Classification by Decision Tree Induction

- Decision tree induction is the learning of decision trees from class-labeled training tuples. A decision tree is a flowchart-like tree structure, where each internal node (nonleaf node) denotes a test on an attribute, each branch represents an outcome of the test, and each leaf node (or *terminal node*) holds a class label. The topmost node in a tree is the root node.

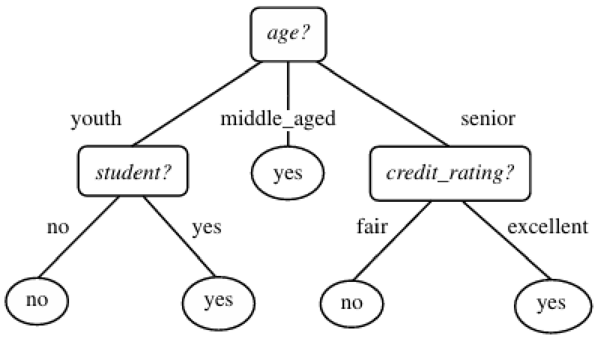


Figure 6.2 A decision tree for the concept *buys_computer*, indicating whether a customer at *AllElectronics* is likely to purchase a computer. Each internal (nonleaf) node represents a test on an attribute. Each leaf node represents a class (either *buys_computer = yes* or *buys_computer = no*).

- A typical decision tree is shown in Figure 6.2. It represents the concept *buys computer*, that is, it predicts whether a customer at *AllElectronics* is likely to purchase a computer. Internal nodes are denoted by rectangles, and leaf nodes are denoted by ovals. Some decision tree algorithms produce only *binary* trees (where each internal node branches to exactly two other nodes), whereas others can produce nonbinary trees.
- *"How are decision trees used for classification?"* Given a tuple, ***X***, for which the associ- ated class label is unknown, the attribute values of the tuple are tested against the decision tree. A path is traced from the root to a leaf node, which holds the class prediction for that tuple. Decision trees can easily be converted to classification rules.
- *"Why are decision tree classifiers so popular?"* The construction of decision tree classifiers does not require any domain knowledge or parameter setting, and therefore is appropriate for exploratory knowledge discovery. Decision trees can handle high dimensional data. Their representation of acquired knowledge in tree form is intuitive and generally easy to assimilate by humans.
- The learning and classification steps of decision tree induction are simple and fast. In general, decision tree classifiers have good accuracy. However, successful use may depend on the data at hand. Decision tree induction algorithms have been used for classification in many application areas, such as medicine, manufacturing and production, financial analysis, astronomy, and molecular biology. Decision trees are the basis of several commercial rule induction systems.

6.3.1 Decision Tree Induction

- During the late 1970s and early 1980s, J. Ross Quinlan, a researcher in machine learning, developed a decision tree algorithm known as **ID3** (Iterative Dichotomiser). This work expanded on earlier work on *concept learning systems*, described by E. B. Hunt, J. Marin, and P. T. Stone.
- Quinlan later presented **C4.5** (a successor of ID3), which became a benchmark to which newer supervised learning algorithms are often compared. In 1984, a group of statisticians (L. Breiman, J. Friedman, R. Olshen, and C. Stone) published the book *Classification and Regression Trees* (**CART**), which described the generation of binary decision trees. ID3 and CART were invented independently of one another at around the same time, yet follow a similar approach for learning decision trees from training tuples. These two cornerstone algorithms spawned a flurry of work on decision tree induction.
- ID3, C4.5, and CART adopt a greedy (i.e., nonbacktracking) approach in which deci- sion trees are constructed in a top-down recursive divide-and-conquer manner. Most algorithms for decision tree induction also follow such a top-down approach, which

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" par- titions 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:

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 6.3 Basic algorithm for inducing a decision tree from training tuples.

starts with a training set of tuples and their associated class labels. The training set is recursively partitioned into smaller subsets as the tree is being built. A basic decision tree algorithm is summarized in Figure 6.3. At first glance, the algorithm may appear long, but fear not! It is quite straightforward. The strategy is as follows.

- The algorithm is called with three parameters: *D*, *attribute list*, and *Attribute selection method*. We refer to *D* as a data partition. Initially, it is the complete set of training tuples and their associated class labels. The parameter *attribute list* is a list of attributes describing the tuples. *Attribute selection method* specifies a heuristic procedure for selecting the attribute that "best" discriminates the given tuples according to class.

- This procedure employs an attribute selection measure, such as information gain or the gini index. Whether the tree is strictly binary is generally driven by the attribute selection measure. Some attribute selection measures, such as the gini index, enforce the resulting tree to be binary. Others, like information gain, do not, therein allowing multiway splits (i.e., two or more branches to be grown from a node).

- The tree starts as a single node, *N*, representing the training tuples in *D*(step 1).[5]

  If the tuples in *D* are all of the same class, then node *N* becomes a leaf and is labeled with that class (steps 2 and 3). Note that steps 4 and 5 are terminating conditions. All of the terminating conditions are explained at the end of the algorithm.

- Otherwise, the algorithm calls Attribute selection method to determine the splitting criterion. The splitting criterion tells us which attribute to test at node N by deter- mining the “best” way to separate or partition the tuples in D into individual classes (step 6). The splitting criterion also tells us which branches to grow from node N with respect to the outcomes of the chosen test.

- More specifically, the splitting criterion indicates the **splitting attribute** and may also indicate either a **split-point** or a **splitting subset**. The splitting criterion is determined so that, ideally, the resulting partitions at each branch are as “pure” as possible. A partition is **pure** if all of the tuples in it belong to the same class. In other words, if we were to split up the tuples in *D* according to the mutually exclusive outcomes of the splitting criterion, we hope for the resulting partitions to be as pure as possible.
- The node *N* is labeled with the splitting criterion, which serves as a test at the node (step 7). A branch is grown from node *N* for each of the outcomes of the splitting criterion. The tuples in *D* are partitioned accordingly (steps 10 to 11). There are three possible scenarios, as illustrated in Figure 6.4. Let *A* be the splitting attribute. *A* has *v* distinct values, $\{a_1, a_2, \ldots, a_v\}$, based on the training data.

1. *A is discrete-valued*: In this case, the outcomes of the test at node *N* correspond directly to the known values of *A*. A branch is created for each known value, a_j, of *A* and labeled with that value (Figure 6.4(a)). Partition D_j is the subset of class-labeled tuples in *D* having value a_j of *A*. Because all of the tuples in a given partition have the same value for *A*, then *A* need not be considered in any future partitioning of the tuples. Therefore, it is removed from *attribute list* (steps 8 to 9).
2. *A is continuous-valued*: In this case, the test at node *N* has two possible outcomes, corresponding to the conditions $A \leq$ *split point* and $A >$ *split point*, respectively,

[5]The partition of class-labeled training tuples at node *N* is the set of tuples that follow a path from the root of the tree to node *N* when being processed by the tree. This set is sometimes referred to in the literature as the *family* of tuples at node *N*. We have referred to this set as the “tuples represented at node *N*,” “the tuples that reach node *N*,” or simply “the tuples at node *N*.” Rather than storing the actual tuples at a node, most implementations store pointers to these tuples.

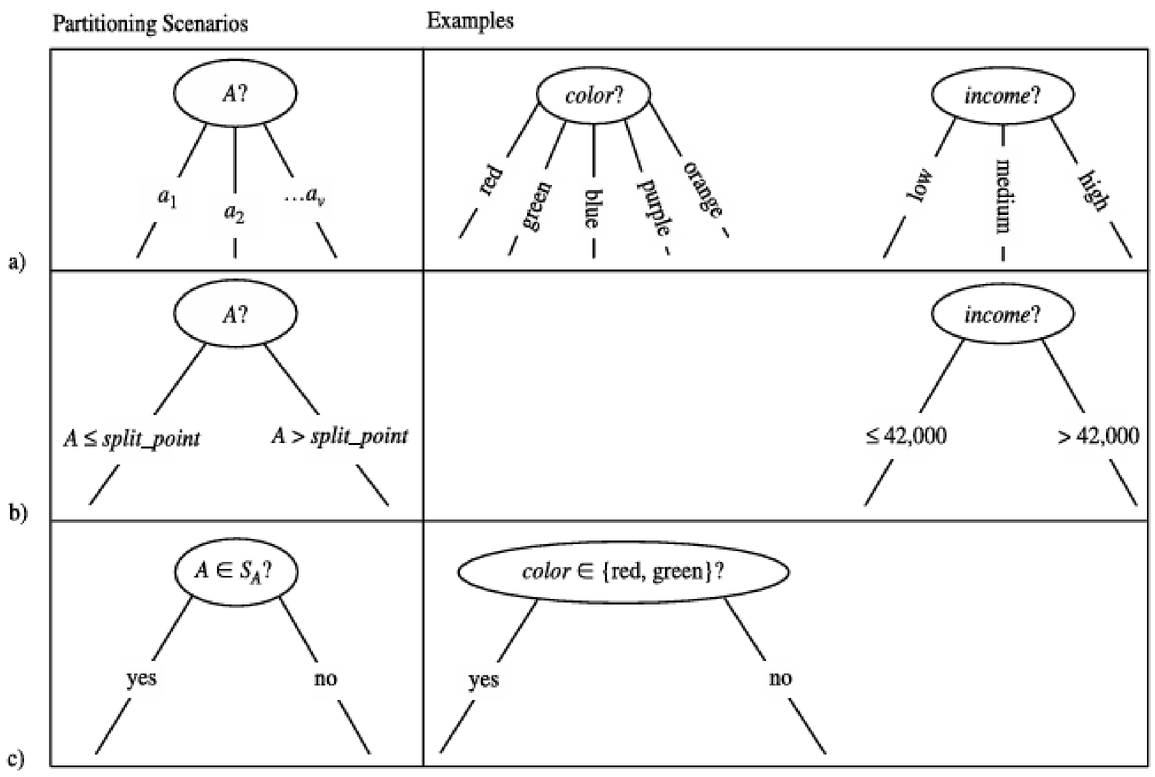


Figure 6.4 Three possibilities for partitioning tuples based on the splitting criterion, shown with examples. Let A be the splitting attribute. (a) If A is discrete-valued, then one branch is grown for each known value of A. (b) If A is continuous-valued, then two branches are grown, corresponding to $A \leq$ *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.

where *split_point* is the split-point returned by *Attribute_selection_method* as part of the splitting criterion. (In practice, the split-point, a, is often taken as the midpoint of two known adjacent values of A and therefore may not actually be a pre-existing value of A from the training data.) Two branches are grown from N and labeled according to the above outcomes (Figure 6.4(b)). The tuples are partitioned such that D_1 holds the subset of class-labeled tuples in D for which $A \leq$ *split_point*, while D_2 holds the rest.

3. *A is discrete-valued* and a *binary tree* must be produced (as dictated by the attribute selection measure or algorithm being used): The test at node N is of the form "$A \in S_A$?". S_A is the splitting subset for A, returned by *Attribute_selection_method* as part of the splitting criterion. It is a subset of the known values of A. If a given tuple has value a_j of A and if $a_j \in S_A$, then the test at node N is satisfied. Two branches are grown from N (Figure 6.4(c)). By convention, the left branch out of N is labeled *yes* so that D_1 corresponds to the subset of class-labeled tuples in D that satisfy the test. The right branch out of N is labeled *no* so that D_2 corresponds to the subset of class-labeled tuples from D that do not satisfy the test.

The algorithm uses the same process recursively to form a decision tree for the tuples at each resulting partition, D_j, of D (step 14).

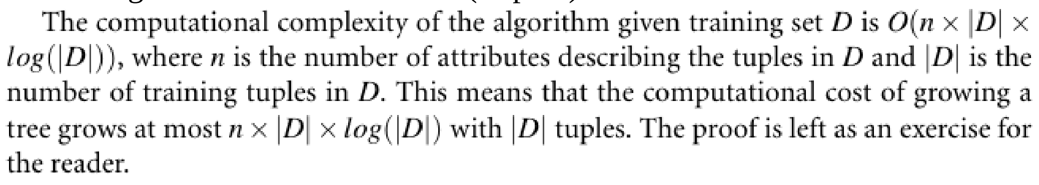
The recursive partitioning stops only when any one of the following terminating conditions is true:

1. All of the tuples in partition D (represented at node N) belong to the same class (steps 2 and 3), or
2. There are no remaining attributes on which the tuples may be further partitioned (step 4). In this case, **majority voting** is employed (step 5). This involves converting node N into a leaf and labeling it with the most common class in D. Alternatively, the class distribution of the node tuples may be

stored.

3. There are no tuples for a given branch, that is, a partition D_j is empty (step 12). In this case, a leaf is created with the majority class in D(step 13).

   The resulting decision tree is returned (step 15).



The computational complexity of the algorithm given training set D is $O(n \times |D| \times log(|D|))$, where n is the number of attributes describing the tuples in D and $|D|$ is the number of training tuples in D. This means that the computational cost of growing a tree grows at most $n \times |D| \times log(|D|)$ with $|D|$ tuples. The proof is left as an exercise for the reader.

- **Incremental** versions of decision tree induction have also been proposed. When given new training data, these restructure the decision tree acquired from learning on previous training data, rather than relearning a new tree from scratch.

6.3.1 Attribute Selection Measures

- An **attribute selection measure** is a heuristic for selecting the splitting criterion that "best" separates a given data partition, D, of class-labeled training tuples into individual classes. If we were to split D into smaller partitions according to the outcomes of the splitting criterion, ideally, each partition would be pure (i.e., all of the tuples that fall into a given partition would belong to the same class). Conceptually, the "best" splitting criterion is the one that most closely results in such a scenario.

- Attribute selection measures are also known as **splitting rules** because they determine how the tuples at a given node are to be split. The attribute selection measure provides a ranking for each attribute describing the given training tuples.

- The attribute having the best score for the measure[6] is chosen as the *splitting attribute* for the given tuples. If the splitting attribute is continuous-valued or if we are restricted to binary trees then, respectively, either a *split point* or a *splitting subset* must also be determined as part of the splitting criterion. The tree node created for partition D is labeled with the splitting criterion, branches are grown for each outcome of the criterion, and the tuples are partitioned accordingly. *This section describes three popular attribute selection measures—information gain, gain ratio, and gini index.*
- The notation used herein is as follows. Let D, the data partition, be a training set of class-labeled tuples. Suppose the class label attribute has m distinct values defining m distinct classes, C_i (for $i = 1, \dots, m$). Let $C_{i,D}$ be the set of tuples of class C_i in D. Let $|D|$ and $|C_{i,D}|$ denote the number of tuples in D and $C_{i,D}$, respectively.

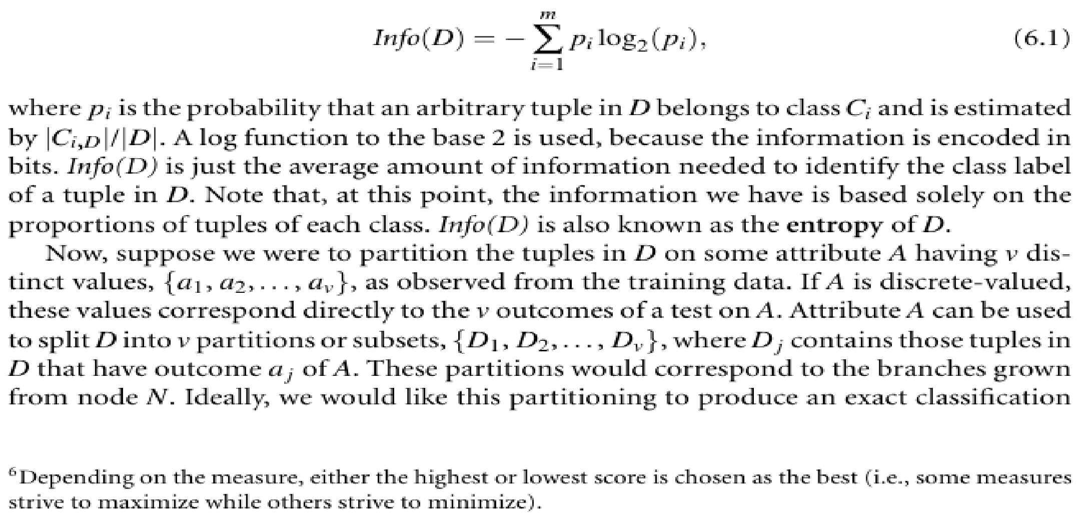
Information gain

- ID3 uses **information gain** as its attribute selection measure. This measure is based on pioneering work by Claude Shannon on information theory, which studied the value or "information content" of messages.

