DT ALGORITHM

FULL

Algorithm: Generate_decision_tree. Generate a decision tree from the training tuples of data partition D.

Input:

- Data partition, D, which is a set of training tuples and their associated class labels;
- *attribute_list*, the set of candidate attributes;
- *Attribute_selection_method*, a procedure to determine the splitting criterion that "best" partitions the data tuples into individual classes. This criterion consists of a *splitting_attribute* and, possibly, either a *split point* or *splitting subset*.

Output: A decision tree.

Method:

```
(1)  create a node N;
(2)  if tuples in D are all of the same class, C then
(3)      return N as a leaf node labeled with the class C;
(4)  if attribute_list is empty then
(5)      return N as a leaf node labeled with the majority class in D; // majority voting
(6)  apply Attribute_selection_method(D, attribute_list) to find the "best" splitting_criterion;
(7)  label node N with splitting_criterion;
(8)  if splitting_attribute is discrete-valued and
         multiway splits allowed then // not restricted to binary trees
(9)      attribute_list ← attribute_list − splitting_attribute; // remove splitting_attribute
(10) for each outcome j of splitting_criterion
     // partition the tuples and grow subtrees for each partition
(11)     let D_j be the set of data tuples in D satisfying outcome j; // a partition
(12)     if D_j is empty then
(13)         attach a leaf labeled with the majority class in D to node N;
(14)     else attach the node returned by Generate_decision_tree(D_j, attribute_list) to node N;
     endfor
(15) return N;
```

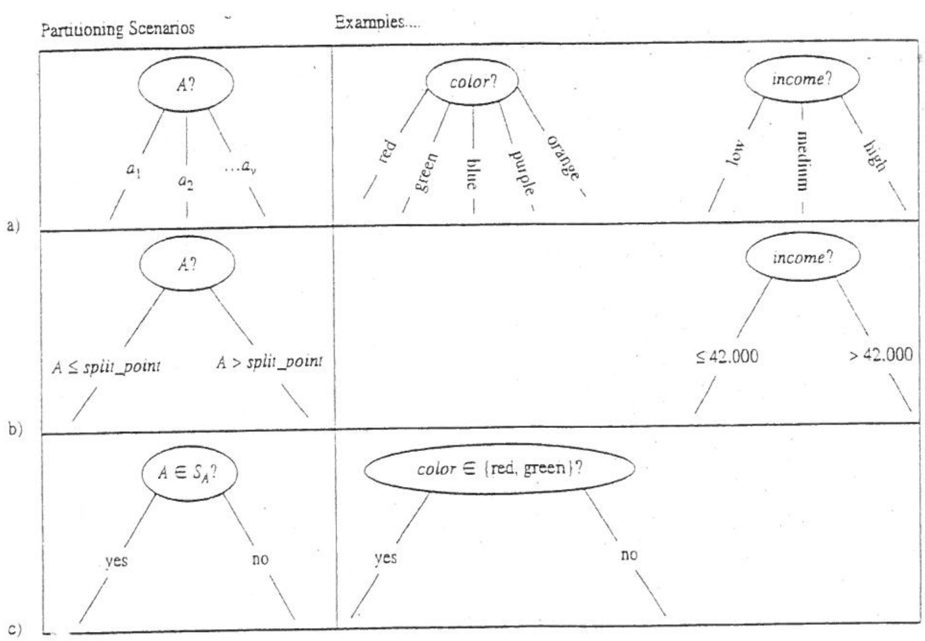

Figure 6.4 Three possibilities for partitioning tuples based on the splitting criterion, shown with examples. Let A be the splitting attribute. (a) If A is discrete-valued, then one branch is grown for each known value of A. (b) If A is continuous-valued, then two branches are grown, corresponding to $A \leq split\_point$ and $A > split\_point$. (c) If A is discrete-valued and a binary tree must be produced, then the test is of the form $A \in S_A$, where S_A is the splitting subset for A.

From Data Mining Book

DECISION TREE ALGORITHM BASIC

Algorithm: Generate_decision_tree. Generate a decision tree from the given training data.

Input: The training samples, *samples*, represented by discrete-valued attributes; the set of candidate attributes, *attribute-list*.

Output: A decision tree.

Method:

(1) create a node N;
(2) if *samples* are all of the same class, C then
(3)   return N as a leaf node labeled with the class C;
(4) if *attribute-list* is empty then
(5)   return N as a leaf node labeled with the most common class in *samples*; // majority voting
(6) select *test-attribute*, the attribute among *attribute-list* with the highest information gain;
(7) label node N with *test-attribute*;
(8) for each known value a_i of *test-attribute* // partition the samples
(9)   grow a branch from node N for the condition *test-attribute* = a_i;
(10)   let s_i be the set of samples in *samples* for which *test-attribute* = a_i; // a partition
(11)   if s_i is empty then
(12)   attach a leaf labeled with the most common class in *samples*;
(13)   else attach the node returned by Generate_decision_tree(s_i, *attribute-list*–*test-attribute*);

Figure 7.3 Basic algorithm for inducing a decision tree from training samples.

Algorithm: **Backpropagation.** Neural network learning for classification, using the backpropagation algorithm.

Input: The training samples, *samples*; the learning rate, l; a multilayer feed-forward network, *network*.

Output: A neural network trained to classify the samples.

Method:

Initialize all weights and biases in *network*;
while terminating condition is not satisfied {
  for each training sample X in *samples* {
    // Propagate the inputs forward:
    for each hidden or output layer unit j {
      $I_j = \sum_i w_{ij} O_i + \theta_j$; //compute the net input of unit j with respect to the previous layer, i
      $O_j = \frac{1}{1+e^{-I_j}}$; } // compute the output of each unit j
    // Backpropagate the errors:
    for each unit j in the output layer
      $Err_j = O_j(1 - O_j)(T_j - O_j)$; // compute the error
    for each unit j in the hidden layers, from the last to the first hidden layer
      $Err_j = O_j(1 - O_j) \sum_k Err_k w_{jk}$; // compute the error with respect to the next higher layer, k
    for each weight w_{ij} in *network* {
      $\Delta w_{ij} = (l) Err_j O_i$; // weight increment
      $w_{ij} = w_{ij} + \Delta w_{ij}$; } // weight update
    for each bias θ_j in *network* {
      $\Delta \theta_j = (l) Err_j$; // bias increment
      $\theta_j = \theta_j + \Delta \theta_j$; } // bias update
  } }

Figure 7.9 Backpropagation algorithm.

NN Algorithm
BACKPROPAGATION