Unit-IV

Classification: Basic Concepts

Classification is a form of data analysis that extracts models describing important data classes. Such models, called classifiers, predict categorical (discrete, unordered) class labels. For example, we can build a classification model to categorize bank loan applications as either safe or risky. Such analysis can help provide us with a better understanding of the data at large.

What Is Classification?

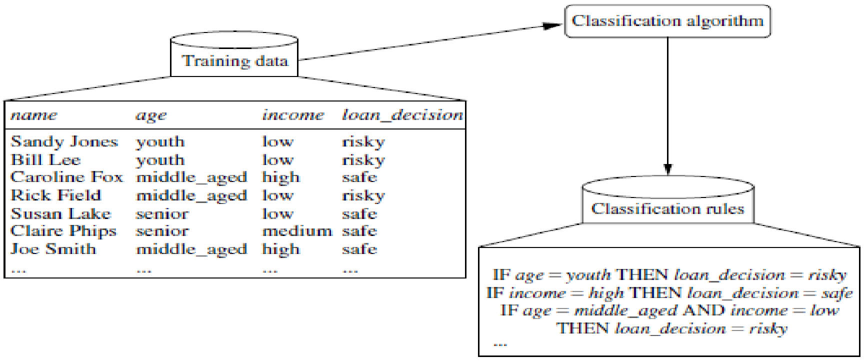
A bank loans officer needs analysis of her data to learn which loan applicants are “safe” and which are “risky” for the bank. A marketing manager at AllElectronics needs data analysis to help guess whether a customer with a given profile will buy a new computer. A medical researcher wants to analyze breast cancer data to predict which one of three specific treatments a patient should receive. In each of these examples, the data analysis task is classification, where a model or classifier is constructed to predict class (categorical) labels, such as “safe” or “risky” for the loan application data; “yes” or “no” for the marketing data; or “treatment A,” “treatment B,” or “treatment C” for the medical data. These categories can be represented by discrete values, where the ordering among values has no meaning.

Suppose that the marketing manager wants to predict how much a given customer will spend during a sale at AllElectronics. This data analysis task is an example of numeric prediction, where the model constructed predicts a continuous-valued function, or ordered value, as opposed to a class label. This model is a predictor. Regression analysis is a statistical methodology that is most often used for numeric prediction;

General Approach to Classification

“How does classification work?” Data classification is a two-step process, consisting of a learning step (where a classification model is constructed) and a classification step (where the model is used to predict class labels for given data).

In the first step, a classifier is built describing a predetermined set of data classes or concepts. This is the learning step (or training phase), where a classification algorithm builds the classifier by analyzing or “learning from” a training set made up of database tuples and their associated class labels. A tuple, X, is represented by an n-dimensional attribute vector, X =(x1, x2, … , xn), depicting n measurements made on the tuple from n database attributes, respectively, A1, A2, …., An.1 Each tuple, X, is assumed to belong to a predefined class as determined by another database attribute called the class label attribute. The class label attribute is discrete-valued and unordered. It is categorical (or nominal) in that each value serves as a category or class. The individual tuples making up the training set are referred to as training tuples.

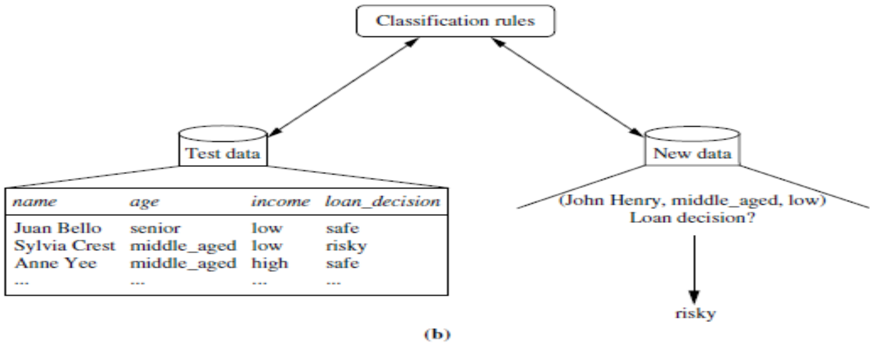


name	*age*	*income*	*loan_decision*
Sandy Jones	youth	low	risky
Bill Lee	youth	low	risky
Caroline Fox	middle_aged	high	safe
Rick Field	middle_aged	low	risky
Susan Lake	senior	low	safe
Claire Phips	senior	medium	safe
Joe Smith	middle_aged	high	safe
...	...	...	...

IF *age* = *youth* THEN *loan_decision* = *risky*
IF *income* = *high* THEN *loan_decision* = *safe*
IF *age* = *middle_aged* AND *income* = *low*
THEN *loan_decision* = *risky*
...

(a)

Source: Data Mining Concepts and Techniques, 3rd Edition, Han, Kamber and Pei



name	age	income	loan_decision
Juan Bello	senior	low	safe
Sylvia Crest	middle_aged	low	risky
Anne Yee	middle_aged	high	safe
...	...	...	...

Figure 8.1 The data classification process: (a) *Learning*: Training data are analyzed by a classification algorithm. Here, the class label attribute is *loan_decision*, and the learned model or classifier is represented in the form of classification rules. (b) *Classification*: Test data are used to estimate the accuracy of the classification rules. If the accuracy is considered acceptable, the rules can be applied to the classification of new data tuples.

Because the class label of each training tuple is provided, this step is also known as supervised learning (i.e., the learning of the classifier is "supervised" in that it is told to which class each training tuple belongs). It contrasts with unsupervised learning (or clustering), in which the class label of each training tuple is not known, and the number or set of classes to be learned may not be known in advance. For example, if we did not have the loan decision data available for the training set, we could use clustering to try to determine "groups of like tuples," which may correspond to risk groups within the loan application data.

"What about classification accuracy?" In the second step (Figure 8.1b), the model is used for classification. First, the predictive accuracy of the classifier is estimated. If we were to use the training set to measure the classifier's accuracy, this estimate would likely be optimistic, because the classifier tends to overfit the data (i.e., during learning it may incorporate some particular anomalies of the training data that are not present in the general data set overall). Therefore, a test set is used, made up of test tuples and their associated class labels. They are independent of the training tuples, meaning that they were not used to construct the classifier.

