

This chapter will demonstrate the ID3 implementation using a simple credit assessment example. Suppose we want to determine a person's credit risk (high, moderate, low) based on data recorded from past loans. We can represent this as a decision tree, where each node examines one aspect of a person's credit profile. For example, if one of the factors we care about is collateral, then the collateral node will have two branches: no collateral and adequate collateral.

The challenge a machine learning algorithm faces is to construct the "best" decision tree given a set of training examples. Exhaustive training sets are rare in machine learning, either because the data is not available, or because such sets would be too large to manage effectively. ID3 builds decision trees under the assumption that the simplest tree that correctly classifies all training instances is most likely to be correct on new instances, since it makes the fewest assumptions from the training data. ID3 infers a simple tree from training data using a greedy algorithm: select the test property that gives the most information about the training set, partition the problem on this property and recur. The implementation that we present illustrates this algorithm.

14.	high	bad	high	none	$15k to $35k

Table 19.1 Training data for the credit example

Figure 19.1 shows a decision tree that correctly classifies this data.

`defstruct` allows us to create structure data items in Lisp. For example, using `defstruct`, we can define a new *data type*, `employee`, by evaluating a form. `employee` is the name of the defined type; `name`,

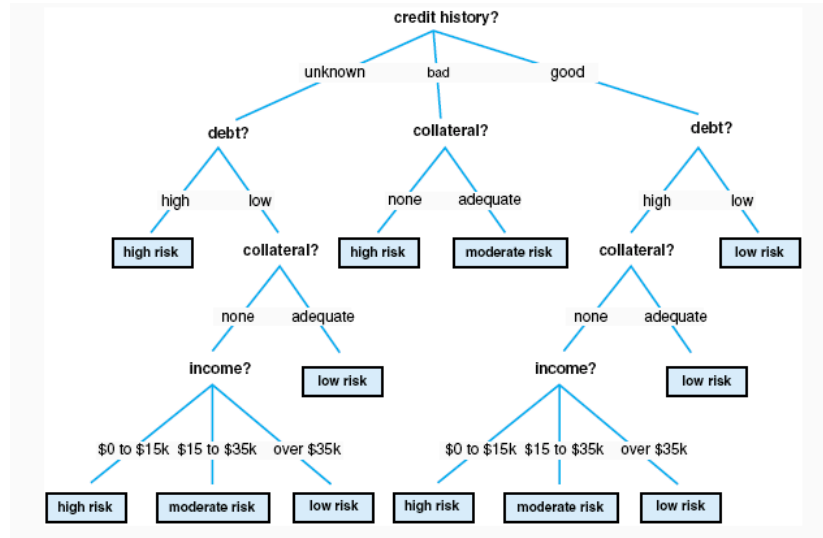

Figure 19.1 A decision tree that covers the data of Table 19.1

Defining Structures Using Defstruct

```
(defstruct employee
    name
    address
    serial-number
    department
    salary)
```

Here, `defstruct` takes as its arguments a symbol, which will become the name of a new type, and a number of slot specifiers. Here, we have defined five slots by name; slot specifiers also allow us to define different properties of slots, including type and initialization information, see Steele (1990).

Evaluating the `defstruct` form has a number of effects, for example:

```
(defstruct <type name>
    <slot name 1>
    <slot name 2>
        …
    <slot name n>)
```

`defstruct` defines a function, named according to the scheme: make-`<type name>`, that lets us create instances of this type. For example, after defining the structure, `employee`, we may bind `new-employee` to an object of this type by evaluating:

```
(setq new-employee (make-employee))
```

We can also use slot names as keyword arguments to the `make` function, giving the instance initial values. For example:

```
(setq new-employee
    (make-employee
        :name '(Doe Jane)
        :address "1234 Main, Randolph, Vt"
        :serial-number 98765
        :department 'Sales
        :salary 4500.00))
```

`defstruct` makes `<type name>` the name of a data type. We may use this name with `typep` to test if an object is of that type, for example:

```
> (typep new-employee 'employee)
t
```