- Let node N represent or hold the tuples of partition D. The attribute with the highest information gain is chosen as the splitting attribute for node N. This attribute minimizes the information needed to classify the tuples in the resulting partitions and reflects the least randomness or "impurity" in these partitions.

- Such an approach minimizes the expected number of tests needed to classify a given tuple and guarantees that a simple (but not necessarily the simplest) tree is found.

The expected information needed to classify a tuple in *D*is given by



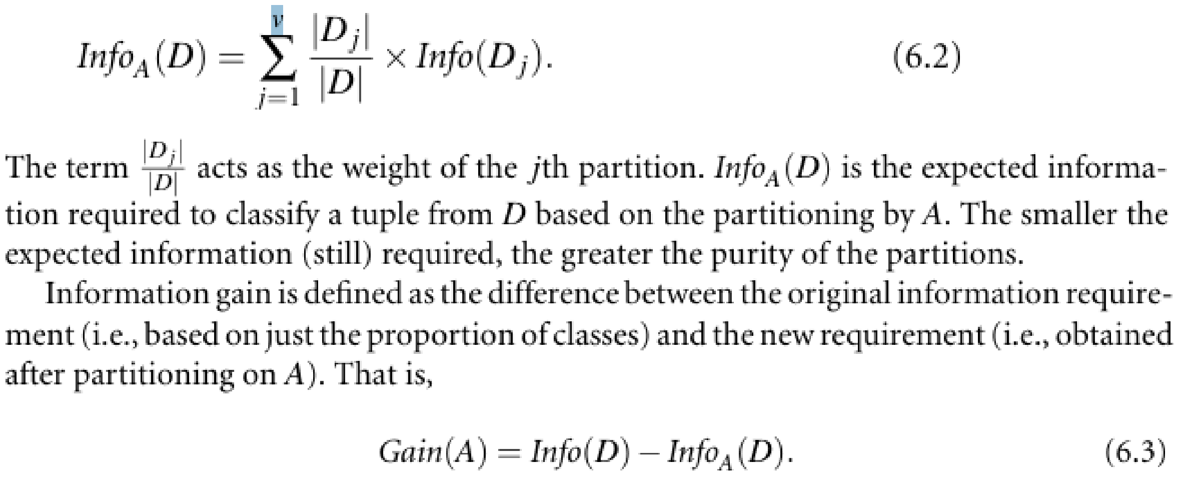
$$Info(D) = -\sum_{i=1}^{m} p_i \log_2(p_i), \tag{6.1}$$

where p_i is the probability that an arbitrary tuple in D belongs to class C_i and is estimated by $|C_{i,D}|/|D|$. A log function to the base 2 is used, because the information is encoded in bits. *Info(D)* is just the average amount of information needed to identify the class label of a tuple in D. Note that, at this point, the information we have is based solely on the proportions of tuples of each class. *Info(D)* is also known as the **entropy** of D.

Now, suppose we were to partition the tuples in D on some attribute A having v distinct values, $\{a_1, a_2, \ldots, a_v\}$, as observed from the training data. If A is discrete-valued, these values correspond directly to the v outcomes of a test on A. Attribute A can be used to split D into v partitions or subsets, $\{D_1, D_2, \ldots, D_v\}$, where D_j contains those tuples in D that have outcome a_j of A. These partitions would correspond to the branches grown from node N. Ideally, we would like this partitioning to produce an exact classification

[6]Depending on the measure, either the highest or lowest score is chosen as the best (i.e., some measures strive to maximize while others strive to minimize).

of the tuples. That is, we would like for each partition to be pure. However, it is quite likely that the partitions will be impure (e.g., where a partition may contain a collec- tion of tuples from different classes rather than from a single class). How much more information would we still need (after the partitioning) in order to arrive at an exact classification? This amount is measured by



$$Info_A(D) = \sum_{j=1}^{v} \frac{|D_j|}{|D|} \times Info(D_j). \tag{6.2}$$

The term $\frac{|D_j|}{|D|}$ acts as the weight of the jth partition. $Info_A(D)$ is the expected information required to classify a tuple from D based on the partitioning by A. The smaller the expected information (still) required, the greater the purity of the partitions.

Information gain is defined as the difference between the original information requirement (i.e., based on just the proportion of classes) and the new requirement (i.e., obtained after partitioning on A). That is,

$$Gain(A) = Info(D) - Info_A(D). \tag{6.3}$$

- In other words, *Gain(A)* tells us how much would be gained by branching on *A*. It is the expected reduction in the information requirement caused by knowing the value of *A*. The attribute *A* with the highest information gain, (*Gain*(*A*)), is chosen as the splitting attribute at node *N*. This is equivalent to saying that we want to partition on the attribute *A* that would do the "best classification," so that the amount of information still required to finish classifying the tuples is minimal (i.e., minimum $Info_A(D)$).

Example 6.1 Induction of a decision tree using information gain. Table 6.1 presents a training set, *D*, of class-labeled tuples randomly selected from the *AllElectronics* customer database. (The data are adapted from [Qui86]. In this example, each attribute is discrete-valued.

- Continuous-valued attributes have been generalized.) The class label attribute, *buys computer*, has two distinct values (namely, { *yes, no}*); therefore, there are two distinct classes (that is, *m* = 2). Let class C_1 correspond to *yes* and class C_2 correspond to *no*
- There are nine tuples of class *yes* and five tuples of class *no*. A (root) node *N* is created for the tuples in *D*. To find the splitting criterion for these tuples, we must compute the information gain of each attribute. We first use Equation (6.1) to compute the expected information needed to classify a tuple in *D*:

$$Info(D) = -\frac{9}{14}\log_2\left(\frac{9}{14}\right) - \frac{5}{14}\log_2\left(\frac{5}{14}\right) = 0.940 \text{ bits.}$$

$$Info(D) = -\frac{9}{14}\log_2\left(\frac{9}{14}\right) - \frac{5}{14}\log_2\left(\frac{5}{14}\right) = 0.940 \text{ bits.}$$

- Next, we need to compute the expected information requirement for each attribute. Let's start with the attribute *age*. We need to look at the distribution of *yes* and *no* tuples for each category of *age*. For the *age* category *youth*, there are two *yes* tuples and three *no* tuples. For the category *middle aged*, there are four *yes* tuples and zero *no* tuples. For the category *senior*, there are three *yes* tuples and two *no* tuples. Using Equation (6.2),

Table 6.1 Class-labeled training tuples from the *AllElectronics* customer database.

RID	*age*	*income*	*student*	*credit_rating*	*Class: buys_computer*
1	youth	high	no	fair	no
2	youth	high	no	excellent	no
3	middle_aged	high	no	fair	yes
4	senior	medium	no	fair	yes
5	senior	low	yes	fair	yes
6	senior	low	yes	excellent	no
7	middle_aged	low	yes	excellent	yes
8	youth	medium	no	fair	no
9	youth	low	yes	fair	yes
10	senior	medium	yes	fair	yes
11	youth	medium	yes	excellent	yes
12	middle_aged	medium	no	excellent	yes
13	middle_aged	high	yes	fair	yes
14	senior	medium	no	excellent	no

the expected information needed to classify a tuple in D if the tuples are partitioned according to *age* is

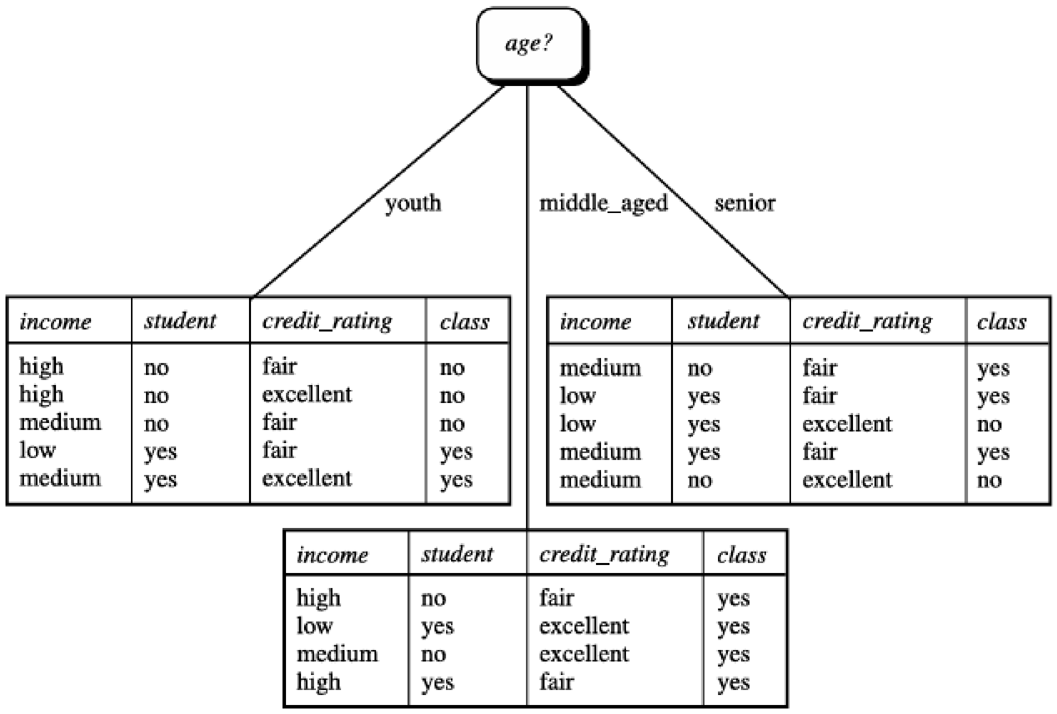
$$\begin{aligned} Info_{age}(D) = &\frac{5}{14} \times \left(-\frac{2}{5}\log_2\frac{2}{5} - \frac{3}{5}\log_2\frac{3}{5}\right) \\ &+\frac{4}{14} \times \left(-\frac{4}{4}\log_2\frac{4}{4} - \frac{0}{4}\log_2\frac{0}{4}\right) \\ &+\frac{5}{14} \times \left(-\frac{3}{5}\log_2\frac{3}{5} - \frac{2}{5}\log_2\frac{2}{5}\right) \\ = &\ 0.694 \text{ bits.} \end{aligned}$$

Hence, the gain in information from such a partitioning would be

$$Gain(age) = Info(D) - Info_{age}(D) = 0.940 - 0.694 = 0.246 \text{ bits.}$$

- Similarly, we can compute *Gain*(*income*) = 0.029 bits, *Gain*(*student*) = 0.151 bits, and *Gain*(*credit rating*) = 0.048 bits. Because *age* has the highest information gain among the attributes, it is selected as the splitting attribute. Node *N* is labeled with *age*, and branches are grown for each of the attribute's values.

- The tuples are then partitioned accordingly, as shown in Figure 6.5. Notice that the tuples falling into the partition for *age* = *middle aged* all belong to the same class. Because they all belong to class *"yes,"* a leaf should therefore be created at the end of this branch and labeled with *"yes."* The final decision tree returned by the algorithm is shown in Figure 6.2.



income	student	credit_rating	class
high	no	fair	no
high	no	excellent	no
medium	no	fair	no
low	yes	fair	yes
medium	yes	excellent	yes

income	student	credit_rating	class
high	no	fair	yes
low	yes	excellent	yes
medium	no	excellent	yes
high	yes	fair	yes

income	student	credit_rating	class
medium	no	fair	yes
low	yes	fair	yes
low	yes	excellent	no
medium	yes	fair	yes
medium	no	excellent	no

Figure 6.5 The attribute *age* has the highest information gain and therefore becomes the splitting attribute at the root node of the decision tree. Branches are grown for each outcome of *age*. The tuples are shown partitioned accordingly.

- *"But how can we compute the information gain of an attribute that is continuous-valued, unlike above?"* Suppose, instead, that we have an attribute *A* that is continuous-valued, rather than discrete-valued.
- (For example, suppose that instead of the discretized version of *age* above, we instead have the raw values for this attribute.) For such a scenario, we must determine the "best" **split-point** for *A*, where the split-point is a threshold on *A*.
- We first sort the values of *A* in increasing order. Typically, the midpoint between each pair of adjacent values is considered as a possible split-point. Therefore, given *v* values of *A*, then *v*-1 possible splits are evaluated. For example, the midpoint between the values a_i and a_{i+1} of *A* is

$$\frac{a_i + a_{i+1}}{2}. \tag{6.4}$$

If the values of *A* are sorted in advance, then determining the best split for *A* requires only one pass through the values. For each possible split-point for *A*, we evaluate $Info_A(D)$, where the number of partitions is two, that is $v = 2$ (or $j = 1, 2$) in Equation (6.2). The point with the minimum expected information requirement for *A* is selected as the *split_point* for *A*. D_1 is the set of tuples in *D* satisfying $A \leq split\_point$, and D_2 is the set of tuples in *D* satisfying $A > split\_point$.

Gain ratio

- The information gain measure is biased toward tests with many outcomes. That is, it prefers to select attributes having a large number of values. For example, consider an attribute that acts as a unique identifier, such as *product_ID*. A split on *product_ID* would result in a large number of partitions (as many as there are values), each one containing just one tuple. Because each partition is pure, the information required to classify data set D based on this partitioning would be $Info_{product\_ID}(D) = 0$. Therefore, the information gained by partitioning on this attribute is maximal. Clearly, such a partitioning is useless for classification.

- C4.5, a successor of ID3, uses an extension to information gain known as *gain ratio*, which attempts to overcome this bias. It applies a kind of normalization to information gain using a "split information" value defined analogously with *Info*(D) as

$$SplitInfo_A(D) = -\sum_{j=1}^{v} \frac{|D_j|}{|D|} \times \log_2\left(\frac{|D_j|}{|D|}\right).$$

- This value represents the potential information generated by splitting the training data set, D, into v partitions, corresponding to the v outcomes of a test on attribute A. Note that, for each outcome, it considers the number of tuples having that outcome with respect to the total number of tuples in D. It differs from information gain, which mea- sures the information with respect to classification that is acquired based on the same partitioning. The gain ratio is defined as

$$GainRatio(A) = \frac{Gain(A)}{SplitInfo(A)}. \tag{6.6}$$

- The attribute with the maximum gain ratio is selected as the splitting attribute. Note, however, that as the split information approaches 0, the ratio becomes unstable. A con- straint is added to avoid this, whereby the information gain of the test selected must be large—at least as great as the average gain over all tests examined.

Example 6.2 Computation of gain ratio for the attribute *income*. A test on *income* splits the data of Table 6.1 into three partitions, namely *low*, *medium*, and *high*, containing four, six, and four tuples, respectively. To compute the gain ratio of *income*, we first use Equation (6.5)to obtain

$$\begin{aligned} SplitInfo_A(D) &= -\frac{4}{14} \times \log_2\left(\frac{4}{14}\right) - \frac{6}{14} \times \log_2\left(\frac{6}{14}\right) - \frac{4}{14} \times \log_2\left(\frac{4}{14}\right). \\ &= 0.926. \end{aligned}$$

From Example 6.1, we have *Gain*(*income*) = 0.029. Therefore, *GainRatio*(*income*) = 0.029/0.926 = 0.03

Gini index

The Gini index is used in CART. Using the notation described above, the Gini index measures the impurity of *D*, a data partition or set of training tuples, as

$$Gini(D) = 1 - \sum_{i=1}^{m} p_i^2, \qquad (6.7)$$

where p_i is the probability that a tuple in D belongs to class C_i and is estimated by $|C_{i,D}|/|D|$. The sum is computed over m classes.

The Gini index considers a binary split for each attribute. Let's first consider the case where A is a discrete-valued attribute having v distinct values, $\{a_1, a_2, \ldots, a_v\}$, occurring in D. To determine the best binary split on A, we examine all of the possible subsets that can be formed using known values of A. Each subset, S_A, can be considered as a binary test for attribute A of the form "$A \in S_A$?". Given a tuple, this test is satisfied if the value of A for the tuple is among the values listed in S_A. If A has v possible values, then there are 2^v possible subsets. For example, if *income* has three possible values, namely {*low, medium, high*}, then the possible subsets are {*low, medium, high*}, {*low, medium*}, {*low, high*}, {*medium, high*}, {*low*}, {*medium*}, {*high*}, and {}. We exclude the power set, {*low, medium, high*}, and the empty set from consideration since, conceptually, they do not represent a split. Therefore, there are $2^v - 2$ possible ways to form two partitions of the data, D, based on a binary split on A.