The accuracy of a classifier on a given test set is the percentage of test set tuples that are correctly classified by the classifier.

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.

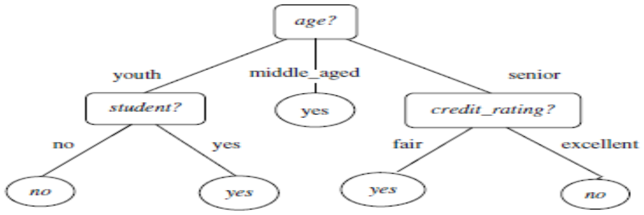


Figure 8.2 A decision tree for the concept *buys_computer*, indicating whether an *AllElectronics* customer is likely to purchase a computer. Each internal (nonleaf) node represents a test on an attribute. Each leaf node represents a class (either *buys_computer* = *yes* or *buys_computer* = *no*).

Source: Data Mining Concepts and Techniques, 3rd Edition, Han, Kamber and Pei

Unit-IV

"How are decision trees used for classification?" Given a tuple, X, for which the associated 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 multidimensional data.

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.
ID3, C4.5, and CART adopt a greedy (i.e., nonbacktracking) approach in which decision trees are constructed in a top-down recursive divide-and-conquer manner. Most algorithms for decision tree induction also follow a top-down approach, which starts with a training set of tuples and their associated class labels. A basic decision tree algorithm is summarized as follows

Strategy:

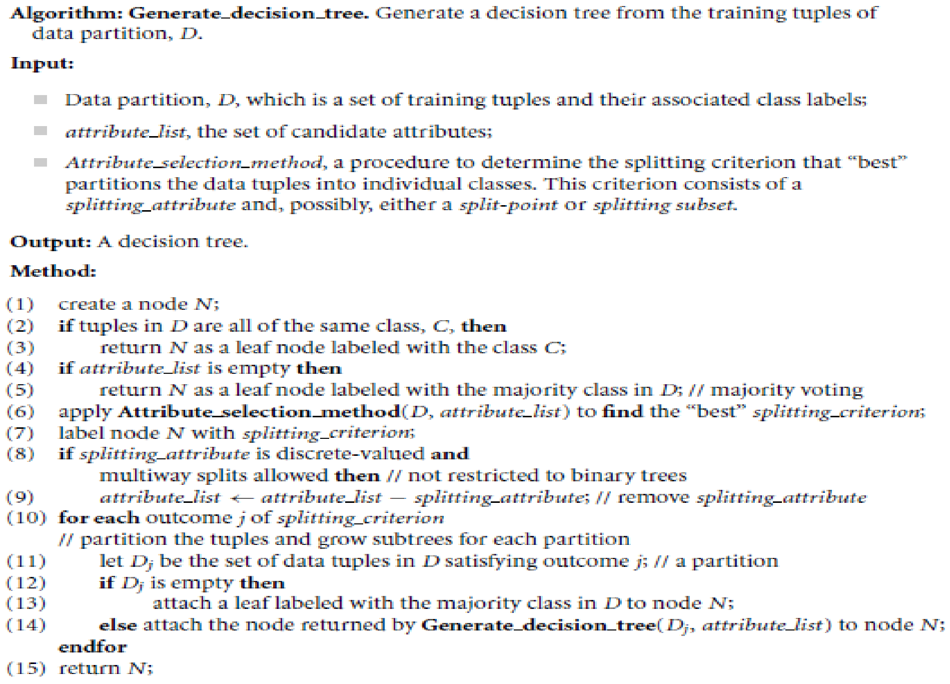
- The tree starts as a single node, N, representing the training tuples in D (step 1).
- If the tuples in D are all of the same class, then node N becomes a leaf and is labeled with that class (steps 2 and 3). Note that steps 4 and 5 are terminating conditions. Otherwise, the algorithm calls Attribute selection method to determine the splitting criterion. The splitting criterion tells us which attribute to test at node N by determining the "best" way to separate or partition the tuples in D into individual classes (step 6) - a split-point or a splitting subset
  A partition is pure if all the tuples in it belong to the same class.

- The node N is labeled with the splitting criterion, which serves as a test at the node (step 7). Let A be the splitting attribute.A has v distinct values, {a1, a2, … , av}, based on the training data.
  1. A is discrete-valued:
  2. A is continuous-valued:
  3. A is discrete-valued and a binary tree must be produced (as dictated by the attribute selection measure or algorithm being used):
  4. The algorithm uses the same process recursively to form a decision tree for the tuples at each resulting partition, Dj , of D (step 14).
  5. The recursive partitioning stops only when any one of the following terminating conditions is true:
     1. All the tuples in partition D (represented at node N) belong to the same class (steps 2 and 3).

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

Source: Data Mining Concepts and Techniques, 3rd Edition, Han, Kamber and Pei

with the most common class in D. Alternatively, the class distribution of the node tuples may be stored



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

Input:

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

Output: A decision tree.

Method:

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

Figure 8.3 Basic algorithm for inducing a decision tree from training tuples.