Furthermore, `defstruct` defines a function, `<type-name>-p`, which we may also use to test if an object is of the defined type. For instance:

```
> (employee-p new-employee)
t
> (employee-p '(Doe Jane))
nil
```

Finally, `defstruct` defines an accessor for each slot of the structure. These accessors are named according to the scheme:

```
<type name>-<slot name>
```

In our example, we may access the values of various slots of `new-employee` using these accessors:

```
> (employee-name new-employee)
(Doe Jane)
> (employee-address new-employee)
"1234 Main, Randolph, Vt"
> (employee-department new-employee)
Sales
```

We may also use these accessors in conjunction with `setf` to change the slot values of an instance. For example:

```
> (employee-salary new-employee)
4500.0
> (setf (employee-salary new-employee) 5000.00)
5000.0
> (employee-salary new-employee)
5000.0
```

So we see that using structures, we can define predicates and accessors of a data type in a single Lisp form. These definitions are central to our implementation of the ID3 algorithm.

When given a set of examples of known classifications, we use the sample information offered in Table 19.1, ID3 induces a tree that will correctly classify all the training instances, and has a high probability of correctly

classifying new people applying for credit, see Figure 19.1. In the discussion of ID3 in Luger (2009, Section 10.3), training instances are offered in a tabular form, explicitly listing the properties and their values for each instance. Thus, Table 19.1 lists a set of instances for learning to predict an individual's credit risk. Throughout this section, we will continue to refer to this data set.

Tables are only one way of representing examples; it is more general to think of them as objects that may be tested for various properties. Our implementation makes few assumptions about the representation of objects. For each property, it requires a function of one argument that may be applied to an object to return a value of that property. For example, if `credit-profile-1` is bound to the first example in Table 19.1, and `history` is a function that returns the value of an object's credit history, then:

```
> (history credit-profile-1)
bad
```

Similarly, we require functions for the other properties of a credit profile:

```
> (debt credit-profile-1)
high
> (collateral credit-profile-1)
none
> (income credit-profile-1)
0-to-15k
> (risk credit-profile-1)
high
```

Next we select a representation for the credit assignment example, making objects as association lists in which the keys are property names and their data are property values. Thus, the first example of Table 19.1 is represented by the association list:

```
((risk . high) (history . bad) (debt . high)
(collateral . none)(income . 0-15k))
```

We now use `defstruct` to define instances as structures. We represent the full set of training instances as a list of association lists and bind this list to examples:

```
(setq examples
    '(((risk . high) (history . bad) (debt . high)
          (collateral . none) (income . 0-15k))
     ((risk . high) (history . unknown)
         (debt . high)(collateral . none)
         (income . 15k-35k))
     ((risk . moderate) (history . unknown)
         (debt . low) (collateral . none)
         (income . 15k-35k))
    ((risk . high) (history . unknown) (debt . low)
          (collateral . none) (income . 0-15k))
```

```
((risk . low) (history . unknown) (debt . low)
     (collateral . none) (income . over-35k))
((risk . low) (history . unknown) (debt . low)
     (collateral . adequate)
     (income . over-35k))
((risk . high) (history . bad) (debt . low)
     (collateral . none) (income . 0-15k))
((risk . moderate) (history . bad) (debt . low)
     (collateral . adequate)
     (income . over-35k))
((risk . low) (history . good) (debt . low)
     (collateral . none) (income . over-35k))
((risk . low) (history . good) (debt . high)
     (collateral . adequate) (income . over-35k))
((risk . high) (history . good) (debt . high)
     (collateral . none) (income . 0-15k))
((risk . moderate) (history . good)
     (debt . high) (collateral . none)
     (income . 15k-35k))
((risk . low) (history . good) (debt . high)
     (collateral . none) (income . over-35k))
((risk . high) (history . bad) (debt . high)
     (collateral . none) (income . 15k-35k))))
```

Since the purpose of a decision tree is the determination of `risk` for a new individual, `test-instance` will include all properties except `risk`:

```
(setq test-instance
       '((history . good) (debt . low)
        (collateral . none) (income . 15k-35k)))
```