When considering a binary split, we compute a weighted sum of the impurity of each resulting partition. For example, if a binary split on A partitions D into D_1 and D_2, the gini index of D given that partitioning is

$$Gini_A(D) = \frac{|D_1|}{|D|} Gini(D_1) + \frac{|D_2|}{|D|} Gini(D_2). \qquad (6.8)$$

For each attribute, each of the possible binary splits is considered. For a discrete-valued attribute, the subset that gives the minimum gini index for that attribute is selected as its splitting subset.

For continuous-valued attributes, each possible split-point must be considered. The strategy is similar to that described above for information gain, where the midpoint between each pair of (sorted) adjacent values is taken as a possible split-point. The point giving the minimum Gini index for a given (continuous-valued) attribute is taken as the split-point of that attribute. Recall that for a possible split-point of A, D_1 is the set of tuples in D satisfying $A \leq split\_point$, and D_2 is the set of tuples in D satisfying $A > split\_point$.

The reduction in impurity that would be incurred by a binary split on a discrete- or continuous-valued attribute A is

$$\Delta Gini(A) = Gini(D) - Gini_A(D). \qquad (6.9)$$

- The attribute that maximizes the reduction in impurity (or, equivalently, has the minimum Gini index) is selected as the splitting attribute. This attribute and either its splitting subset (for a discrete-valued splitting attribute) or split-point (for a continuous-valued splitting attribute) together form the splitting criterion.

- **Example 6.3 Induction of a decision tree using gini index.** Let *D* be the training data of Table 6.1 where there are nine tuples belonging to the class *buys computer = yes* and the remaining five tuples belong to the class *buys computer = no*. A (root) node *N* is created for the tuples in *D*. We first use Equation (6.7) for Gini index to compute the impurity of *D:*

$$Gini(D) = 1 - \left(\frac{9}{14}\right)^2 - \left(\frac{5}{14}\right)^2 = 0.459.$$

To find the splitting criterion for the tuples in *D*, we need to compute the gini index for each attribute. Let's start with the attribute *income* and consider each of the possible splitting subsets. Consider the subset *low, medium* . This would result in 10 tuples in partition D_1 satisfying the condition "*income low, medium* ." The remaining four tuples of *D* would be assigned to partition D_2. The Gini index value computed based on this partitioning is

$$\begin{aligned}
&Gini_{income \in \{low, medium\}}(D) \\
&= \frac{10}{14} Gini(D_1) + \frac{4}{14} Gini(D_2) \\
&= \frac{10}{14}\left(1 - \left(\frac{6}{10}\right)^2 - \left(\frac{4}{10}\right)^2\right) + \frac{4}{14}\left(1 - \left(\frac{1}{4}\right)^2 - \left(\frac{3}{4}\right)^2\right) \\
&= 0.450 \\
&= Gini_{income \in \{high\}}(D).
\end{aligned}$$

Similarly, the Gini index values for splits on the remaining subsets are: 0.315 (for the subsets {*low, high*} and {*medium*}) and 0.300 (for the subsets {*medium, high*} and {*low*}). Therefore, the best binary split for attribute *income* is on {*medium, high*} (or {*low*}) because it minimizes the gini index. Evaluating the attribute, we obtain {*youth, senior*} (or {*middle_aged*}) as the best split for *age* with a Gini index of 0.375; the attributes {*student*} and {*credit_rating*} are both binary, with Gini index values of 0.367 and 0.429, respectively.

The attribute *income* and splitting subset {*medium, high*} therefore give the minimum gini index overall, with a reduction in impurity of $0.459 - 0.300 = 0.159$. The binary split "$income \in \{medium, high\}$" results in the maximum reduction in impurity of the tuples in *D* and is returned as the splitting criterion. Node *N* is labeled with the criterion, two branches are grown from it, and the tuples are partitioned accordingly. Hence, the Gini index has selected *income* instead of *age* at the root node, unlike the (nonbinary) tree created by information gain (Example 6.1). ■

- This section on attribute selection measures was not intended to be exhaustive. We have shown three measures that are commonly used for building decision trees. These measures are not without their biases. Information gain, as we saw, is biased toward mul- tivalued attributes. Although the gain ratio adjusts for this bias, it tends to prefer unbal- anced splits in which one partition is much smaller than the others.
- The Gini index is biased toward multivalued attributes and has difficulty when the number of classes is large. It also tends to favor tests that result in equal-sized partitions and purity in both partitions. Although biased, these measures give reasonably good results in practice.
- Many other attribute selection measures have been proposed. CHAID, a decision tree algorithm that is popular in marketing, uses an attribute selection measure that is based on the statistical □^{2} test for independence. Other measures include C-SEP (which per- forms better than information gain and Gini index in certain cases) and G-statistic
- Attribute selection measures based on the **Minimum Description Length (MDL)** prin- ciple have the least bias toward multivalued attributes. MDL-based measures use encoding techniques to define the "best" decision tree as the one that requires the fewest number of bits to both (1) encode the tree and (2) encode the exceptions to the tree (i.e., cases that are not correctly classified by the tree). Its main idea is that the simplest of solutions is preferred.

- Other attribute selection measures consider **multivariate splits** (i.e., where the parti- tioning of tuples is based on a *combination* of attributes, rather than on a single attribute). The CART system, for example, can find multivariate splits based on a linear combina- tion of attributes. Multivariate splits are a form of **attribute** (or feature) **construction**, where new attributes are created based on the existing ones.

- *"Which attribute selection measure is the best?"* All measures have some bias. It has been shown that the time complexity of decision tree induction generally increases exponen- tially with tree height. Hence, measures that tend to produce shallower trees (e.g., with multiway rather than binary splits, and that favor more balanced splits) may be pre- ferred. However, some studies have found that shallow trees tend to have a large number of leaves and higher error rates. Despite several comparative studies, no one attribute selection measure has been found to be significantly superior to others. Most measures give quite good results.

6.3.2 Tree Pruning

- When a decision tree is built, many of the branches will reflect anomalies in the training data due to noise or outliers. Tree pruning methods address this problem of *overfit- ting* the data. Such methods typically use statistical measures to remove the least reli- able branches. An unpruned tree and a pruned version of it are shown in Figure 6.6.

- Pruned trees tend to be smaller and less complex and, thus, easier to comprehend. They are usually faster and better at correctly classifying independent test data (i.e., of previ- ously unseen tuples) than unpruned trees.
"How does tree pruning work?" There are two common approaches to tree pruning: *prepruning* and *postpruning*.

In the **prepruning** approach, a tree is "pruned" by halting its construction early (e.g., by deciding not to further split or partition the subset of training tuples at a given node).

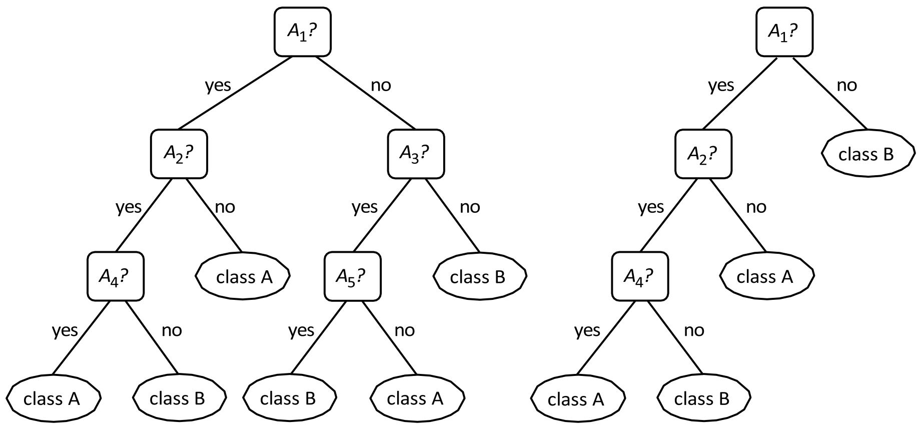


Figure 6.6 An unpruned decision tree and a pruned version of it.

Upon halting, the node becomes a leaf. The leaf may hold the most frequent class among the subset tuples or the probability distribution of those tuples.

When constructing a tree, measures such as statistical significance, information gain, Gini index, and so on can be used to assess the goodness of a split. If partitioning the tuples at a node would result in a split that falls below a prespecified threshold, then fur- ther partitioning of the given subset is halted. There are difficulties, however, in choosing an appropriate threshold. High thresholds could result in oversimplified trees, whereas low thresholds could result in very little simplification.

- The second and more common approach is **postpruning**, which removes subtrees from a "fully grown" tree. A subtree at a given node is pruned by removing its branches

and replacing it with a leaf. The leaf is labeled with the most frequent class among the subtree being replaced. For example, notice the subtree at node "A_3?" in the unpruned tree of Figure 6.6. Suppose that the most common class within this subtree is "*class B.*"

In the pruned version of the tree, the subtree in question is pruned by replacing it with the leaf "*class B.*"

- The **cost complexity** pruning algorithm used in CART is an example of the postprun- ing approach. This approach considers the cost complexity of a tree to be a function of the number of leaves in the tree and the error rate of the tree (where the **error rate** is the percentage of tuples misclassified by the tree). It starts from the bottom of the tree.
- For each internal node, *N*, it computes the cost complexity of the subtree at *N*, and the cost complexity of the subtree at *N* if it were to be pruned (i.e., replaced by a leaf node). The two values are compared. If pruning the subtree at node *N* would result in a smaller cost complexity, then the subtree is pruned.
- Otherwise, it is kept. A **pruning set** of class-labeled tuples is used to estimate cost complexity. This set is independent of the training set used to build the unpruned tree and of any test set used for accuracy estima- tion. The algorithm generates a set of progressively pruned trees. In general, the smallest decision tree that minimizes the cost complexity is preferred.
- C4.5 uses a method called **pessimistic pruning**, which is similar to the cost complex- ity method in that it also uses error rate estimates to make decisions regarding subtree pruning. Pessimistic pruning, however, does not require the use of a prune set. Instead, it uses the training set to estimate error rates. Recall that an estimate of accuracy or error based on the training set is overly optimistic and, therefore, strongly biased. The pes- simistic pruning method therefore adjusts the error rates obtained from the training set by adding a penalty, so as to counter the bias incurred.
- Rather than pruning trees based on estimated error rates, we can prune trees based on the number of bits required to encode them. The "best" pruned tree is the one that minimizes the number of encoding bits. This method adopts the Minimum Description Length (MDL) principle, which was briefly introduced in Section 6.3.2. The basic idea is that the simplest solution is preferred. Unlike cost complexity pruning, it does not require an independent set of tuples.
- Alternatively, prepruning and postpruning may be interleaved for a combined approach. Postpruning requires more computation than prepruning, yet generally leads to a more reliable tree. No single pruning method has been found to be superior over all others. Although some pruning methods do depend on the availability of additional data for pruning, this is usually not a concern when dealing with large databases.
- Although pruned trees tend to be more compact than their unpruned counterparts, they may still be rather large and complex. Decision trees can suffer from *repetition* and *replication* (Figure 6.7), making them overwhelming to interpret. **Repetition** occurs when

an attribute is repeatedly tested along a given branch of the tree (such as *“age < 60?”*, followed by *“age < 45”?*, and so on).

In **replication**, duplicate subtrees exist within themtree. These situations can impede the accuracy and comprehensibility of a decision tree. The use of multivariate splits (splits based on a combination of attributes) can prevent these problems.

Another approach is to use a different form of knowledge representation, such as rules, instead of decision trees. This is described in Section 6.5.2, which shows how a *rule-based classifier* can be constructed by extracting IF-THEN rules from a decision tree.

6.3.3 Scalability and Decision Tree Induction

“What if D, the disk-resident training set of class-labeled tuples, does not fit in memory? In other words, how scalable is decision tree induction?” The efficiency of existing deci- sion tree algorithms, such as ID3, C4.5, and CART, has been well established for relatively small data sets. Efficiency becomes an issue of concern when these algorithms are applied to the mining of very large real-world databases.

The pioneering decision tree algorithms that we have discussed so far have the restriction that the training tuples should reside *in memory*. In data mining applications, very large training sets of millions of tuples are common. Most often, the training data will not fit in memory! Decision tree construction therefore becomes inefficient due to swapping of the training tuples in and out of main and cache memories. More scalable approaches, capable of handling training data that are too large to fit in memory, are required. Earlier strategies to “save space” included discretizing continuous-valued attributes and sampling data at each node. These techniques, however, still assume that the training set can fit in memory.

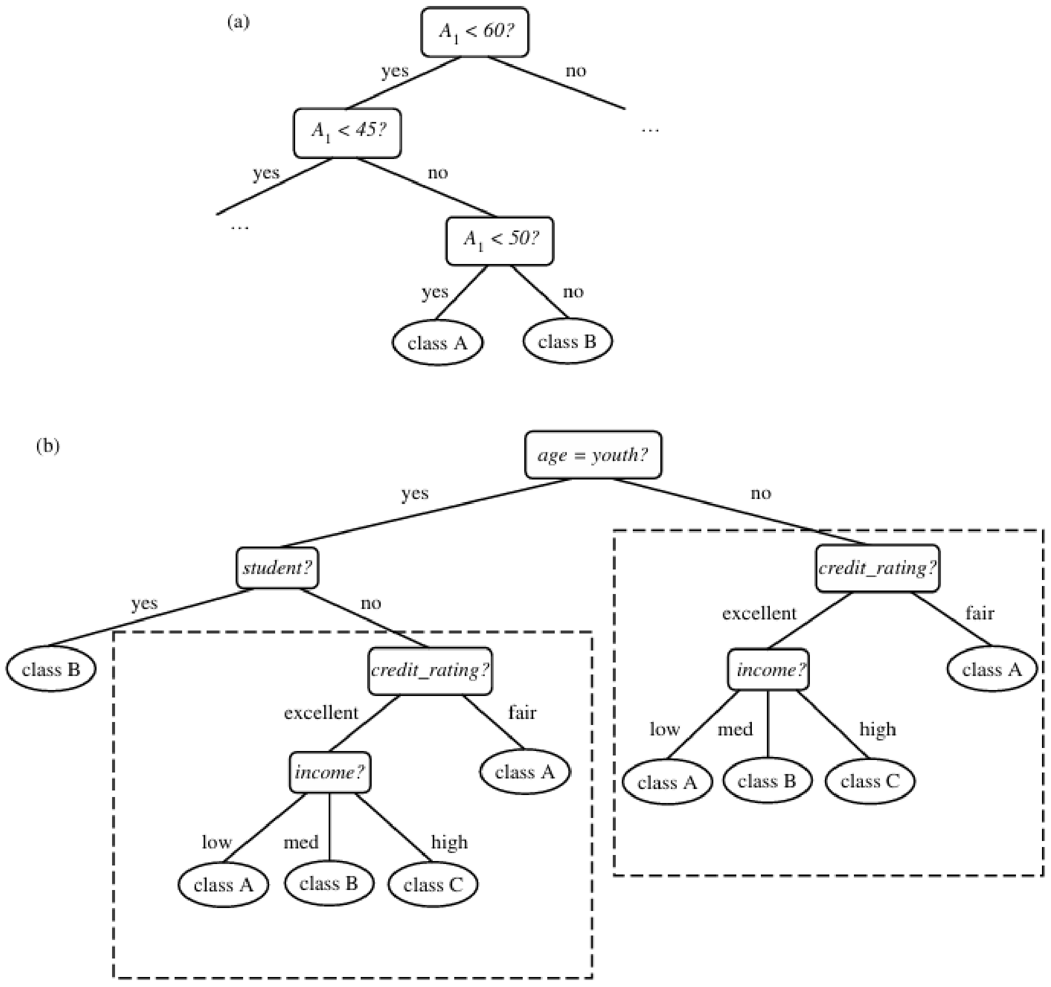
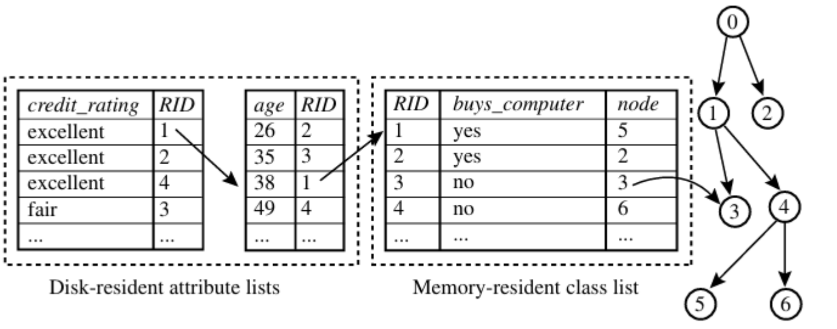


Figure 6.7 An example of subtree (a) **repetition** (where an attribute is repeatedly tested along a given branch of the tree, e.g., *age*) and (b) **replication** (where duplicate subtrees exist within a tree, such as the subtree headed by the node "*credit_rating?*").