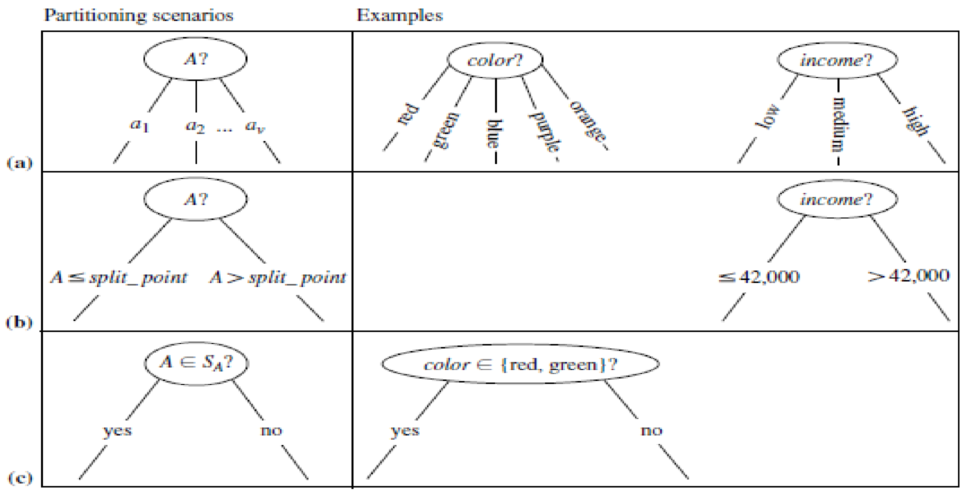


Figure 8.4 This figure shows three possibilities for partitioning tuples based on the splitting criterion, each with examples. Let *A* be the splitting attribute. (a) If *A* is discrete-valued, then one branch is grown for each known value of *A*. (b) If *A* is continuous-valued, then two branches are 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*.

3. There are no tuples for a given branch, that is, a partition Dj 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).

Source: Data Mining Concepts and Techniques, 3rd Edition, Han, Kamber and Pei

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.

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 the tuples that fall into a given partition would belong to the same class).

The attribute selection measure provides a ranking for each attribute describing the given training tuples. The attribute having the best score for the measure4 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 three popular attribute selection measures—information gain, gain ratio, and Gini index.

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.

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

$$Info(D) = -\sum_{i=1}^{m} p_i \log_2(p_i), \qquad (8.1)$$

where pi is the nonzero probability that an arbitrary tuple in D belongs to class Ci 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. 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 Dj contains those tuples in D that have outcome aj of A.

How much more information would we still need (after the partitioning) to arrive at an exact classification? This amount is measured by

$$Info_A(D) = \sum_{j=1}^{v} \frac{|D_j|}{|D|} \times Info(D_j). \qquad (8.2)$$

The term $\frac{|D_j|}{|D|}$ acts as the weight of the jth partition. InfoA.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,

$$Gain(A) = Info(D) - Info_A(D). \qquad (8.3)$$

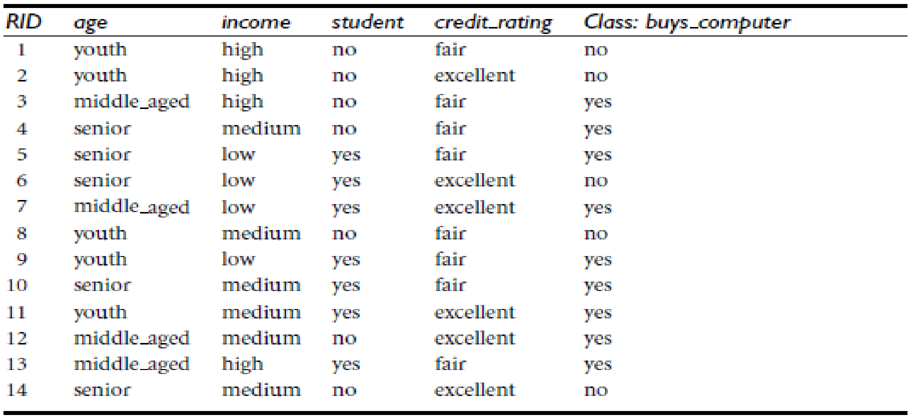
Source: Data Mining Concepts and Techniques, 3rd Edition, Han, Kamber and Pei

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 nodeN.

Example 8.1 Induction of a decision tree using information gain. Table 8.1 presents a training set, D, of class-labeled tuples randomly selected from the AllElectronics customer database. (The data are adapted from Quinlan [Qui86]. In this example, each attribute is discretevalued. Continuous-valued attributes have been generalized.) The class label attribute, buys computer, has two distinct values (namely, fyes, nog); therefore, there are two distinct classes (i.e., m = 2). Let class C1 correspond to yes and class C2 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 Eq. (8.1) to compute the expected information needed to classify a tuple in D:

Table 8.1 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

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

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 Eq. (8.2), the expected information needed to classify a tuple in D if the tuples are partitioned according to age is

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

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

Source: Data Mining Concepts and Techniques, 3rd Edition, Han, Kamber and Pei

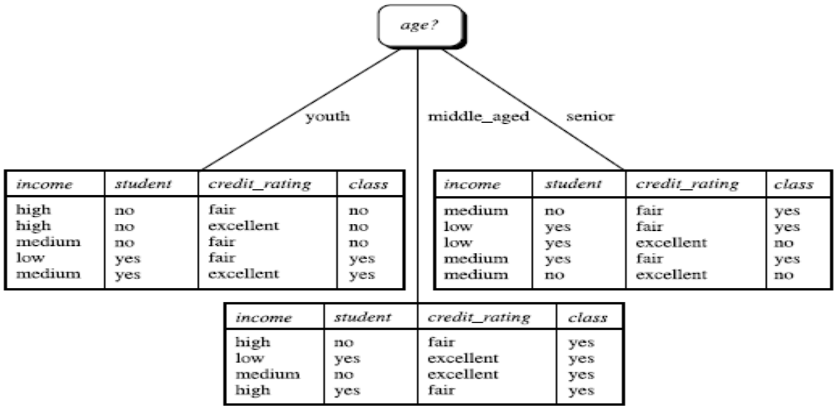
$$Gain(age) = Info(D) - Info_{age}(D) = 0.940 - 0.694 = 0.246 \text{ bits.}$$

Similarly, we can compute Gain.income/ D 0.029 bits, Gain.student/ D 0.151 bits, and Gain.credit rating/ D 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 8.5. Notice that the tuples falling into the partition for age D 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 "yes." The final decision tree returned by the algorithm was shown earlier in Figure 8.2.

"But how can we compute the information gain of an attribute that is continuousvalued,unlike in the example?" 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 from the example, we 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 ai and ai+1 of A is

$$\frac{a_i + a_{i+1}}{2}. \qquad (8.4)$$



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 8.5 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.

The point with the minimum expected information requirement for A is selected as the split point for A. D1 is the set of tuples in D satisfying A <=split point, and D2 is the set of tuples in D satisfying A > split point.