Given this representation of `object`s, we next define `property`:

```
(defun history (object)
       (cdr (assoc 'history object :test #'equal)))
(defun debt (object)
       (cdr (assoc 'debt object :test #'equal)))
(defun collateral (object)
       (cdr (assoc 'collateral object :test
            #'equal)))
(defun income (object)
       (cdr (assoc 'income object :test #'equal)))
(defun risk (object)
       (cdr (assoc 'risk object :test #'equal)))
```

A property is a function on objects; we represent these functions as a slot in a structure that includes other useful information:

```
(defstruct property
   name
   test
   values)
```

The test slot of an instance of property is bound to a function that returns a property value. name is the name of the property, and is included solely to help the user inspect definitions. values is a list of all the values that may be returned by test. Requiring that the values of each property be known in advance simplifies the implementation greatly, and is not unreasonable.

We now define decision-tree using the following structures:

```
(defstruct decision-tree
     test-name
     test
     branches)
(defstruct leaf
     value)
```

Thus decision-tree is either an instance of decision-tree or an instance of leaf. leaf has one slot, a value corresponding to a classification. Instances of type decision-tree represent internal nodes of the tree, and consist of a test, a test-name and a set of branches. test is a function of one argument that takes an object and returns the value of a property. In classifying an object, we apply test to it using funcall and use the returned value to select a branch of the tree. test-name is the name of the property. We include it to make it easier for the user to inspect decision trees; it plays no real role in the program's execution. branches is an association list of possible subtrees: the keys are the different values returned by test; the data are subtrees.

For example, the tree of Figure 19.1 would correspond to the following set of nested structures. The #S is a convention of Common Lisp I/O; it indicates that an s-expression represents a structure.

```
#S(decision-tree
    :test-name income
    :test #<Compiled-function income #x3525CE>
    :branches
         ((0-15k . #S(leaf :value high))
         (15k-35k . #S(decision-tree
            :test-name history
            :test
            #<Compiled-function history #x3514D6>
            :branches
```

```
((good . #S(leaf :value moderate))
(bad . #S(leaf :value high))
(unknown . #S(decision-tree
    :test-name debt
    :test
    #<Compiled-function debt #x351A7E>
    :branches
     ((high . #S(leaf :value high))
     (low . #S(leaf
         :value moderate)))))))
(over-35k . #S(decision-tree
    :test-name history
    :test
         #<Co…d-fun.. history #x3514D6>
    :branches
         ((good . #S(leaf :value low))
         (bad . #S(leaf :value
                           moderate))
         (unknown . #S(leaf :value
                           low)))))))
```

Although a set of training examples is, conceptually, just a collection of objects, we will make it part of a structure that includes slots for other information used by the algorithm. We define `example-frame` as:

```
(defstruct example-frame
       instances
       properties
       classifier
       size
       information)
```

`instances` is a list of objects of known classification; this is the training set used to construct a decision tree. `properties` is a list of objects of type `property`; these are the `properties` that may be used in the nodes of that tree. `classifier` is also an instance of `property`; it represents the classification that ID3 is attempting to learn. Since the examples are of known classification, we include it as another `property`. `size` is the number of examples in the `instances` slot; `information` is the information content of that set of examples. We compute `size` and `information` content from the examples. Since these values take time to compute and will be used several times, we save them in these slots.

ID3 constructs trees recursively. Given a set of examples, each an instance of `example-frame`, it selects a property and uses it to partition the set of training instances into non-intersecting subsets. Each subset contains all the instances that have the same value for that property. The property

selected becomes the test at the current node of the tree. For each subset in the partition, ID3 recursively constructs a subtree using the remaining properties. The algorithm halts when a set of examples all belong to the same class, at which point it creates a leaf.