- More recent decision tree algorithms that address the scalability issue have been proposed. Algorithms for the induction of decision trees from very large training sets include SLIQ and SPRINT, both of which can handle categorical and continuous valued attributes. Both algorithms propose presorting techniques on disk-resident data sets that are too large to fit in memory.
- Both define the use of new data structures to facilitate the tree construction. SLIQ employs disk-resident attribute lists and a single memory-resident class list. The attribute lists and class list generated by SLIQ for the tuple data of Table 6.2 are shown in Figure 6.8. Each attribute has an associated attribute list, indexed by RID (a record identifier). Each tuple is represented by a linkage of one entry from each attribute list to an entry in the class list (holding the class label of the given tuple), which in turn is linked to its corresponding leaf node

Table 6.2 Tuple data for the class *buys_computer*.

RID	*credit_rating*	*age*	*buys_computer*
1	excellent	38	yes
2	excellent	26	yes
3	fair	35	no
4	excellent	49	no
...	...	...	...



credit_rating	*RID*
excellent	1
excellent	2
excellent	4
fair	3
...	...

age	*RID*
26	2
35	3
38	1
49	4
...	...

RID	*buys_computer*	*node*
1	yes	5
2	yes	2
3	no	3
4	no	6
...	...	...

Figure 6.8 Attribute list and class list data structures used in SLIQ for the tuple data of Table 6.2.

credit_rating	*buys_computer*	*RID*
excellent	yes	1
excellent	yes	2
excellent	no	4
fair	no	3
...	...	...

age	*buys_computer*	*RID*
26	yes	2
35	no	3
38	yes	1
49	no	4
...	...	...

Figure 6.9 Attribute list data structure used in SPRINT for the tuple data of Table 6.2.

in the decision tree. The class list remains in memory because it is often accessed and modified in the building and pruning phases. The size of the class list grows proportionally with the number of tuples in the training set. When a class list cannot f it into memory, the performance of SLIQ decreases. SPRINT uses a different attribute list data structure that holds the class and RID information, as shown in Figure 6.9. When a node is split, the attribute lists are par titioned and distributed among the resulting child nodes accordingly. When a list is

age	*buys_computer*	
	yes	no
youth	2	3
middle_aged	4	0
senior	3	2

income	*buys_computer*	
	yes	no
low	3	1
medium	4	2
high	2	2

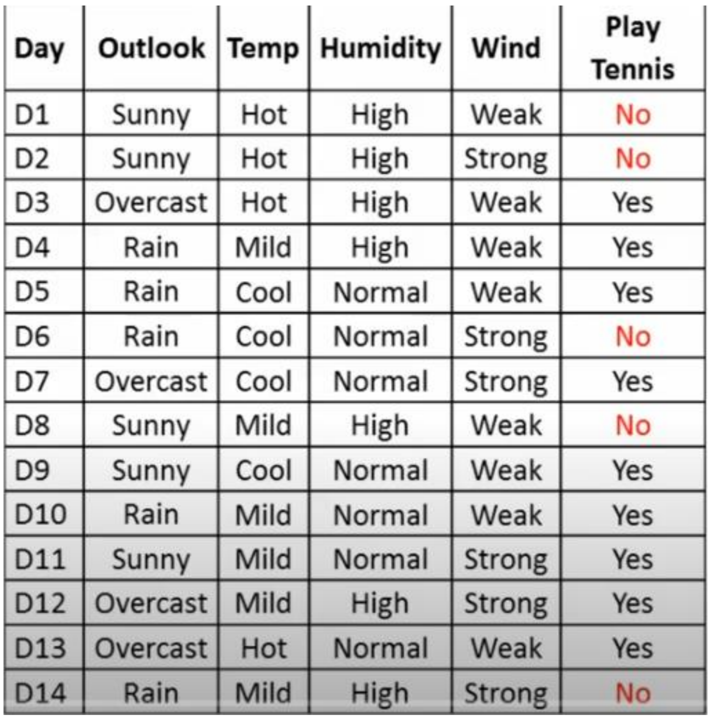
student	*buys_computer*	
	yes	no
yes	6	1
no	3	4

credit_rating	*buys_computer*	
	yes	no
fair	6	2
excellent	3	3

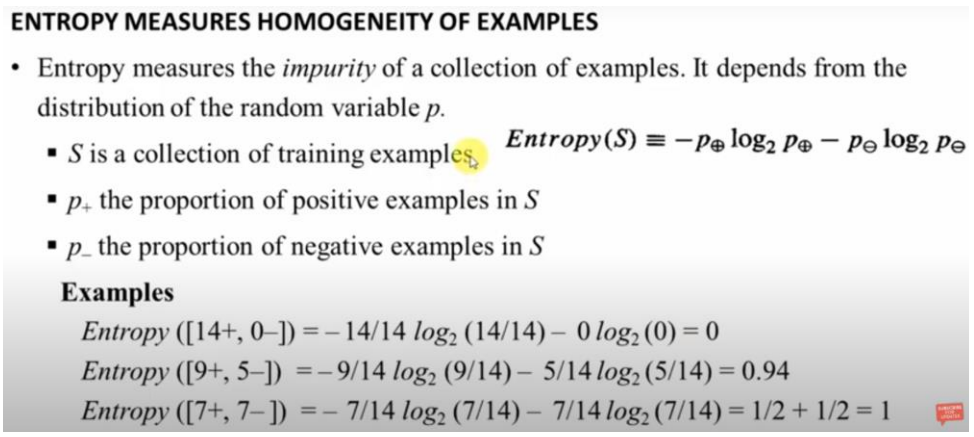
Figure 6.10 The use of data structures to hold aggregate information regarding the training data (such as these AVC-sets describing the data of Table 6.1) are one approach to improving the scalability of decision tree induction. partitioned, the order of the records in the list is maintained. Hence, partitioning lists does not require resorting. SPRINT was designed to be easily parallelized, further contributing to its scalability.

- While both SLIQ and SPRINT handle disk-residentdata sets that are too large to fit into memory, thescalability of SLIQ is limitedby theuse of its memory-residentdatastructure. SPRINT removes all memory restrictions, yet requires the use of a hash tree proportional in size to the training set. This may become expensive as the training set size grows.
- To further enhance the scalability of decision tree induction, a method called Rain- Forest was proposed. It adapts to the amount of main memory available and applies to any decision tree induction algorithm.
- The method maintains an **AVC-set** (where AVC stands for "Attribute-Value, Classlabel") for each attribute, at each tree node, describing the training tuples at the node. The AVC-set of an attribute *A* at node *N* gives the class label counts for each value of *A* for the tuples at *N*. Figure 6.10 shows AVC-sets for the tuple data of Table 6.1.
- The set of all AVC-sets at a node *N* is the **AVC-group** of *N*. The size of an AVC-set for attribute *A* at node *N* depends only on the number of distinct val- ues of *A*and the number of classes in the set of tuples at *N*. Typically, this size should fit in memory, even for real-world data. RainForest has techniques, however, for handling the case where the AVC-group does not fit in memory. RainForest can use any attribute selection measure and was shown to be more efficient than earlier approaches employing aggregate data structures, such as SLIQ and SPRINT.
- BOAT (Bootstrapped Optimistic Algorithm for Tree Construction) is a decision tree algorithm that takes a completely different approach to scalability—it is not based on the use of any special data structures. Instead, it uses a statistical technique known as "boot- strapping" (Section 6.13.3) to create several smaller samples (or subsets) of the given training data, each of which fits in memory.
- Each subset is used to construct a tree, resulting in several trees. The trees are examined and used to construct a new tree, T^r, that turns out to be "very close" to the tree that would have been generated if all of the original training data had fit in memory. BOAT can use any attribute selection measure that selects binary splits and that is based on the notion of purity of partitions, such as the gini index.
- BOAT uses a lower bound on the attribute selection measure in order to detect if this "very good" tree, T^r, is different from the "real" tree, *T*, that would have been generated using the entire data. It refines T^r in order to arrive at *T*.
- BOAT usually requires only two scans of *D*. This is quite an improvement, even in comparison to traditional decision tree algorithms (such as the basic algorithm in Figure 6.3), which require one scan per level of the tree! BOAT was found to be two to three times faster than RainForest, while constructing exactly the same tree. An addi- tional advantage of BOAT is that it can be used for incremental updates. That is, BOAT can take new insertions and deletions for the training data and update the decision tree to reflect these changes, without having to reconstruct the tree from scratch.

Example to solve decision tree



Day	Outlook	Temp	Humidity	Wind	Play Tennis
D1	Sunny	Hot	High	Weak	No
D2	Sunny	Hot	High	Strong	No
D3	Overcast	Hot	High	Weak	Yes
D4	Rain	Mild	High	Weak	Yes
D5	Rain	Cool	Normal	Weak	Yes
D6	Rain	Cool	Normal	Strong	No
D7	Overcast	Cool	Normal	Strong	Yes
D8	Sunny	Mild	High	Weak	No
D9	Sunny	Cool	Normal	Weak	Yes
D10	Rain	Mild	Normal	Weak	Yes
D11	Sunny	Mild	Normal	Strong	Yes
D12	Overcast	Mild	High	Strong	Yes
D13	Overcast	Hot	Normal	Weak	Yes
D14	Rain	Mild	High	Strong	No



ENTROPY MEASURES HOMOGENEITY OF EXAMPLES

- Entropy measures the *impurity* of a collection of examples. It depends from the distribution of the random variable *p*.
  - S is a collection of training examples
  - p_+ the proportion of positive examples in S
  - p_- the proportion of negative examples in S

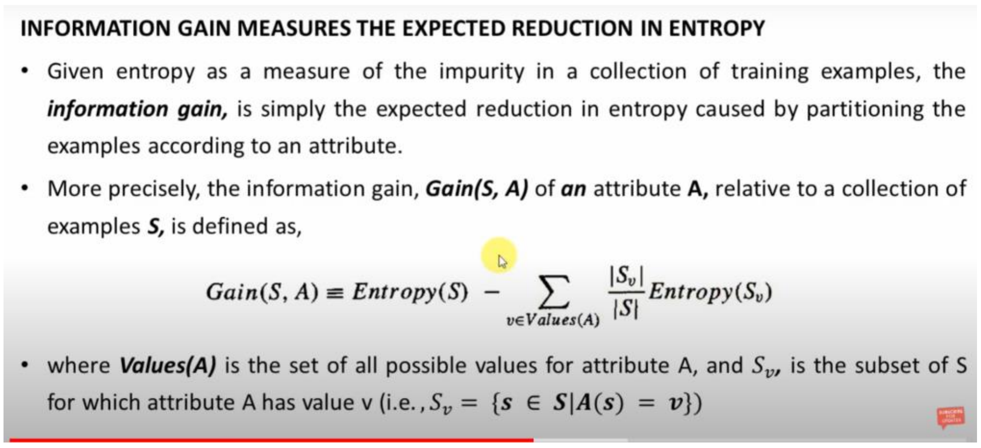
$$Entropy(S) \equiv -p_{\oplus}\log_2 p_{\oplus} - p_{\ominus}\log_2 p_{\ominus}$$

Examples

$Entropy\ ([14+, 0-]) = -14/14\ log_2\ (14/14) - 0\ log_2\ (0) = 0$

$Entropy\ ([9+, 5-]) = -9/14\ log_2\ (9/14) - 5/14\ log_2\ (5/14) = 0.94$

$Entropy\ ([7+, 7-]) = -7/14\ log_2\ (7/14) - 7/14\ log_2\ (7/14) = 1/2 + 1/2 = 1$

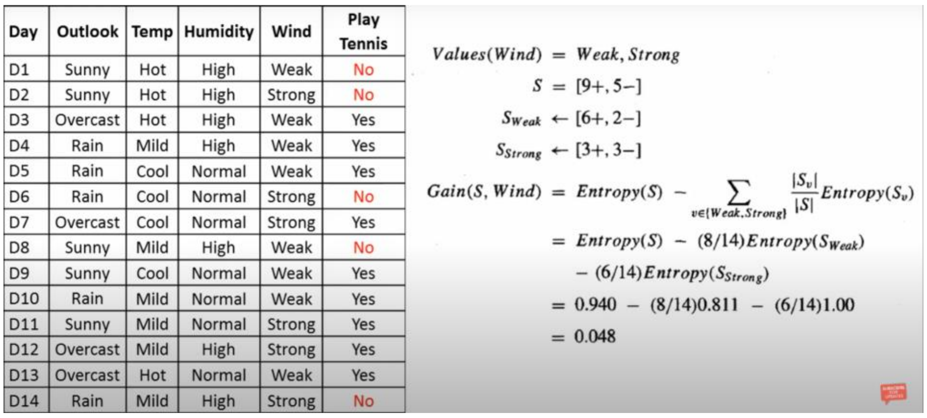


INFORMATION GAIN MEASURES THE EXPECTED REDUCTION IN ENTROPY

- Given entropy as a measure of the impurity in a collection of training examples, the ***information gain,*** is simply the expected reduction in entropy caused by partitioning the examples according to an attribute.
- More precisely, the information gain, ***Gain(S, A)*** of ***an*** attribute **A,** relative to a collection of examples ***S,*** is defined as,

$$Gain(S, A) \equiv Entropy(S) - \sum_{v \in Values(A)} \frac{|S_v|}{|S|} Entropy(S_v)$$

- where ***Values(A)*** is the set of all possible values for attribute A, and S_v, is the subset of S for which attribute A has value v (i.e., $S_v = \{s \in S | A(s) = v\}$)



Day	Outlook	Temp	Humidity	Wind	Play Tennis
D1	Sunny	Hot	High	Weak	No
D2	Sunny	Hot	High	Strong	No
D3	Overcast	Hot	High	Weak	Yes
D4	Rain	Mild	High	Weak	Yes
D5	Rain	Cool	Normal	Weak	Yes
D6	Rain	Cool	Normal	Strong	No
D7	Overcast	Cool	Normal	Strong	Yes
D8	Sunny	Mild	High	Weak	No
D9	Sunny	Cool	Normal	Weak	Yes
D10	Rain	Mild	Normal	Weak	Yes
D11	Sunny	Mild	Normal	Strong	Yes
D12	Overcast	Mild	High	Strong	Yes
D13	Overcast	Hot	Normal	Weak	Yes
D14	Rain	Mild	High	Strong	No

$$Values(Wind) = Weak, Strong$$

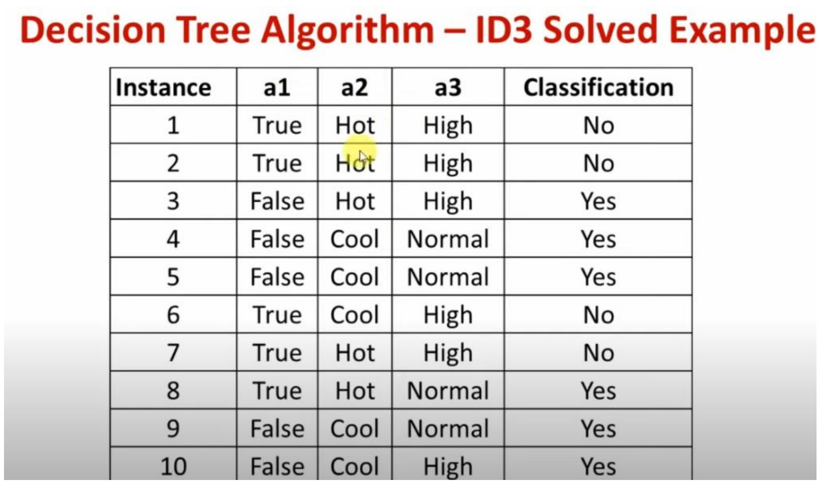
$$S = [9+, 5-]$$

$$S_{Weak} \leftarrow [6+, 2-]$$

$$S_{Strong} \leftarrow [3+, 3-]$$

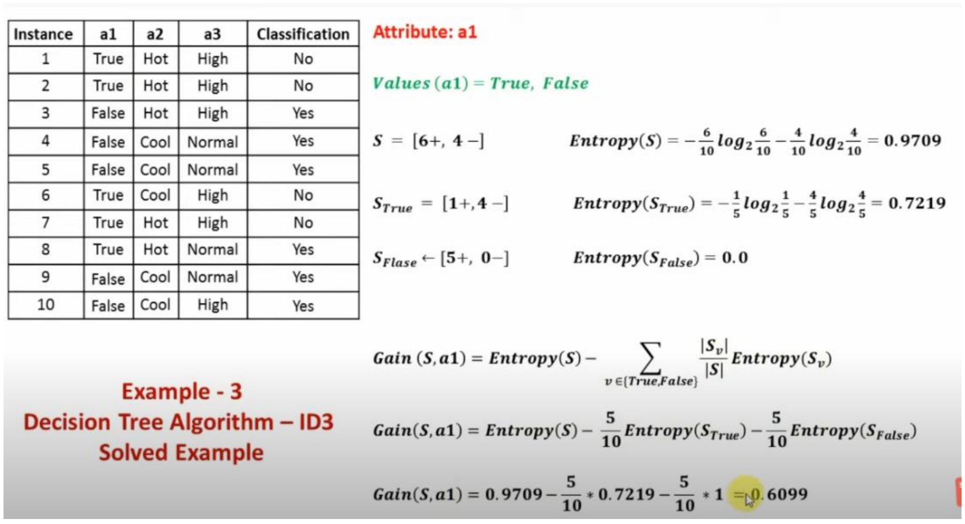
$$\begin{aligned} Gain(S, Wind) &= Entropy(S) - \sum_{v \in \{Weak, Strong\}} \frac{|S_v|}{|S|} Entropy(S_v) \\ &= Entropy(S) - (8/14) Entropy(S_{Weak}) \\ &\quad - (6/14) Entropy(S_{Strong}) \\ &= 0.940 - (8/14)0.811 - (6/14)1.00 \\ &= 0.048 \end{aligned}$$