Gain Ratio

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

$$SplitInfo_A(D) = -\sum_{j=1}^{v} \frac{|D_j|}{|D|} \times \log_2\left(\frac{|D_j|}{|D|}\right). \qquad (8.5)$$

This value represents the potential information generated by splitting the training data set, D, into v

Source: Data Mining Concepts and Techniques, 3rd Edition, Han, Kamber and Pei

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. The gain ratio is defined as

$$GainRatio(A) = \frac{Gain(A)}{SplitInfo_A(D)}. \qquad (8.6)$$

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.

Example 8.2 Computation of gain ratio for the attribute income. A test on income splits the data of Table 8.1 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 Eq. (8.5) to obtain

$$SplitInfo_{income}(D) = -\frac{4}{14} \times \log_2\left(\frac{4}{14}\right) - \frac{6}{14} \times \log_2\left(\frac{6}{14}\right) - \frac{4}{14} \times \log_2\left(\frac{4}{14}\right)$$

$$= 1.557.$$

From Example 8.1, we have Gain(income) D 0.029. Therefore, GainRatio(income) =0.029/1.557 = 0.019.

Gini Index

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

$$Gini(D) = 1 - \sum_{i=1}^{m} p_i^2, \qquad (8.7)$$

where pi is the probability that a tuple in D belongs to class Ci 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, {a1, a2, … , av}, occurring in D. To determine the best binary split on A, we examine all the possible subsets that can be formed using known values of A. Each subset, SA, can be considered as a binary test for attribute A of the form "A ∈ SA?" Given a tuple, this test is satisfied if the value of A for the tuple is among the values listed in SA. If A has v possible values, then there are 2v possible subsets.

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 D1 and D2, the Gini index of D given that partitioning is

$$Gini_A(D) = \frac{|D_1|}{|D|}Gini(D_1) + \frac{|D_2|}{|D|}Gini(D_2). \qquad (8.8)$$

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 earlier for information gain, where the midpoint between each pair of (sorted) adjacent values is taken as a possible split-point.

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

Source: Data Mining Concepts and Techniques, 3rd Edition, Han, Kamber and Pei

$$\Delta Gini(A) = Gini(D) - Gini_A(D). \qquad (8.9)$$

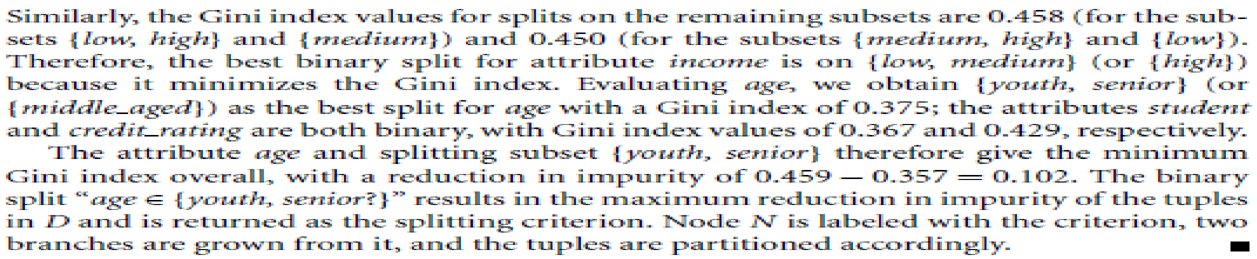
The attribute that maximizes the reduction in impurity (or, equivalently, has the minimum Gini index) is selected as the splitting attribute.

Example 8.3 Induction of a decision tree using the Gini index. Let D be the training data shown earlier in Table 8.1, 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 Eq. (8.7) for the 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 D1 satisfying the condition "$income \in \{low, medium\}$." The remaining four tuples of D would be assigned to partition D2. The Gini index value computed based on this partitioning is

$$\begin{aligned} &Gini_{income \in \{low,medium\}}(D) \\ &\quad = \frac{10}{14}Gini(D_1) + \frac{4}{14}Gini(D_2) \\ &\quad = \frac{10}{14}\left(1-\left(\frac{7}{10}\right)^2-\left(\frac{3}{10}\right)^2\right)+\frac{4}{14}\left(1-\left(\frac{2}{4}\right)^2-\left(\frac{2}{4}\right)^2\right) \\ &\quad = 0.443 \\ &\quad = Gini_{income \in \{high\}}(D). \end{aligned}$$



Similarly, the Gini index values for splits on the remaining subsets are 0.458 (for the subsets {*low, high*} and {*medium*}) and 0.450 (for the subsets {*medium, high*} and {*low*}). Therefore, the best binary split for attribute *income* is on {*low, medium*} (or {*high*}) because it minimizes the Gini index. Evaluating *age*, 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 *age* and splitting subset {*youth, senior*} therefore give the minimum Gini index overall, with a reduction in impurity of $0.459 - 0.357 = 0.102$. The binary split "$age \in \{youth, senior?\}$" 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. ■

Other Attribute Selection Measures

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 χ^2 test for independence. Other measures include C-SEP (which performs better than information gain and the Gini index in certain cases) and G-statistic (an information theoretic measure that is a close approximation to χ^2 distribution).

Attribute selection measures based on the **Minimum Description Length (MDL)** principle 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).

Source: Data Mining Concepts and Techniques, 3rd Edition, Han, Kamber and Pei

Other attribute selection measures consider **multivariate splits** (i.e., where the partitioning of tuples is based on a *combination* of attributes, rather than on a single attribute).

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 *overfitting* the data.

"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).

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.