Our final structure definition is `partition`, a division of an example set into subproblems using a particular property. We define the type `partition`:

```
(defstruct partition
       test-name
       test
       components
       info-gain)
```

In an instance of `partition`, the `test` slot is bound to the property used to create the `partition`. `test-name` is the name of the `test`, included for readability. `components` will be bound to the subproblems of the `partition`. In our implementation, `components` is an association list: the keys are the different values of the selected `test`; each datum is an instance of `example-frame`. `info-gain` is the information gain that results from using `test` as the node of the tree. As with `size` and `information` in the `example-frame` structure, this slot caches a value that is costly to compute and is used several times in the algorithm. By organizing our program around these data types, we make our implementation more clearly reflect the structure of the algorithm.

19.2 Implementing ID3

The heart of our implementation is the function `build-tree`, which takes an instance of `example-frame`, and recursively constructs a decision tree.

```
(defun build-tree (training-frame)
    (cond
                          ;Case 1: empty example set.
    ((null (example-frame-instances training-frame))
         (make-leaf :value
              “unable to classify: no examples”))
                   ;Case 2: all tests have been used.
    ((null (example-frame-properties
                       training-frame))
         (make-leaf :value (list-classes
                       training-frame)))
                 ;Case 3: all examples in same class.
    ((zerop (example-frame-information
                       training-frame))
         (make-leaf :value (funcall (property-test
```

```
                  (example-frame-classifier
                          training-frame))
                      (car (example-frame-instances
                               training-frame)))))
                         ;Case 4: select test and recur.
         (t (let ((part (choose-partition
              (gen-partitions training-frame))))
                  (make-decision-tree
                    :test-name
                          (partition-test-name part)
                    :test (partition-test part)
                    :branches (mapcar #'(lambda (x)
                         (cons (car x)
                          (build-tree (cdr x))))
                          (partition-components
                                        part)))))))
```

Using **cond**, **build-tree** analyzes four possible cases. In case 1, the example frame does not contain any training instances. This might occur if ID3 is given an incomplete set of training examples, with no instances for a given value of a property. In this case it creates a leaf consisting of the message: "**unable to classify: no examples**".

The second case occurs if the **properties** slot of **training-frame** is empty. In recursively building the decision tree, once the algorithm selects a property, it deletes it from the **properties** slot in the example frames for all subproblems. If the example **set** is inconsistent, the algorithm may exhaust all properties before arriving at an unambiguous classification of training instances. In this case, it creates a leaf whose value is a list of all classes remaining in the set of training instances.

The third case represents a successful termination of a branch of the tree. If **training-frame** has an information content of zero, then all of the examples belong to the same class; this follows from Shannon's definition of information, see Luger (2009, Section 13.3). The algorithm halts, returning a leaf node in which the value is equal to this remaining class.

The first three cases terminate tree construction; the fourth case recursively calls **build-tree** to construct the subtrees of the current node. **gen-partitions** produces a list of all possible partitions of the example set, using each test in the properties slot of **training-frame**. **choose-partition** selects the test that gives the greatest information gain. After binding the resulting partition to the variable part in a **let** block, **build-tree** constructs a node of a decision tree in which the test is that used in the chosen partition, and the **branches** slot is bound to an association list of subtrees. Each key in **branches** is a value of the test and each datum is a decision tree constructed by a recursive call to **build-tree**. Since the **components** slot of **part** is already an association list in which the keys are property values and the data are instances of **example-frame**, we implement the construction of subtrees using **mapcar** to apply **build-tree** to each datum in this association list.

gen-partitions takes one argument, training-frame, an object of type example-frame-properties, and generates all partitions of its instances. Each partition is created using a different property from the properties slot. gen-partitions employs a function, partition, that takes an instance of an example frame and an instance of a property; it partitions the examples using that property. Note the use of mapcar to generate a partition for each element of the example-frame-properties slot of training-frame.

```
(defun gen-partitions (training-frame)
       (mapcar #'(lambda (x)
              (partition training-frame x))
         (example-frame-properties training-frame)))
```

choose-partition searches a list of candidate partitions and chooses the one with the highest information gain:

```
(defun choose-partition (candidates)
    (cond ((null candidates) nil)
         ((= (list-length candidates) 1)
                 (car candidates))
          (t (let ((best (choose-partition
                      (cdr candidates))))
     (if (> (partition-info-gain (car candidates))
          (partition-info-gain best))
              (car candidates) best)))))
```