ID3 Example:



Decision Tree Algorithm – ID3 Solved Example

Instance	a1	a2	a3	Classification
1	True	Hot	High	No
2	True	Hot	High	No
3	False	Hot	High	Yes
4	False	Cool	Normal	Yes
5	False	Cool	Normal	Yes
6	True	Cool	High	No
7	True	Hot	High	No
8	True	Hot	Normal	Yes
9	False	Cool	Normal	Yes
10	False	Cool	High	Yes



Instance	a1	a2	a3	Classification
1	True	Hot	High	No
2	True	Hot	High	No
3	False	Hot	High	Yes
4	False	Cool	Normal	Yes
5	False	Cool	Normal	Yes
6	True	Cool	High	No
7	True	Hot	High	No
8	True	Hot	Normal	Yes
9	False	Cool	Normal	Yes
10	False	Cool	High	Yes

Example - 3
Decision Tree Algorithm – ID3
Solved Example

Attribute: a1

$Values(a1) = True,\ False$

$S = [6+,\ 4-]$ $\qquad Entropy(S) = -\frac{6}{10} log_2 \frac{6}{10} - \frac{4}{10} log_2 \frac{4}{10} = 0.9709$

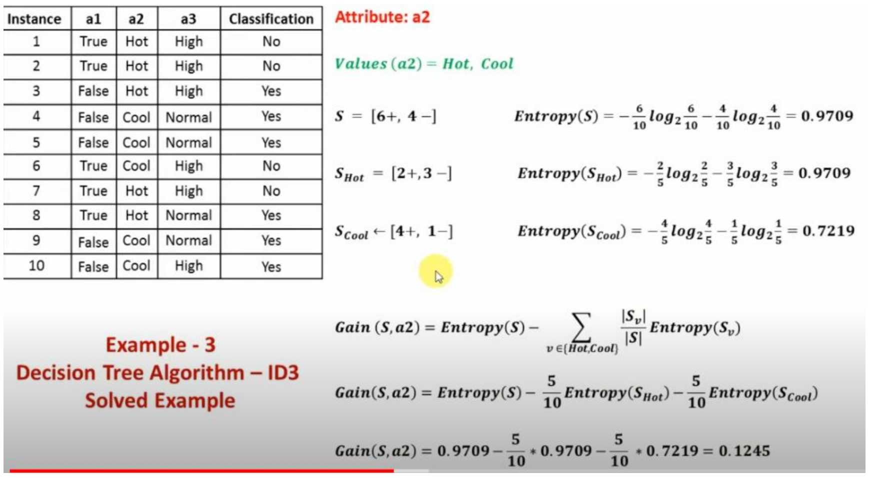
$S_{True} = [1+, 4-]$ $\qquad Entropy(S_{True}) = -\frac{1}{5} log_2 \frac{1}{5} - \frac{4}{5} log_2 \frac{4}{5} = 0.7219$

$S_{Flase} \leftarrow [5+,\ 0-]$ $\qquad Entropy(S_{False}) = 0.0$

$$Gain\ (S, a1) = Entropy(S) - \sum_{v \in \{True, False\}} \frac{|S_v|}{|S|} Entropy(S_v)$$

$$Gain(S, a1) = Entropy(S) - \frac{5}{10} Entropy(S_{True}) - \frac{5}{10} Entropy(S_{False})$$

$$Gain(S, a1) = 0.9709 - \frac{5}{10} * 0.7219 - \frac{5}{10} * 1 = 0.6099$$



Instance	a1	a2	a3	Classification
1	True	Hot	High	No
2	True	Hot	High	No
3	False	Hot	High	Yes
4	False	Cool	Normal	Yes
5	False	Cool	Normal	Yes
6	True	Cool	High	No
7	True	Hot	High	No
8	True	Hot	Normal	Yes
9	False	Cool	Normal	Yes
10	False	Cool	High	Yes

Attribute: a2

$Values\,(a2) = Hot,\ Cool$

$S = [6+,\ 4-]$ $\quad Entropy(S) = -\frac{6}{10}\log_2\frac{6}{10} - \frac{4}{10}\log_2\frac{4}{10} = 0.9709$

$S_{Hot} = [2+, 3-]$ $\quad Entropy(S_{Hot}) = -\frac{2}{5}\log_2\frac{2}{5} - \frac{3}{5}\log_2\frac{3}{5} = 0.9709$

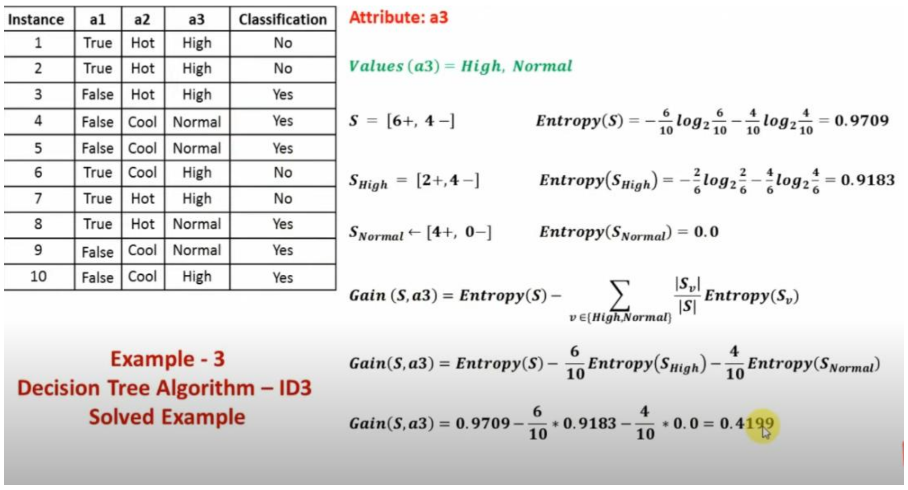
$S_{Cool} \leftarrow [4+,\ 1-]$ $\quad Entropy(S_{Cool}) = -\frac{4}{5}\log_2\frac{4}{5} - \frac{1}{5}\log_2\frac{1}{5} = 0.7219$

$$Gain\,(S, a2) = Entropy(S) - \sum_{v \in \{Hot, Cool\}} \frac{|S_v|}{|S|} Entropy(S_v)$$

$$Gain(S, a2) = Entropy(S) - \frac{5}{10} Entropy(S_{Hot}) - \frac{5}{10} Entropy(S_{Cool})$$

$$Gain(S, a2) = 0.9709 - \frac{5}{10} * 0.9709 - \frac{5}{10} * 0.7219 = 0.1245$$

Example - 3
Decision Tree Algorithm – ID3
Solved Example



Instance	a1	a2	a3	Classification
1	True	Hot	High	No
2	True	Hot	High	No
3	False	Hot	High	Yes
4	False	Cool	Normal	Yes
5	False	Cool	Normal	Yes
6	True	Cool	High	No
7	True	Hot	High	No
8	True	Hot	Normal	Yes
9	False	Cool	Normal	Yes
10	False	Cool	High	Yes

Attribute: a3

$Values\,(a3) = High,\ Normal$

$S = [6+,\ 4-]$ $\quad Entropy(S) = -\frac{6}{10}\log_2\frac{6}{10} - \frac{4}{10}\log_2\frac{4}{10} = 0.9709$

$S_{High} = [2+, 4-]$ $\quad Entropy(S_{High}) = -\frac{2}{6}\log_2\frac{2}{6} - \frac{4}{6}\log_2\frac{4}{6} = 0.9183$

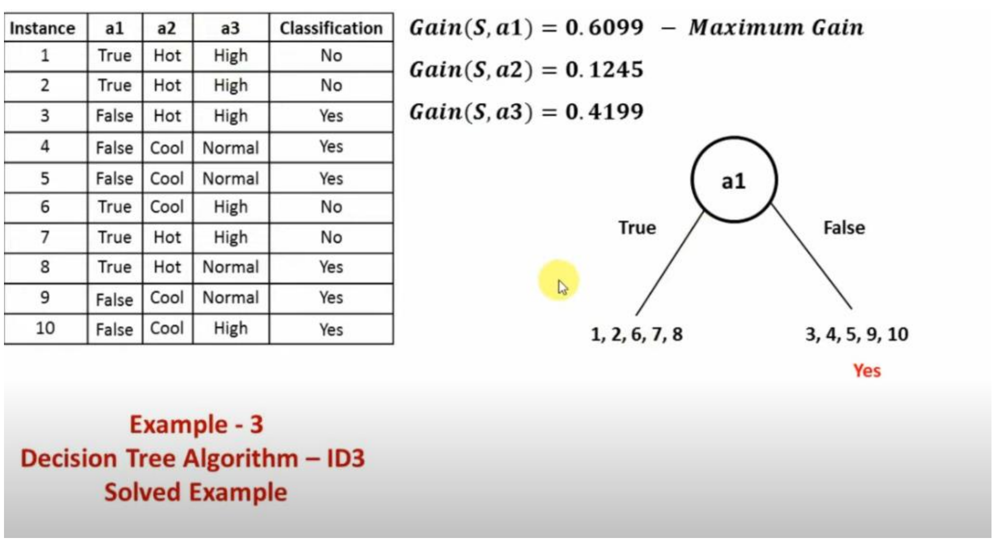
$S_{Normal} \leftarrow [4+,\ 0-]$ $\quad Entropy(S_{Normal}) = 0.0$

$$Gain\,(S, a3) = Entropy(S) - \sum_{v \in \{High, Normal\}} \frac{|S_v|}{|S|} Entropy(S_v)$$

$$Gain(S, a3) = Entropy(S) - \frac{6}{10} Entropy(S_{High}) - \frac{4}{10} Entropy(S_{Normal})$$

$$Gain(S, a3) = 0.9709 - \frac{6}{10} * 0.9183 - \frac{4}{10} * 0.0 = 0.4199$$

Example - 3
Decision Tree Algorithm – ID3
Solved Example



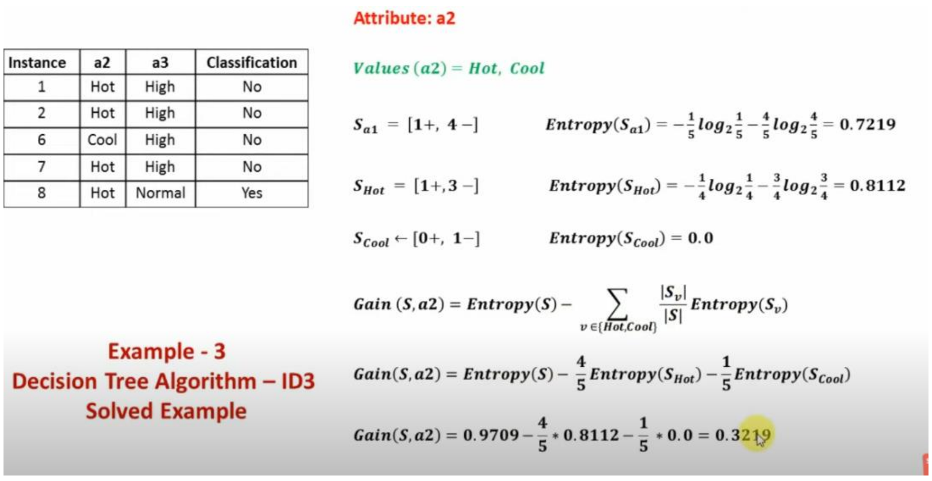
Instance	a1	a2	a3	Classification
1	True	Hot	High	No
2	True	Hot	High	No
3	False	Hot	High	Yes
4	False	Cool	Normal	Yes
5	False	Cool	Normal	Yes
6	True	Cool	High	No
7	True	Hot	High	No
8	True	Hot	Normal	Yes
9	False	Cool	Normal	Yes
10	False	Cool	High	Yes

$Gain(S, a1) = 0.6099$ – *Maximum Gain*

$Gain(S, a2) = 0.1245$

$Gain(S, a3) = 0.4199$

Example - 3
Decision Tree Algorithm – ID3
Solved Example



Attribute: a2

Instance	a2	a3	Classification
1	Hot	High	No
2	Hot	High	No
6	Cool	High	No
7	Hot	High	No
8	Hot	Normal	Yes

$Values\,(a2) = Hot,\ Cool$

$S_{a1} = [1+,\ 4-]$ $\qquad Entropy(S_{a1}) = -\frac{1}{5}log_2\frac{1}{5} - \frac{4}{5}log_2\frac{4}{5} = 0.7219$

$S_{Hot} = [1+, 3-]$ $\qquad Entropy(S_{Hot}) = -\frac{1}{4}log_2\frac{1}{4} - \frac{3}{4}log_2\frac{3}{4} = 0.8112$

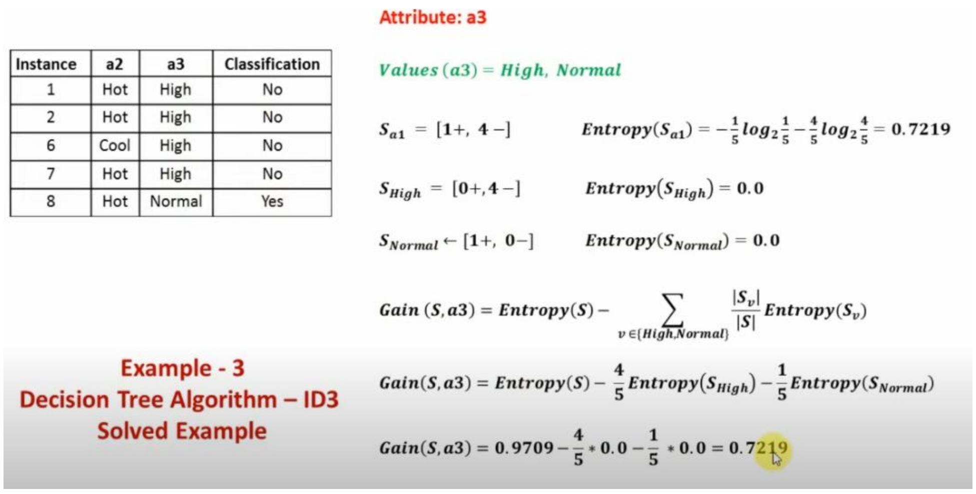
$S_{Cool} \leftarrow [0+,\ 1-]$ $\qquad Entropy(S_{Cool}) = 0.0$

$$Gain\,(S, a2) = Entropy(S) - \sum_{v\in\{Hot,Cool\}} \frac{|S_v|}{|S|} Entropy(S_v)$$

$$Gain(S, a2) = Entropy(S) - \frac{4}{5}Entropy(S_{Hot}) - \frac{1}{5}Entropy(S_{Cool})$$

$$Gain(S, a2) = 0.9709 - \frac{4}{5} * 0.8112 - \frac{1}{5} * 0.0 = 0.3219$$

Example - 3
Decision Tree Algorithm – ID3
Solved Example



Attribute: a3

Instance	a2	a3	Classification
1	Hot	High	No
2	Hot	High	No
6	Cool	High	No
7	Hot	High	No
8	Hot	Normal	Yes

$Values\,(a3) = High,\ Normal$

$S_{a1} = [1+,\ 4-]$ $\qquad Entropy(S_{a1}) = -\frac{1}{5}log_2\frac{1}{5} - \frac{4}{5}log_2\frac{4}{5} = 0.7219$