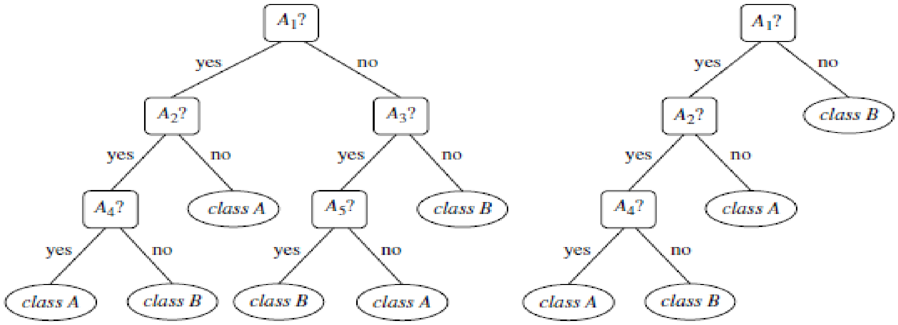
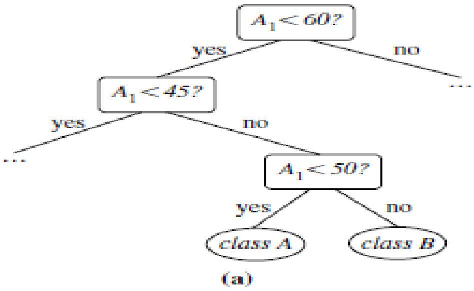


Figure 8.6 An unpruned decision tree and a pruned version of it.

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.

Decision trees can suffer from *repetition* and *replication* (Figure 8.7), making themoverwhelming to interpret. **Repetition** occurs when an attribute is repeatedly tested along a given branch of the tree. In **replication**, duplicate subtrees exist within the tree.



Source: Data Mining Concepts and Techniques, 3rd Edition, Han, Kamber and Pei

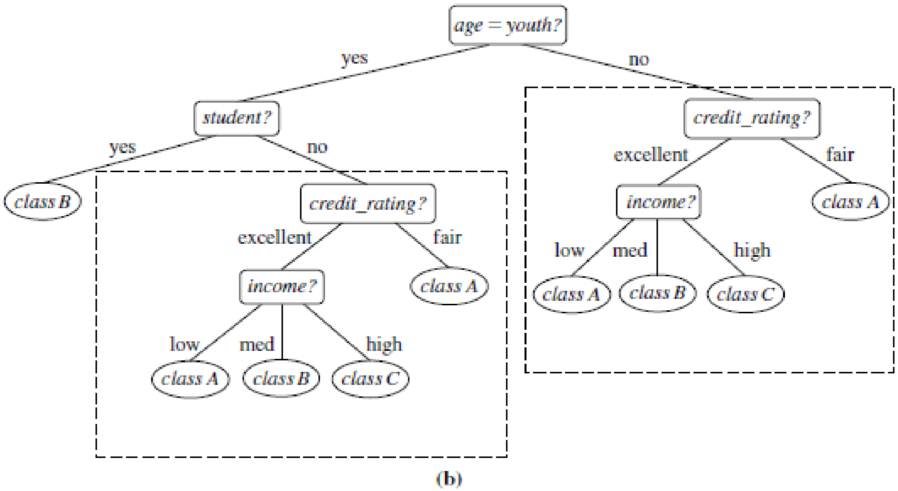
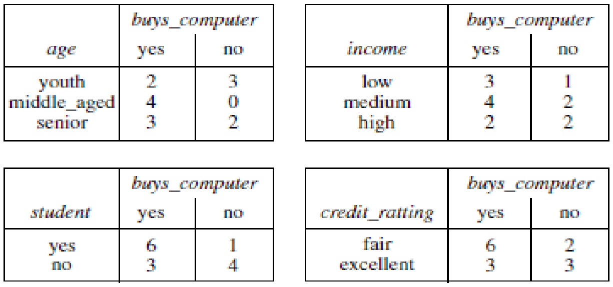


Figure 8.7 An example of: (a) subtree **repetition**, where an attribute is repeatedly tested along a given branch of the tree (e.g., *age*) and (b) subtree **replication**, where duplicate subtrees exist within a tree (e.g., the subtree headed by the node "*credit_rating?*").

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 decision 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.

RainForest, for example, 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*. 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 values 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 also has techniques, however, for handling the case where the AVC-group does not fit in memory.
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 "bootstrapping" 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*0, that turns out to be "very close" to the tree that would have been generated if all the original training data had fit in memory.



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

Figure 8.8 The use of data structures to hold aggregate information regarding the training data (e.g., these AVC-sets describing Table 8.1's data) are one approach to improving the scalability of decision tree induction.

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 to detect if this "very good" tree, *T*0, is different from the "real" tree, *T*, that would have been generated using all of the data. It refines *T*0 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.

Visual Mining for Decision Tree Induction

"Are there any interactive approaches to decision tree induction that allow us to visualize the data and the tree as it is being constructed? Can we use any knowledge of our data to help in building the tree?" In this section, you will learn about an approach to decision tree induction that supports these options. **Perception-based classification (PBC)** is an interactive approach based on multidimensional visualization techniques and allows the user to incorporate background knowledge about the data when building a decision tree. By visually interacting with the data, the user is also likely to develop a deeper understanding of the data. The resulting trees tend to be smaller than those built using traditional decision tree induction methods and so are easier to interpret, while achieving about the same accuracy. The trees constructed with PBC were compared with trees generated by the CART, C4.5, and SPRINT algorithms from various data sets.

Bayes Classification Methods

"What are Bayesian classifiers?" Bayesian classifiers are statistical classifiers. They can predict class membership probabilities such as the probability that a given tuple belongs to a particular class.

Na¨ıve 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- conditional independence*.

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 $\boldsymbol{X}$.

Source: Data Mining Concepts and Techniques, 3rd Edition, Han, Kamber and Pei

P(*H*|***X****)* is the **posterior probability**, or *a posteriori probability*, of *H* conditioned on ***X***. For example, suppose our world of data tuples is confined to customers described by 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 our example, this is the probability that any given customer will buy a computer, regardless of age, income, or any other information, for that matter.

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.

$$P(H|X) = \frac{P(X|H)P(H)}{P(X)}. \qquad (8.10)$$

Na¨ıve Bayesian Classification

The **na¨ıve 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, ***X*** =(*x*1, *x*2,... , *xn)*, depicting *n* measurements made on the tuple from *n* attributes, respectively, *A*1, *A*2,.... , *An*.