partition is the most complex function in the implementation. It takes as arguments an example frame and a property, and returns an instance of a partition structure:

```
(defun partition (root-frame property)
      (let ((parts (mapcar #'(lambda (x)
              (cons x (make-example-frame)))
                  (property-values property))))
          (dolist (instance
              (example-frame-instances root-frame))
            (push instance (example-frame-instances
                (cdr (assoc (funcall
                      (property-test property)
                                    instance)
                           parts)))))
          (mapcar #'(lambda (x)
              (let ((frame (cdr x)))
              (setf (example-frame-properties frame)
                    (remove property
                       (example-frame-properties
                                 root-frame)))
```

```
                  (setf (example-frame-classifier frame)
                            (example-frame-classifier
                                      root-frame))
                  (setf (example-frame-size frame)
                   (list-length
                      (example-frame-instances frame)))
                  (setf
                      (example-frame-information frame)
                      (compute-information
                      (example-frame-instances frame)
                      (example-frame-classifier
                                      root-frame)))))
              parts)
      (make-partition
             :test-name (property-name property)
             :test (property-test property)
             :components parts
             :info-gain
      (compute-info-gain root-frame parts))))
```

partition begins by defining a local variable, parts, using a let block. It initializes parts to an association list whose keys are the possible values of the test in property, and whose data will be the subproblems of the partition. partition implements this using the dolist macro. dolist binds local variables to each element of a list and evaluates its body for each binding At this point, they are empty instances of example-frame: the instance slots of each subproblem are bound to nil. Using a dolist form, partition pushes each element of the instances slot of root-frame onto the instances slot of the appropriate subproblem in parts. push is a Lisp macro that modifies a list by adding a new first element; unlike cons, push permanently adds a new element to the list.

This section of the code accomplishes the actual partitioning of root-frame. After the dolist terminates, parts is bound to an association list in which each key is a value of property and each datum is an example frame whose instances share that value. Using mapcar, the algorithm then completes the information required of each example frame in parts, assigning appropriate values to the properties, classifier, size and information slots. It then constructs an instance of partition, binding the components slot to parts.

list-classes is used in case 2 of build-tree to create a leaf node for an ambiguous classification. It employs a do loop to enumerate the classes in a list of examples. The do loop initializes classes to all the values of the classifier in training-frame. For each element of classes, it adds it to classes-present if it can find an element of the instances slot of training-frame that belongs to that class.

```
(defun list-classes (training-frame)
      (do
        ((classes (property-values
             (example-frame-classifier
                  training-frame)) (cdr classes))
         (classifier (property-test
             (example-frame-classifier
                  training-frame))) classes-present)
          ((null classes) classes-present)
          (if (member (car classes)
             (example-frame-instances
                  training-frame)
                  :test #'(lambda (x y)
                       (equal x (funcall
                            classifier y))))
          (push (car classes) classes-present))))
```

The remaining functions compute the information content of `examples`. `compute-information` determines the information content of a list of `examples`. It counts the number of instances in each class, and computes the proportion of the total training set belonging to each class. Assuming this proportion equals the probability that an object belongs to a class, it computes the information content of examples using Shannon's definition:

```
(defun compute-information (examples classifier)
    (let ((class-count
     (mapcar #'(lambda (x) (cons x 0))
           (property-values classifier))) (size 0))
    ;count number of instances in each class
     (dolist (instance examples)
           (incf size) (incf (cdr (assoc
             (funcall (property-test classifier)
                       instance) class-count))))
    ;compute information content of examples
    (sum #'(lambda (x) (if (= (cdr x) 0) 0
             (* –1
             (/ (cdr x) size)
             (log (/ (cdr x) size) 2))))
                       class-count)))
```