$S_{High} = [0+, 4-]$ $\qquad Entropy(S_{High}) = 0.0$

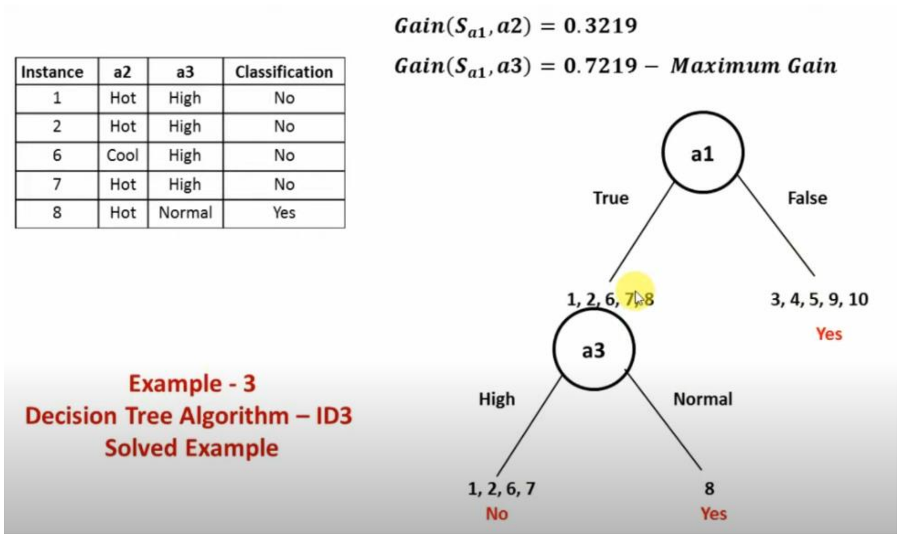
$S_{Normal} \leftarrow [1+,\ 0-]$ $\qquad Entropy(S_{Normal}) = 0.0$

$$Gain\,(S, a3) = Entropy(S) - \sum_{v\in\{High,Normal\}} \frac{|S_v|}{|S|} Entropy(S_v)$$

$$Gain(S, a3) = Entropy(S) - \frac{4}{5}Entropy(S_{High}) - \frac{1}{5}Entropy(S_{Normal})$$

$$Gain(S, a3) = 0.9709 - \frac{4}{5} * 0.0 - \frac{1}{5} * 0.0 = 0.7219$$

Example - 3
Decision Tree Algorithm – ID3
Solved Example



$Gain(S_{a1}, a2) = 0.3219$

$Gain(S_{a1}, a3) = 0.7219 -$ $Maximum\ Gain$

Instance	a2	a3	Classification
1	Hot	High	No
2	Hot	High	No
6	Cool	High	No
7	Hot	High	No
8	Hot	Normal	Yes

Example - 3
Decision Tree Algorithm – ID3
Solved Example

Bayesian Classification

- *"What are Bayesian classifiers?"* Bayesian classifiers are statistical classifiers. They can pre- dict class membership probabilities, such as the probability that a given tuple belongs to a particular class.
- Bayesian classification is based on Bayes' theorem, described below. Studies compar- ing classification algorithms have found a simple Bayesian classifier known as the *naive Bayesian classifier* to be comparable in performance with decision tree and selected neu- ral network classifiers. Bayesian classifiers have also exhibited high accuracy and speed when applied to large databases.
- Naïve Bayesian classifiers assume that the effect of an attribute value on a given class is independent of the values of the other attributes. This assumption is called *class condi- tional independence*. It is made to simplify the computations involved and, in this sense, is considered "naïve." *Bayesian belief networks* are graphical models, which unlike naïve Bayesian classifiers, allow the representation of dependencies among subsets of attributes. Bayesian belief networks can also be used for classification.

  $P(H|\boldsymbol{X})$ is the **posterior probability**, or *a posteriori probability*, of H conditioned on$\boldsymbol{X}$. For example, suppose our world of data tuples is confined to customers described by

6.3.4 Bayes' Theorem

- Bayes' theorem is named after Thomas Bayes, a nonconformist English clergyman who did early work in probability and decision theory during the 18th century. Let $\boldsymbol{X}$ be a data tuple. In Bayesian terms, $\boldsymbol{X}$ is considered "evidence."
- As usual, it is described by measurements made on a set of n attributes. Let H be some hypothesis, such as that the data tuple $\boldsymbol{X}$ belongs to a specified class C.
- For classification problems, we want to determine $P(H| \boldsymbol{X})$, the probability that the hypothesis H holds given the "evidence" or observed data tuple $\boldsymbol{X}$. In other words, we are looking for the probability that tuple $\boldsymbol{X}$ belongs to class C, given that we know the attribute description of $\boldsymbol{X}$.

the attributes *age* and *income*, respectively, and that X is a 35-year-old customer with an income of $40,000. Suppose that H is the hypothesis that our customer will buy a computer. Then $P(H|X)$ reflects the probability that customer X will buy a computer given that we know the customer's age and income.

In contrast, $P(H)$ is the **prior probability**, or *a priori probability*, of H. For example, this is the probability that any given customer will buy a computer, regardless of age, income, or any other information, for that matter. The posterior probability, $P(H|X)$, is based on more information (e.g., customer information) than the prior probability, $P(H)$, which is independent of X.

Similarly, $P(X|H)$ is the posterior probability of X conditioned on H. That is, it is the probability that a customer, X, is 35 years old and earns $40,000, given that we know the customer will buy a computer.

$P(X)$ is the prior probability of X. Using our example, it is the probability that a person from our set of customers is 35 years old and earns $40,000.

"How are these probabilities estimated?" $P(H)$, $P(X|H)$, and $P(X)$ may be estimated from the given data, as we shall see below. **Bayes' theorem** is useful in that it provides a way of calculating the posterior probability, $P(H|X)$, from $P(H)$, $P(X|H)$, and $P(X)$. Bayes' theorem is

$$P(H|X) = \frac{P(X|H)P(H)}{P(X)}. \qquad (6.10)$$

Now that we've got that out of the way, in the next section, we will look at how Bayes' theorem is used in the naive Bayesian classifier.

6.3.1 Naïve Bayesian Classification

The **naïve Bayesian** classifier, or **simple Bayesian** classifier, works as follows:

1. Let D be a training set of tuples and their associated class labels. As usual, each tuple is represented by an n-dimensional attribute vector, $\boldsymbol{X} = (x_1, x_2, \ldots, x_n)$, depicting n measurements made on the tuple from n attributes, respectively, $A_1, A_2, \ldots, A_n$.
2. Suppose that there are m classes, $C_1, C_2, \ldots, C_m$. Given a tuple, $\boldsymbol{X}$, the classifier will predict that $\boldsymbol{X}$ belongs to the class having the highest posterior probability, condi- tioned on $\boldsymbol{X}$. That is, the naïve Bayesian classifier predicts that tuple $\boldsymbol{X}$ belongs to the class C_i if and only if

$P(C_i|\boldsymbol{X}) > P(C_j|\boldsymbol{X})$ for $1 \le j \le m, j$ *not equal to i.*

Thus we maximize $P(C_i|\boldsymbol{X})$. The class C_i for which $P(C_i|\boldsymbol{X})$ is maximized is called the *maximum posteriori hypothesis*. By Bayes' theorem (Equation (6.10)),

$$P(C_i|\boldsymbol{X}) = \frac{P(\boldsymbol{X}|C_i)P(C_i)}{P(\boldsymbol{X})}. \tag{6.11}$$

3. As $P(\boldsymbol{X})$ is constant for all classes, only $P(\boldsymbol{X}|C_i)P(C_i)$ need be maximized. If the class prior probabilities are not known, then it is commonly assumed that the classes are equally likely, that is, $P(C_1) = P(C_2) = \quad = P(C_m)$, and we would therefore maxi- mize

$P(\boldsymbol{X}|C_i)$. Otherwise, we maximize $P(\boldsymbol{X}|C_i)P(C_i)$. Note that the class prior prob- abilities may be estimated by $P(C_i) = |C_{i,D}| / D$, where $|C_{i,D}|$ is the number of training tuples of class C_i in D.

4. Given data sets with many attributes, it would be extremely computationally expensive to compute $P(\boldsymbol{X}|C_i)$. In order to reduce computation in evaluating $P(\boldsymbol{X}|C_i)$, the naive assumption of **class conditional independence** is made. This presumes that the values of the attributes are conditionally independent of one another, given the class label of the tuple (i.e., that there are no dependence relationships among the attributes). Thus,

$$\begin{aligned} P(\boldsymbol{X}|C_i) &= \prod_{k=1}^{n} P(x_k|C_i) \qquad (6.12) \\ &= P(x_1|C_i) \times P(x_2|C_i) \times \cdots \times P(x_n|C_i). \end{aligned}$$

We can easily estimate the probabilities $P(x_1 C_i), P(x_2 C_i), \ldots, P(x_n C_i)$ from the train- ing tuples. Recall that here x_k refers to the value of attribute A_k for tuple $\boldsymbol{X}$. For each attribute, we look at whether the attribute is categorical or continuous-valued. For instance, to compute $P(\boldsymbol{X}|C_i)$, we consider the following:

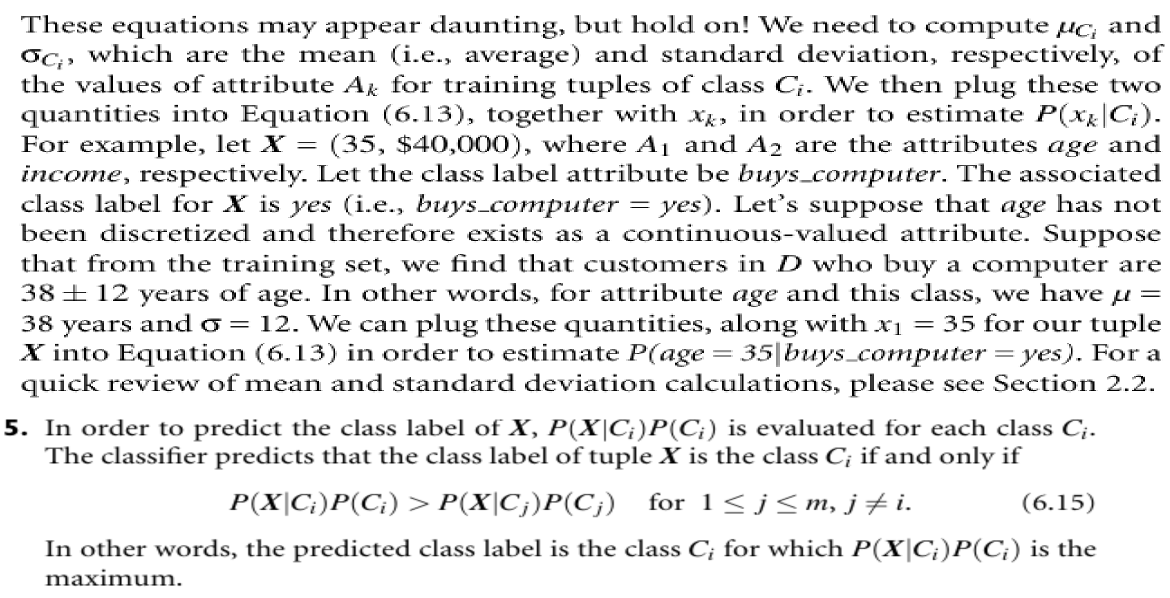
(a) If A_k is categorical, then $P(x_k|C_i)$ is the number of tuples of class C_i in D having the value x_k for A_k, divided by $|C_{i,D}|$, the number of tuples of class C_i in D.

(b) If A_k is continuous-valued, then we need to do a bit more work, but the calculation is pretty straightforward. A continuous-valued attribute is typically assumed to have a Gaussian distribution with a mean μ and standard deviation □, defined by

$$g(x, \mu, \sigma) = \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{(x-\mu)^2}{2\sigma^2}}, \tag{6.13}$$

so that

$$P(x_k|C_i) = g(x_k, \mu_{C_i}, \sigma_{C_i}). \tag{6.14}$$


These equations may appear daunting, but hold on! We need to compute μ_{C_i} and σ_{C_i}, which are the mean (i.e., average) and standard deviation, respectively, of the values of attribute A_k for training tuples of class C_i. We then plug these two quantities into Equation (6.13), together with x_k, in order to estimate $P(x_k|C_i)$. For example, let $\boldsymbol{X} = (35, \$40{,}000)$, where A_1 and A_2 are the attributes *age* and *income*, respectively. Let the class label attribute be *buys_computer*. The associated class label for $\boldsymbol{X}$ is *yes* (i.e., *buys_computer* = *yes*). Let's suppose that *age* has not been discretized and therefore exists as a continuous-valued attribute. Suppose that from the training set, we find that customers in D who buy a computer are 38 ± 12 years of age. In other words, for attribute *age* and this class, we have $\mu = 38$ years and $\sigma = 12$. We can plug these quantities, along with $x_1 = 35$ for our tuple $\boldsymbol{X}$ into Equation (6.13) in order to estimate $P(age = 35|buys\_computer = yes)$. For a quick review of mean and standard deviation calculations, please see Section 2.2.

5. In order to predict the class label of $\boldsymbol{X}$, $P(\boldsymbol{X}|C_i)P(C_i)$ is evaluated for each class C_i. The classifier predicts that the class label of tuple $\boldsymbol{X}$ is the class C_i if and only if

$$P(\boldsymbol{X}|C_i)P(C_i) > P(\boldsymbol{X}|C_j)P(C_j) \quad \text{for } 1 \leq j \leq m,\ j \neq i. \qquad (6.15)$$

In other words, the predicted class label is the class C_i for which $P(\boldsymbol{X}|C_i)P(C_i)$ is the maximum.

- *"How effective are Bayesian classifiers?"* Various empirical studies of this classifier in comparison to decision tree and neural network classifiers have found it to be comparable in some domains. In theory, Bayesian classifiers have the minimum error rate in comparison to all other classifiers. However, in practice this is not always the case, owing to inaccuracies in the assumptions made for its use, such as class conditional independence, and the lack of available probability data.
- Bayesian classifiers are also useful in that they provide a theoretical justification for other classifiers that do not explicitly use Bayes' theorem. For example, under certain assumptions, it can be shown that many neural network and curve-fitting algorithms output the *maximum posteriori* hypothesis, as does the naïve Bayesian classifier.