2. Suppose that there are *m* classes, *C*1, *C*2,.... , *Cm*. Given a tuple, ***X***, the classifier will predict that ***X*** belongs to the class having the highest posterior probability, conditioned on ***X***. That is, the na¨ıve Bayesian classifier predicts that tuple ***X*** belongs to the class *Ci* if and only if

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

Thus, we maximize *P(Ci*|***X****)*. The class *Ci* for which *P(Ci*|***X****)* is maximized is called the *maximum posteriori hypothesis*. By Bayes' theorem (Eq. 8.10),

$$P(C_i|X) = \frac{P(X|C_i)P(C_i)}{P(X)}. \qquad (8.11)$$

3. As *P(****X****)* is constant for all classes, only *P(****X****|Ci)P(Ci*) needs to 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) = … = *P(Cm)*, and we would therefore maximize *P(****X****|Ci)*. Otherwise, we maximize *P(****X****|Ci)P(Ci)*. Note that the class prior probabilities may be estimated by *P(Ci)* =|*Ci*,*D*|/|*D*|, where |*Ci*,*D*| is the number of training tuples of class *Ci* in *D*.

4. Given data sets with many attributes, it would be extremely computationally expensive to compute *P(****X****|Ci)*. To reduce computation in evaluating *P(****X****|Ci)*, the na¨ıve assumption of **class-conditional independence** is made. This presumes that the attributes' values 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,

Source: Data Mining Concepts and Techniques, 3rd Edition, Han, Kamber and Pei

$$P(X|C_i) = \prod_{k=1}^{n} P(x_k|C_i) \quad (8.12)$$
$$= P(x_1|C_i) \times P(x_2|C_i) \times \cdots \times P(x_n|C_i).$$

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

(a) If *Ak* is categorical, then *P(*xk*|Ci)*, is the number of tuples of class *Ci* in *D* having the value *xk* for *Ak*, divided by |*Ci*,*D*|, the number of tuples of class *Ci* in *D*.
(b) If *Ak* 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 σ, defined by

$$g(x, \mu, \sigma) = \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{(x-\mu)^2}{2\sigma^2}}, \quad (8.13)$$

so that

$$P(x_k|C_i) = g(x_k, \mu_{C_i}, \sigma_{C_i}). \quad (8.14)$$

For example, let ***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 ***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 μ = 38 years and σ = 12. We can plug these quantities, along with *x*1 = 35 for our tuple ***X***, into Eq. (8.13) to estimate *P(age = 35|buys_computer = yes)*.
5. 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

$$P(X|C_i)P(C_i) > P(X|C_j)P(C_j) \quad \text{for } 1 \leq j \leq m, j \neq i. \quad (8.15)$$

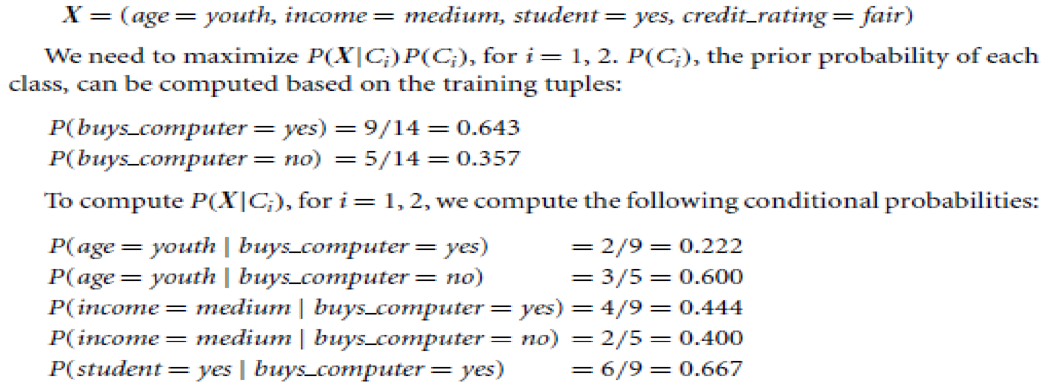
In other words, the predicted class label is the class *Ci* for which *P(**X**|Ci)P(Ci*) is the maximum.

"How effective are Bayesian classifiers?"
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.

Example 8.4 Predicting a class label using na¨ıve Bayesian classification. We wish to predict the class label of a tuple using na¨ıve Bayesian classification, given the same training data as in Example 8.3 for decision tree induction. The training data were shown earlier in Table 8.1. The data tuples are

Source: Data Mining Concepts and Techniques, 3rd Edition, Han, Kamber and Pei

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



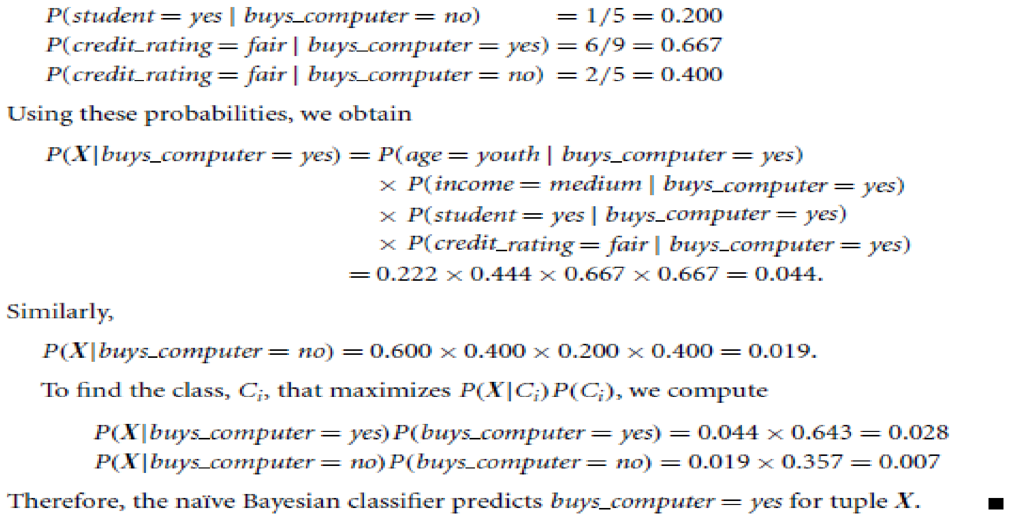
$$X = (age = youth,\ income = medium,\ student = yes,\ credit\_rating = fair)$$