`compute-info-gain` gets the information gain of a partition by subtracting the weighted average of the information in its components from that of its parent examples.

```
(defun compute-info-gain (root parts)
       (– (example-frame-information root)
           (sum #'(lambda (x)
```

```
                (* (example-frame-information (cdr x))
                   (/ (example-frame-size (cdr x))
                      (example-frame-size root))))
           parts)))
```

`sum` computes the values returned by applying `f` to all elements of `list-of-numbers`:

```
(defun sum (f list-of-numbers)
       (apply '+ (mapcar f list-of-numbers)))
```

This completes the implementation of `build-tree`. The remaining component of the algorithm is a function, `classify`, that takes as arguments a decision tree as constructed by `build-tree`, and an object to be classified; it determines the classification of the object by recursively walking the tree. The definition of `classify` is straightforward: `classify` halts when it encounters a leaf, otherwise it applies the test from the current node to `instance`, and uses the result as the key to select a branch in a call to `assoc`.

```
(defun classify (instance tree)
    (if (leaf-p tree)
         (leaf-value tree)
         (classify instance
              (cdr (assoc
                   (funcall (decision-tree-test tree)
                            instance)
                   (decision-tree-branches tree))))))
```

Using the object definitions just defined, we now call `build-tree` on the credit example of Table 19.1. We bind tests to a list of property definitions for `history`, `debt`, `collateral` and `income`. `classifier` tests the risk of an instance. Using these definitions we bind the credit examples to an instance of `example-frame`.

```
(setq tests
       (list (make-property
            :name 'history
            :test #'history
            :values '(good bad unknown))
       (make-property
            :name 'debt
            :test #'debt
            :values '(high low))
       (make-property
            :name 'collateral
            :test #'collateral
            :values '(none adequate))
```

```
        (make-property
             :name 'income
             :test #'income
             :values
              '(0-to-15k 15k-to-35k over-35k))))
(setq classifier
        (make-property
              :name 'risk
              :test #'risk
              :values '(high moderate low)))
(setq credit-examples
        (make-example-frame
              :instances examples
              :properties tests
              :classifier classifier
              :size (list-length examples)
              :information (compute-information
examples classifier)))
```

Using these definitions, we may now induce decision trees, and use them to classify instances according to their credit risk:

```
> (setq credit-tree (build-tree credit-examples))
#S(decision-tree
    :test-name income
    :test #<Compiled-function income #x3525CE>
    :branches
         ((0-to-15k . #S(leaf :value high))
         (15k-to-35k . #S(decision-tree
             :test-name history
             :test
             #<Compiled-function history #x3514D6>
             :branches
              ((good . #S(leaf :value moderate))
              (bad . #S(leaf :value high))
              (unknown . #S(decision-tree
             :test-name debt
             :test
                 #<Compiled-function debt #x351A7E>
             :branches
                 ((high . #S(leaf :value high))
                  (low .
                 #S(leaf :value moderate)))))))
```

```
              (over-35k . #S(decision-tree
                 :test-name history
                 :test #<Compiled-function history #x…6>
                 :branches
                     ((good . #S(leaf :value low))
                     (bad . #S(leaf :value moderate))
                      (unknown .
                         #S(leaf :value low)))))))
>(classify '((history . good) (debt . low)
(collateral . none) (income . 15k-to-35k)) credit-
tree)
moderate
```

Exercises

1. Run the ID3 algorithm in another problem domain and set of examples of your choice. This will require a set of examples similar to those of Table 19.1.

2. Take the credit example in the text and randomly select two-thirds of the situations. Use these cases to create the decision tree of this chapter. Test the resulting tree using the other one-third of the test cases. Do this again, randomly selecting another two-thirds. Test again on the other one-third. Can you conclude anything from your results?

3. Consider the issues of "bagging" and "boosting" presented in Luger (2009, Section 10.3.4). Apply these techniques to the example of this chapter.

4. There are a number of other decision-tree-type learning algorithms. Get on the www and examine algorithms such QA4. Test your results from the algorithms of this chapter against the results of QA4.

5. There are a number of test bed data collections available on the www for comparing results of decision tree induction algorithms. Check out Chapter 29 and compare results for various test domains.