Example 6.4 Predicting a class label using naïve Bayesian classification. We wish to predict the class label of a tuple using naïve Bayesian classification, given the same training data as in Example 6.3 for decision tree induction. The training data are in Table 6.1. The data tuples are described by the attributes *age*, *income*, *student*, and *credit_rating*. The class label attribute, *buys_computer*, has two distinct values (namely, {*yes, no*}). Let C_1 correspond to the class *buys_computer* = *yes* and C_2 correspond to *buys_computer* = *no.* The tuple we wish to classify is

$$\boldsymbol{X} = (age = youth,\ income = medium,\ student = yes,\ credit\_rating = fair)$$

We need to maximize $P(\boldsymbol{X}|C_i)P(C_i)$, for $i = 1, 2$. $P(C_i)$, the prior probability of each class, can be computed based on the training tuples:

$$P(buys\_computer = yes) = 9/14 = 0.643$$
$$P(buys\_computer = no) = 5/14 = 0.357$$

To compute $P\boldsymbol{X}|C_i)$, for $i = 1, 2$, we compute the following conditional probabilities:

$$\begin{aligned}
&P(age = youth \mid buys\_computer = yes) &&= 2/9 = 0.222\\
&P(age = youth \mid buys\_computer = no) &&= 3/5 = 0.600\\
&P(income = medium \mid buys\_computer = yes) &&= 4/9 = 0.444\\
&P(income = medium \mid buys\_computer = no) &&= 2/5 = 0.400\\
&P(student = yes \mid buys\_computer = yes) &&= 6/9 = 0.667\\
&P(student = yes \mid buys\_computer = no) &&= 1/5 = 0.200\\
&P(credit\_rating = fair \mid buys\_computer = yes) &&= 6/9 = 0.667\\
&P(credit\_rating = fair \mid buys\_computer = no) &&= 2/5 = 0.400
\end{aligned}$$

Using the above probabilities, we obtain

$$\begin{aligned}
P(\boldsymbol{X}|buys\_computer = yes) = {} & P(age = youth \mid buys\_computer = yes) \times\\
& P(income = medium \mid buys\_computer = yes) \times\\
& P(student = yes \mid buys\_computer = yes) \times\\
& P(credit\_rating = fair \mid buys\_computer = yes)\\
= {} & 0.222 \times 0.444 \times 0.667 \times 0.667 = 0.044.
\end{aligned}$$

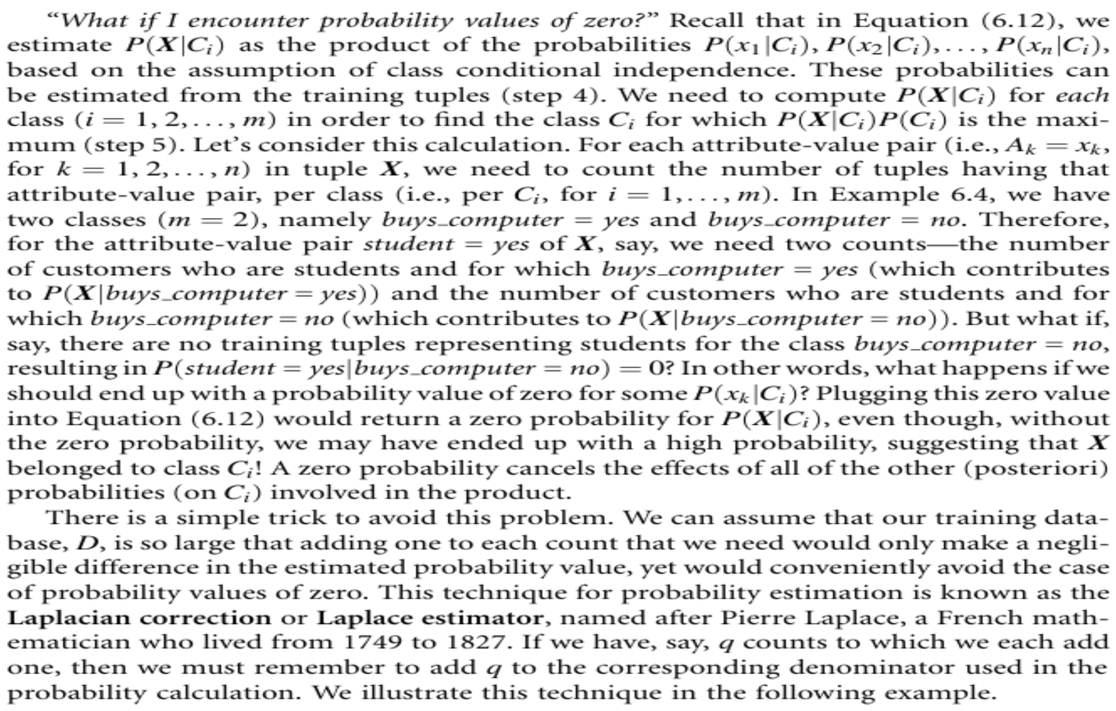
Similarly,

$$P(\boldsymbol{X}|buys\_computer = no) = 0.600 \times 0.400 \times 0.200 \times 0.400 = 0.019.$$

To find the class, C_i, that maximizes $P(\boldsymbol{X}|C_i)P(C_i)$, we compute

$$P(\boldsymbol{X}|buys\_computer = yes)P(buys\_computer = yes) = 0.044 \times 0.643 = 0.028$$
$$P(\boldsymbol{X}|buys\_computer = no)P(buys\_computer = no) = 0.019 \times 0.357 = 0.007$$

Therefore, the naïve Bayesian classifier predicts *buys_computer* = *yes* for tuple $\boldsymbol{X}$.



"What if I encounter probability values of zero?" Recall that in Equation (6.12), we estimate $P(\mathbf{X}|C_i)$ as the product of the probabilities $P(x_1|C_i), P(x_2|C_i), \ldots, P(x_n|C_i)$, based on the assumption of class conditional independence. These probabilities can be estimated from the training tuples (step 4). We need to compute $P(\mathbf{X}|C_i)$ for *each* class ($i = 1, 2, \ldots, m$) in order to find the class C_i for which $P(\mathbf{X}|C_i)P(C_i)$ is the maximum (step 5). Let's consider this calculation. For each attribute-value pair (i.e., $A_k = x_k$, for $k = 1, 2, \ldots, n$) in tuple $\mathbf{X}$, we need to count the number of tuples having that attribute-value pair, per class (i.e., per C_i, for $i = 1, \ldots, m$). In Example 6.4, we have two classes ($m = 2$), namely *buys_computer* = *yes* and *buys_computer* = *no*. Therefore, for the attribute-value pair *student* = *yes* of $\mathbf{X}$, say, we need two counts—the number of customers who are students and for which *buys_computer* = *yes* (which contributes to $P(\mathbf{X}|buys\_computer = yes)$) and the number of customers who are students and for which *buys_computer* = *no* (which contributes to $P(\mathbf{X}|buys\_computer = no)$). But what if, say, there are no training tuples representing students for the class *buys_computer* = *no*, resulting in $P(student = yes|buys\_computer = no) = 0$? In other words, what happens if we should end up with a probability value of zero for some $P(x_k|C_i)$? Plugging this zero value into Equation (6.12) would return a zero probability for $P(\mathbf{X}|C_i)$, even though, without the zero probability, we may have ended up with a high probability, suggesting that $\mathbf{X}$ belonged to class C_i! A zero probability cancels the effects of all of the other (posteriori) probabilities (on C_i) involved in the product.

There is a simple trick to avoid this problem. We can assume that our training database, D, is so large that adding one to each count that we need would only make a negligible difference in the estimated probability value, yet would conveniently avoid the case of probability values of zero. This technique for probability estimation is known as the **Laplacian correction** or **Laplace estimator**, named after Pierre Laplace, a French mathematician who lived from 1749 to 1827. If we have, say, q counts to which we each add one, then we must remember to add q to the corresponding denominator used in the probability calculation. We illustrate this technique in the following example.

Example 6.5 Using the Laplacian correction to avoid computing probability values of zero. Suppose that for the class *buys computer* = *yes* in some training database, *D*, containing 1,000 tuples, we have 0 tuples with *income* = *low*, 990 tuples with *income* = *medium*, and 10

tuples with *income* = *high*. The probabilities of these events, without the Laplacian cor- rection, are 0, 0.990 (from 999/1000), and 0.010 (from 10/1,000), respectively. Using the Laplacian correction for the three quantities, we pretend that we have 1 more tuple for each income-value pair. In this way, we instead obtain the following probabilities (rounded up to three decimal places):

$$\frac{1}{1,003} = 0.001, \frac{991}{1,003} = 0.988, \text{ and } \frac{11}{1,003} = 0.011,$$

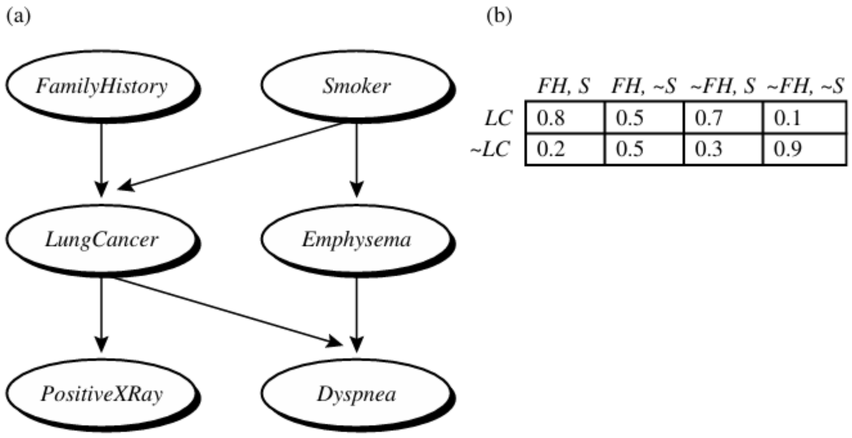
respectively. The "corrected" probability estimates are close to their "uncorrected" coun- terparts, yet the zero probability value is avoided. ■

6.3.1 Bayesian Belief Networks

- The naïve Bayesian classifier makes the assumption of class conditional independence, that is, given the class label of a tuple, the values of the attributes are assumed to be con- ditionally independent of one another. This simplifies computation. When the assump- tion holds true, then the naïve Bayesian classifier is the most accurate in comparison with all other classifiers. In practice, however, dependencies can exist between variables.
- **Bayesian belief networks** specify joint conditional probability distributions. They allow class conditional independencies to be defined between subsets of variables. They pro- vide a graphical model of causal relationships, on which learning can be performed. Trained Bayesian belief networks can be used for classification. Bayesian belief networks are also known as **belief networks**, **Bayesian networks**, and **probabilistic networks**. For brevity, we

will refer to them as belief networks.

- A belief network is defined by two components—a *directed acyclic graph* and a set of *conditional probability tables* (Figure 6.11). Each node in the directed acyclic graph represents a random variable.
- The variables may be discrete or continuous-valued. They may correspond to actual attributes given in the data or to "hidden variables" believed to form a relationship (e.g., in the case of medical data, a hidden variable may indicate a syndrome, representing a number of symptoms that, together, characterize a specific disease). Each arc represents a probabilistic dependence. If an arc is drawn from a node *Y* to a node *Z*, then *Y* is a **parent** or **immediate predecessor** of *Z*, and *Z* is a **descendant** of *Y* . *Eachvariable is conditionally independent of its nondescendants in the graph, given its parents.*
- Figure 6.11 is a simple belief network, adapted from [RBKK95] for six Boolean vari- ables. The arcs in Figure 6.11(a) allow a representation of causal knowledge. For example, having lung cancer is influenced by a person's family history of lung cancer, as well as whether or not the person is a smoker. Note that the variable *PositiveXRay* is indepen- dent of whether the patient has a family history of lung cancer or is a smoker, given that we know the patient has lung cancer. In other words, once we know the outcome of the variable *LungCancer*, then the variables *FamilyHistory* and *Smoker* do not provide



	FH, S	*FH, ~S*	*~FH, S*	*~FH, ~S*
LC	0.8	0.5	0.7	0.1
~LC	0.2	0.5	0.3	0.9

Figure 6.11 A simple Bayesian belief network: (a) A proposed causal model, represented by a directed acyclic graph. (b) The conditional probability table for the values of the variable *LungCancer (LC)* showing each possible combination of the values of its parent nodes, *FamilyHistory (FH)* and *Smoker (S)*. Figure is adapted from [RBKK95].

any additional information regarding *PositiveXRay*. The arcs also show that the variable *LungCancer* is conditionally independent of *Emphysema*, given its parents, *FamilyHistory* and *Smoker*.

- A belief network has one **conditional probability table (CPT)** for each variable. The CPT for a variable *Y* specifies the conditional distribution *P*(*Y* | *Parents*(*Y*)), where *Parents*(*Y*) are the parents of *Y*. Figure 6.11(b) shows a CPT for the variable *LungCancer*. The conditional probability for each known value of *LungCancer* is given for each possible combination of values of its parents. For instance, from the upper leftmost and bottom rightmost entries, respectively, we see that

P(*LungCancer = yes* | *FamilyHistory = yes, Smoker = yes*) = *0.8*

P(*LungCancer = no* | *FamilyHistory = no, Smoker = no*) = *0.9*

- Let $\boldsymbol{X} = (x_1, \ldots, x_n)$ be a data tuple described by the variables or attributes $Y_1, \ldots, Y_n$, respectively. Recall that each variable is conditionally independent of its non-descendants in the network graph, given its parents. This allows the network to provide a complete representation of the existing joint probability distribution with the following equation:

$$P(x_1, \ldots, x_n) = \prod_{i=1}^{n} P(x_i | Parents(Y_i)), \tag{6.16}$$

where $P(x_1, \ldots, x_n)$ is the probability of a particular combination of values of $\boldsymbol{X}$, and the values for $P(x_i|Parents(Y_i))$ correspond to the entries in the CPT for Y_i.

- A node within the network can be selected as an "output" node, representing a class label attribute. There may be more than one output node. Various algorithms for learning can be applied to the network. Rather than returning a single class label, the classification process can return a probability distribution that gives the probability of each class.

6.3.1 Training Bayesian Belief Networks

- *"How does a Bayesian belief network learn?"* In the learning or training of a belief network, a number of scenarios are possible. The network **topology** (or "layout" of nodes and arcs) may be given in advance or inferred from the data. The network variables may be *observable* or *hidden* in all or some of the training tuples. The case of hidden data is also referred to as *missing values* or *incomplete data*.

- Several algorithms exist for learning the network topology from the training data given observable variables. The problem is one of discrete optimization. For solutions, please see the bibliographic notes at the end of this chapter. Human experts usually have a good grasp of the direct conditional dependencies that hold in the domain under anal- ysis, which helps in network design. Experts must specify conditional probabilities for the nodes that participate in direct dependencies. These probabilities can then be used to compute the remaining probability values.

- If the network topology is known and the variables are observable, then training the network is straightforward. It consists of computing the CPT entries, as is similarly done when computing the probabilities involved in naive Bayesian classification.

- When the network topology is given and some of the variables are hidden, there are various methods to choose from for training the belief network. We will describe a promising method of gradient descent. For those without an advanced math back- ground, the description may look rather intimidating with its calculus-packed formulae. However, packaged software exists to solve these equations, and the general idea is easy to follow.

Let D be a training set of data tuples, $\boldsymbol{X_1}, \boldsymbol{X_2}, \ldots, \boldsymbol{X}_{|\boldsymbol{D}|}$. Training the belief network means that we must learn the values of the CPT entries. Let w_{ijk} be a CPT entry for the variable $Y_i = y_{ij}$ having the parents $U_i = u_{ik}$, where $w_{ijk} \equiv P(Y_i = y_{ij}|U_i = u_{ik})$. For example, if w_{ijk} is the upper leftmost CPT entry of Figure 6.11(b), then Y_i is *LungCancer*; y_{ij} is its value, "*yes*"; U_i lists the parent nodes of Y_i, namely, {*FamilyHistory, Smoker*}; and u_{ik} lists the values of the parent nodes, namely, {"*yes*", "*yes*"}. The w_{ijk} are viewed as weights, analogous to the weights in hidden units of neural networks (Section 6.6). The set of weights is collectively referred to as $\boldsymbol{W}$. The weights are initialized to random probability values. A *gradient descent* strategy performs greedy hill-climbing. At each iteration, the weights are updated and will eventually converge to a local optimum solution.

A **gradient descent** strategy is used to search for the w_{ijk} values that best model the data, based on the assumption that each possible setting of w_{ijk} is equally likely. Such a strategy is iterative. It searches for a solution along the negative of the gradient (i.e., steepest descent) of a criterion function. We want to find the set of weights, $\boldsymbol{W}$, that maximize this function. To start with, the weights are initialized to random probability values.

The gradient descent method performs greedy hill-climbing in that, at each iteration or step along the way, the algorithm moves toward what appears to be the best solution at the moment, without backtracking. The weights are updated at each iteration. Eventu- ally, they converge to a local optimum solution.

For our problem, we maximize $P_w(D) = \prod_{d=1}^{|D|} P_w(\boldsymbol{X}_d)$. This can be done by following the gradient of $\ln P_w(S)$, which makes the problem simpler. Given the network topology and initialized w_{ijk}, the algorithm proceeds as follows:

1. **Compute the gradients:** For each i, j, k, compute

$$\frac{\partial \ln P_w(D)}{\partial w_{ijk}} = \sum_{d=1}^{|D|} \frac{P(Y_i = y_{ij}, U_i = u_{ik}|\boldsymbol{X}_d)}{w_{ijk}}. \tag{6.17}$$

The probability in the right-hand side of Equation (6.17) is to be calculated for each training tuple, $\boldsymbol{X_d}$, in D. For brevity, let's refer to this probability simply as p. When the variables represented by Y_i and U_i are hidden for some $\boldsymbol{X_d}$, then the corresponding probability p can be computed from the observed variables of the tuple using standard algorithms for Bayesian network inference such as those available in the commercial software package HUGIN (*http://www.hugin.dk*).

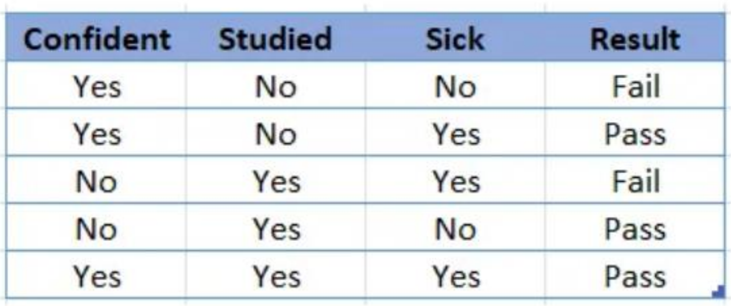
2. **Take a small step in the direction of the gradient:** The weights are updated by

$$w_{ijk} \leftarrow w_{ijk} + (l) \frac{\partial \ln P_w(D)}{\partial w_{ijk}}, \tag{6.18}$$

where l is the **learning rate** representing the step size and $\frac{\partial \ln P_w(D)}{\partial w_{ijk}}$ is computed from Equation (6.17). The learning rate is set to a small constant and helps with convergence.