We need to maximize $P(X|C_i)P(C_i)$, 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 $P(X|C_i)$, for $i = 1, 2$, we compute the following conditional probabilities:

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



$$\begin{aligned}
&P(student = yes \mid buys\_computer = no) &&= 1/5 = 0.200\\
&P(credit\_rating = fair \mid buys\_computer = yes) &&= 6/9 = 0.667\\
&P(credit\_rating = fair \mid buys\_computer = no) &&= 2/5 = 0.400
\end{aligned}$$

Using these probabilities, we obtain

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

Similarly,

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

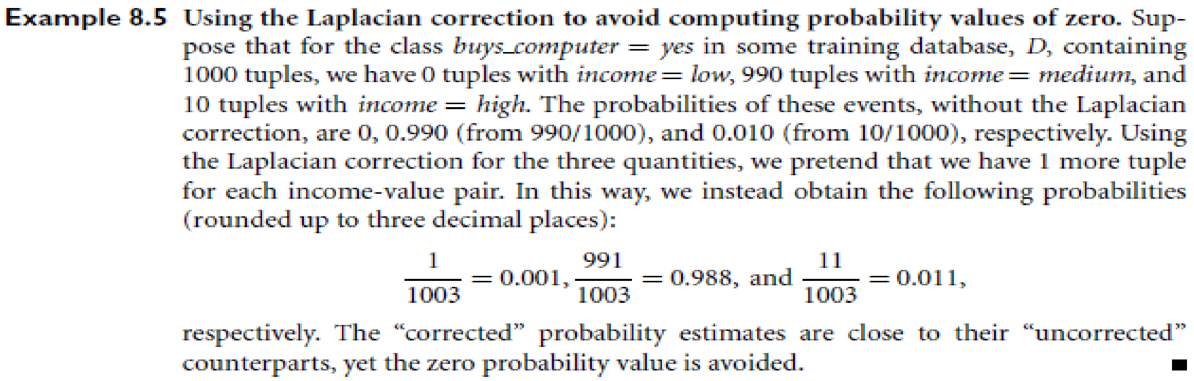
To find the class, C_i, that maximizes $P(X|C_i)P(C_i)$, we compute

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

Therefore, the naïve Bayesian classifier predicts $buys\_computer = yes$ for tuple X. ■

what happens if we should end up with a probability value of zero for some *P(xk|Ci)*?

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**.



Example 8.5 **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 1000 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 correction, are 0, 0.990 (from 990/1000), and 0.010 (from 10/1000), 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):

$$\frac{1}{1003} = 0.001, \frac{991}{1003} = 0.988, \text{ and } \frac{11}{1003} = 0.011,$$

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

Source: Data Mining Concepts and Techniques, 3rd Edition, Han, Kamber and Pei

Unit-IV

Rule-Based Classification

Using IF-THEN Rules for Classification

A rule-based classifier uses a set of IF-THEN rules for classification. An **IF-THEN** rule is an expression of the form

IF *condition* THEN *conclusion*.

An example is rule *R*1,

*R*1: IF *age* =*youth* AND *student* =*yes* THEN *buys computer* = *yes*.

The "IF" part (or left side) of a rule is known as the **rule antecedent** or **precondition**. The "THEN" part (or right side) is the **rule consequent**. In the rule antecedent, the condition consists of one or more *attribute tests* (e.g., *age* = *youth* and *student* = *yes*) that are logically ANDed. The rule's consequent contains a class prediction (in this case, we are predicting whether a customer will buy a computer). *R*1 can also be written as

$R1: (age = youth) \wedge (student = yes) \Rightarrow (buys\_computer = yes).$

If the condition (i.e., all the attribute tests) in a rule antecedent holds true for a given tuple, we say that the rule antecedent is **satisfied** (or simply, that the rule is satisfied) and that the rule **covers** the tuple.

A rule *R* can be assessed by its coverage and accuracy. Given a tuple, ***X***, from a classlabeled data set, *D*, let *ncovers* be the number of tuples covered by *R*; *ncorrect* be the number of tuples correctly classified by *R*; and |*D*|*1* be the number of tuples in *D*. We can define the **coverage** and **accuracy** of *R* as

$$coverage(R) = \frac{n_{covers}}{|D|} \qquad (8.16)$$

$$accuracy(R) = \frac{n_{correct}}{n_{covers}}. \qquad (8.17)$$

That is, a rule's coverage is the percentage of tuples that are covered by the rule.

Example 8.6 Rule accuracy and coverage. Let's go back to our data in Table 8.1. These are classlabeled

tuples from the *AllElectronics* customer database. Our task is to predict whether a customer will buy a computer. Consider rule *R*1, which covers 2 of the 14 tuples. It can correctly classify both tuples. Therefore, *coverage(R*1)= 2/14 = 14.28% and *accuracy(R*1) = 2/2 = 100%.

If a rule is satisfied by ***X***, the rule is said to be **triggered**. For example, suppose we have

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

We would like to classify ***X*** according to *buys computer*. ***X*** satisfies *R*1, which triggers the rule.

If *R*1 is the only rule satisfied, then the rule **fires** by returning the class prediction for ***X***. Note that triggering does not always mean firing because there may be more than one rule that is satisfied! If more than one rule is triggered, we have a potential problem. What if they each specify a different class? Or what if no rule is satisfied by ***X***?

We tackle the first question. If more than one rule is triggered, we need a **conflict resolution strategy** to figure out which rule gets to fire and assign its class prediction to ***X***. There are many possible strategies. We look at two, namely *size ordering* and *rule ordering*.

The **size ordering** scheme assigns the highest priority to the triggering rule that has the "toughest" requirements, where toughness is measured by the rule antecedent *size*. That is, the triggering rule with the most attribute tests is fired.