3. **Renormalize the weights:** Because the weights w_{ijk} are probability values, they must be between 0.0 and 1.0, and $\sum_j w_{ijk}$ must equal 1 for all i, k. These criteria are achieved by renormalizing the weights after they have been updated by Equation (6.18).

Algorithms that follow this form of learning are called *Adaptive Probabilistic Networks*. Other methods for training belief networks are referenced in the bibliographic notes at the end of this chapter. Belief networks are computationally intensive. Because belief net- works provide explicit representations of causal structure, a human expert can provide prior knowledge to the training process in the form of network topology and/or condi- tional probability values. This can significantly improve the learning rate.



Confident	Studied	Sick	Result
Yes	No	No	Fail
Yes	No	Yes	Pass
No	Yes	Yes	Fail
No	Yes	No	Pass
Yes	Yes	Yes	Pass

Data set for classification

Find out whether the object with attribute **Confident = Yes, Sick = No** will Fail or Pass using Bayesian classification.

Solution:
The data tuples are described by the attributes *Confident*, *Studied* and *Sick*. The class label attribute, *Result*, has two distinct values (namely, {*Pass*, *Fail*}).
Let, C1 correspond to the class *Result* = *Pass* and
C2 correspond to *Result* = *Fail*.
The tuple we wish to classify is
X = (Confident = Yes, Sick = No)
Formula:
To predict the class label of **X**, P(**X**| Ci)P(Ci) is evaluated for each class Ci . The classifier predicts that the class label of tuple X is the class Ci if and only if

$$P(\mathbf{X}|C_i)P(C_i) > P(\mathbf{X}|C_j)P(C_j) \quad \text{for } 1 \le j \le m, j \ne i.$$

In other words, the predicted class label is the class Ci for which P(**X**| Ci)P(Ci) is the maximum.

$$posterior = \frac{prior \times likelihood}{evidence}$$

Step 1: *(Compute prior probability)*
We need to maximize P (**X**| Ci) P(Ci), for i = 1, 2. P(Ci), the prior probability of each class, can be computed based on the training tuples:
P(Result = Pass) = 3/5 = 0.6
P(Result = Fail) = 2/5 = 0.4

Step 2: *(Compute likelihood probability)*
To compute P(**X**| Ci), for i = 1, 2, we compute the following conditional probabilities:
P(Confident = Yes | Result = Pass) = 2/3 = 0.6667

P(Confident = Yes | Result = Fail) = 1/2 = 0.5
P(Sick = No | Result = Pass) = 1/3 = 0.3333
P(Sick = No | Result = Fail) = 1/2 = 0.5

Step 3: (*Compute posterior probability)*
P(**X**| Result = Pass)
= P(Confident = Yes | Result = Pass) × P(Sick = No | Result = Pass)
= 0.6667 * 0.3333
= 0.2222
P(X| Result = Fail)
= P(Confident = Yes | Result = Fail) × P(Sick = No | Result = Fail)
= 0.5 * 0.5
= 0.25

Step 4: *(predict the class for X)*
To find the class, Ci , that maximizes P(**X**| Ci)P(Ci), we compute
P(X| Result = Pass)P(Result = Pass) = 0.2222 × 0.6 = 0.1333
P(X| Result = Fail)P(Result = Fail) = 0.25 × 0.4= 0.1

Therefore, the naive Bayesian classifier predicts *Result = Pass* for tuple **X**.

Lazy Learners (or Learning from Your Neighbors)

- The classification methods discussed so far in this chapter—decision tree induction, Bayesian classification, rule-based classification, classification by backpropagation, sup- port vector machines, and classification based on association rule mining—are all examples of *eager learners.* **Eager learners**, when given a set of training tuples, will construct a generalization (i.e., classification) model beore receiving new (e.g., test) tuples to classify. We can think of the learned model as being ready and eager to classify previously unseen tuples.
- Imagine a contrasting lazy approach, in which the learner instead waits until the last minute before doing any model construction in order to classify a given test tuple. That is, when given a training tuple, a lazy learner simply stores it (or does only a little minor processing) and waits until it is given a test tuple.
- Only when it sees the test tuple does it perform generalization in order to classify the tuple based on its similarity to the stored training tuples. Unlike eager learning methods, lazy learners do less work when a training tuple is presented and more work when making a classification or prediction. Because lazy learners store the training tuples or "instances," they are also referred to as **instance- based learners**, even though all learning is essentially based on instances.
- When making a classification or prediction, lazy learners can be computationally expensive. They require efficient storage techniques and are well-suited to implementation on parallel hardware. They offer little explanation or insight into the structure of the data. Lazy learners, however, naturally support incremental learning. They are able to model complex decision spaces having hyper-polygonal shapes that may not be as easily describable by other learning algorithms (such as hyper-rectangular shapes modeled by decision trees). In this section, we look at two examples of lazy learners: *k-nearest- neighbor classifiers* and *case-based reasoning classifiers*.

6.9.1 *k*-Nearest-Neighbor Classifiers

- The *k*-nearest-neighbor method was first described in the early 1950s. The method is labor intensive when given large training sets, and did not gain popularity until the 1960s when increased computing power became available. It has since been widely used in the area of pattern recognition.
- Nearest-neighbor classifiers are based on learning by analogy, that is, by comparing a given test tuple with training tuples that are similar to it. The training tuples are described by *n* attributes. Each tuple represents a point in an *n*-dimensional space. In this way, all of the training tuples are stored in an *n*-dimensional pattern space. When given an unknown tuple, a ***k*-nearest-neighbor classifier** searches the pattern space for the *k* training tuples that are closest to the unknown tuple. These *k* training tuples are the *k* "nearest neighbors" of the unknown tuple.
- "Closeness" is defined in terms of a distance metric, such as Euclidean distance. The Euclidean distance between two points or tuples, say, $\boldsymbol{X}_1 = (x_{11}, x_{12}, \ldots, x_{1n})$ and $\boldsymbol{X}_2 = (x_{21}, x_{22}, \ldots, x_{2n})$, is

$$dist(\mathbf{X_1}, \mathbf{X_2}) = \sqrt{\sum_{i=1}^{n} (x_{1i} - x_{2i})^2}. \qquad (6.45)$$

- In other words, for each numeric attribute, we take the difference between the corre- sponding values of that attribute in tuple $\mathbf{X_1}$ and in tuple $\mathbf{X_2}$, square this difference, and accumulate it. The square root is taken of the total accumulated distance count. Typically, we normalize the values of each attribute before using Equation (6.45).
- This helps prevent attributes with initially large ranges (such as *income*) from outweighing attributes with initially smaller ranges (such as binary attributes). Min-max normalization, for example, can be used to transform a value *v* of a numeric attribute *A* to v^r in the range [0, 1] by

$$v' = \frac{v - min_A}{max_A - min_A}, \qquad (6.46)$$

computing

where min_A and max_A are the minimum and maximum values of attribute *A*. Chapter 2 describes other methods for data normalization as a form of data transformation.

- For *k*-nearest-neighbor classification, the unknown tuple is assigned the most common class among its *k* nearest neighbors. When *k* = 1, the unknown tuple is assigned the class of the training tuple that is closest to it in pattern space. Nearest- neighbor classifiers can also be used for prediction, that is, to return a real-valued prediction for a given unknown tuple. In this case, the classifier returns the average value of the real-valued labels associated with the *k* nearest neighbors of the unknown tuple.
- *"But how can distance be computed for attributes that not numeric, but categorical, such as color?"* The above discussion assumes that the attributes used to describe the tuples are all numeric. For categorical attributes, a simple method is to compare the corresponding value of the attribute in tuple $\mathbf{X_1}$ with that in tuple $\mathbf{X_2}$. If the two are identical (e.g., tuples $\mathbf{X_1}$ and $\mathbf{X_2}$ both have the color blue), then the difference between the two is taken as 0.
- If the two are different (e.g., tuple $\mathbf{X_1}$ is blue but tuple $\mathbf{X_2}$ is red), then the difference is considered to be 1. Other methods may incorporate more sophisticated schemes for differential grading (e.g., where a larger difference score is assigned, say, for blue and white than for blue and black)

"*What about missing values?*" In general, if the value of a given attribute *A* is missing in tuple $\mathbf{X_1}$ and/or in tuple $\mathbf{X_2}$, we assume the maximum possible difference. Suppose that each of the attributes have been mapped to the range [0, 1]. For categorical attributes, we take the difference value to be 1 if either one or both of the corresponding values of *A* are missing. If *A* is numeric and missing from both tuples $\mathbf{X_1}$ and $\mathbf{X_2}$, then the difference is also taken to be 1. If only one value is missing and the other (which we'll call v') is present and normalized, then we can take the difference to be either $|1 - v'|$ or $|0 - v'|$ (i.e., $1 - v'$ or v'), whichever is greater.

- *"How can I determine a good value for k, the number of neighbors?"* This can be deter- mined experimentally. Starting with *k* = 1, we use a test set to estimate the error rate of the classifier. This process can be repeated each time by incrementing *k* to allow for one more neighbor. The *k* value that gives the minimum error rate may be selected. In general, the larger the number of training tuples is, the larger the value of *k* will be (so that classification and prediction decisions can be based on a larger portion of the stored tuples). As the number of training tuples approaches infinity and $k = 1$, the error rate can be no worse then twice the Bayes error rate (the latter being the theoretical minimum). If k also approaches infinity, the error rate approaches the Bayes error rate.

- Nearest-neighbor classifiers use distance-based comparisons that intrinsically assign equal weight to each attribute. They therefore can suffer from poor accuracy when given noisy or irrelevant attributes. The method, however, has been modified to incorporate attribute weighting and the pruning of noisy data tuples. The choice of a distance metric can be critical. The Manhattan (city block) distance (Section 7.2.1), or other distance measurements, may also be used.

Nearest-neighbor classifiers can be extremely slow when classifying test tuples. If D is a training database of $|D|$ tuples and $k = 1$, then $O(|D|)$ comparisons are required in order to classify a given test tuple. By presorting and arranging the stored tuples

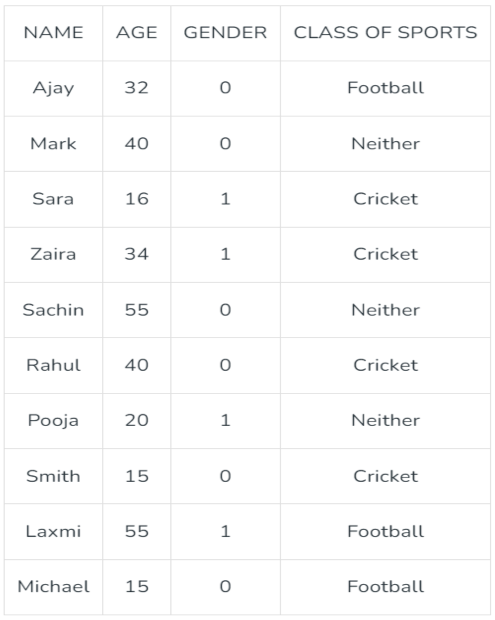
into search trees, the number of comparisons can be reduced to $O(log(|D|)$. Parallel implementation can reduce the running time to a constant, that is $O(1)$, which is inde- pendent of $|D|$. Other techniques to speed up classification time include the use of *partial distance* calculations and *editing* the stored tuples. In the **partial distance** method, we compute the distance based on a subset of the n attributes. If this distance exceeds a threshold, then further computation for the given stored tuple is halted, and the process moves on to the next stored tuple. The **editing** method removes training tuples that prove useless. This method is also referred to as **pruning** or **condensing** because it reduces the total number of tuples stored.

6.9.1 Case-Based Reasoning

- **Case-based reasoning** (CBR) classifiers use a database of problem solutions to solve new problems. Unlike nearest-neighbor classifiers, which store training tuples as points in Euclidean space, CBR stores the tuples or “cases” for problem solving as complex symbolic descriptions.
- Business applications of CBR include problem resolution for customer service help desks, where cases describe product-related diagnostic problems. CBR has also been applied to areas such as engineering and law, where cases are either technical designs or legal rulings, respectively. Medical education is another area for CBR, where patient case histories and treatments are used to help diagnose and treat new patients.
- When given a new case to classify, a case-based reasoner will first check if an iden- tical training case exists. If one is found, then the accompanying solution to that case is returned. If no identical case is found, then the case-based reasoner will search for training cases having components that are similar to those of the new case. Conceptu- ally, these training cases may be considered as neighbors of the new case. If cases are represented as graphs, this involves searching for subgraphs that are similar to sub- graphs within the new case.
- The case-based reasoner tries to combine the solutions of the neighboring training cases in order to propose a solution for the new case. If incompatibilities arise with the individual solutions, then backtracking to search for other solutions may be necessary. The case-based

reasoner may employ background knowledge and problem-solving strategies in order to propose a feasible combined solution.

- Challenges in case-based reasoning include finding a good similarity metric (e.g., for matching subgraphs) and suitable methods for combining solutions. Other challenges include the selection of salient features for indexing training cases and the development of efficient indexing techniques.
- A trade-off between accuracy and efficiency evolves as the number of stored cases becomes very large. As this number increases, the case-based reasoner becomes more intelligent. After a certain point, however, the efficiency of the system will suffer as the time required to search for and process relevant cases increases. As with nearest-neighbor classifiers, one solution is to edit the training database. Cases that are redundant or that have not proved useful may be discarded for the sake of improved performance. These decisions, however, are not clear-cut and their automa- tion remains an active area of research.



NAME	AGE	GENDER	CLASS OF SPORTS
Ajay	32	0	Football
Mark	40	0	Neither
Sara	16	1	Cricket
Zaira	34	1	Cricket
Sachin	55	0	Neither
Rahul	40	0	Cricket
Pooja	20	1	Neither
Smith	15	0	Cricket
Laxmi	55	1	Football
Michael	15	0	Football

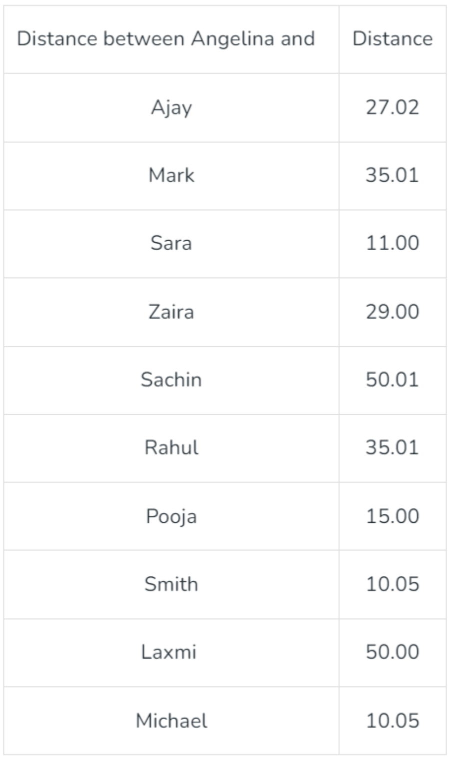
Here male is denoted with numeric value 0 and female with 1. Let's find in which class of people Angelina will lie whose k factor is 3 and age is 5. So we have to find out the distance using

$d=\sqrt{(x2-x1)^2+(y2-y1)^2}$ *to find the distance between any two points.*

So let's find out the distance between Ajay and Angelina using formula

$d=\sqrt{(age2-age1)^2+(gender2-gender1)^2}$
$d=\sqrt{(5-32)^2+(1-0)^2}$
$d=\sqrt{729}+1$
$d=27.02$

Similarly, we find out all distance one by one.



Distance between Angelina and	Distance
Ajay	27.02
Mark	35.01
Sara	11.00
Zaira	29.00
Sachin	50.01
Rahul	35.01
Pooja	15.00
Smith	10.05
Laxmi	50.00
Michael	10.05

So the value of ***k*** factor is 3 for Angelina. And the closest to 3 is 9,10,10.5 that is closest to Angelina are Zaira, Smith and Michael.

Zaira	9	cricket
Michael	10	cricket
smith	10.5	football

so according to KNN algorithm, Angelina will be in the class of people who like cricket. So this is how KNN algorithm works.