The **rule ordering** scheme prioritizes the rules beforehand. The ordering may be *class-based* or *rule-based*. With **class-based ordering**, the classes are sorted in order of decreasing "importance" such as by decreasing *order of prevalence*. That is, all the rules for the most prevalent (or most frequent) class come first, the rules for the next prevalent class come next, and so on.

With **rule-based ordering**, the rules are organized into one long priority list, according to some measure of rule quality, such as accuracy, coverage, or size (number of attribute tests in the rule antecedent), or based on advice from domain experts. When rule ordering is used, the rule set is known as a **decision list**.

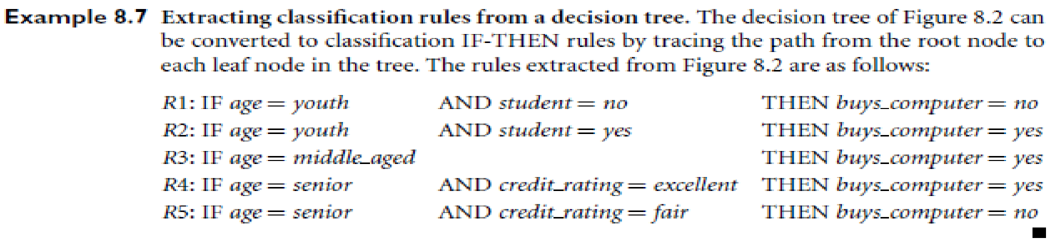
Note that in the first strategy, overall the rules are *unordered*. They can be applied in any order when classifying a tuple. That is, a disjunction (logical OR) is implied between each of the rules. Each rule represents a standalone nugget or piece of knowledge. This is in contrast to the rule ordering (decision list) scheme for which rules must be applied in the prescribed order so as to avoid conflicts. Each rule in a decision list implies the negation of the rules that come before it in the list. Hence, rules in a decision list are more difficult to interpret. Now that we have seen how we can handle conflicts, let's go back to the scenario where there is no rule satisfied by ***X***. How, then, can we determine the class label of ***X***? In this case, a fallback or **default rule** can be set up to specify a default class, based on a training set. This may be the class in majority or the majority class of the tuples that were not covered by any rule. The default rule is evaluated at the end, if and only if no other rule covers ***X***.

Rule Extraction from a Decision Tree

it is easy to understand how decision trees work and they are known for their accuracy. Decision trees can become large and difficult to interpret. In this subsection, we look at how to build a rulebased classifier by extracting IF-THEN rules from a decision tree. In comparison with a decision tree, the IF-THEN rules may be easier for humans to understand, particularly if the decision tree is very large.

To extract rules from a decision tree, one rule is created for each path from the root to a leaf node. Each splitting criterion along a given path is logically ANDed to form the rule antecedent ("IF" part). The leaf node holds the class prediction, forming the rule consequent ("THEN" part).

Source: Data Mining Concepts and Techniques, 3rd Edition, Han, Kamber and Pei



Example 8.7 **Extracting classification rules from a decision tree.** The decision tree of Figure 8.2 can be converted to classification IF-THEN rules by tracing the path from the root node to each leaf node in the tree. The rules extracted from Figure 8.2 are as follows:

*R*1: IF *age* = *youth* AND *student* = *no* THEN *buys_computer* = *no*
*R*2: IF *age* = *youth* AND *student* = *yes* THEN *buys_computer* = *yes*
*R*3: IF *age* = *middle_aged* THEN *buys_computer* = *yes*
*R*4: IF *age* = *senior* AND *credit_rating* = *excellent* THEN *buys_computer* = *yes*
*R*5: IF *age* = *senior* AND *credit_rating* = *fair* THEN *buys_computer* = *no* ■

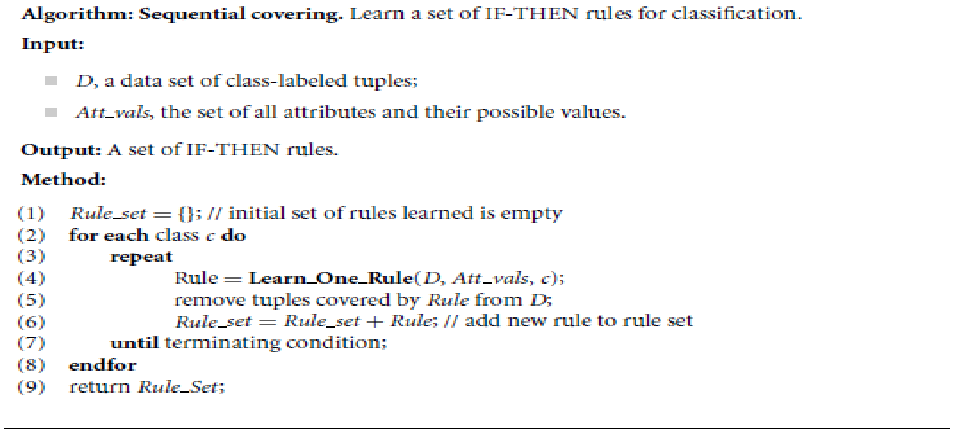
A disjunction (logical OR) is implied between each of the extracted rules. Because the rules are extracted directly from the tree, they are **mutually exclusive** and **exhaustive**. *Mutually exclusive* means that we cannot have rule conflicts here because no two rules will be triggered for the same tuple. (We have one rule per leaf, and any tuple can map to only one leaf.) *Exhaustive* means there is one rule for each possible attribute–value combination, so that this set of rules does not require a default rule. Therefore, the order of the rules does not matter—they are *unordered*.

"How can we prune the rule set?" For a given rule antecedent, any condition that does not improve the estimated accuracy of the rule can be pruned (i.e., removed), thereby generalizing the rule. C4.5 extracts rules from an unpruned tree, and then prunes the rules using a pessimistic approach similar to its tree pruning method. The training tuples and their associated class labels are used to estimate rule accuracy. However, because this would result in an optimistic estimate, alternatively, the estimate is adjusted to compensate for the bias, resulting in a pessimistic estimate.

Other problems arise during rule pruning, however, as the rules *will no longer be* mutually exclusive and exhaustive. For conflict resolution, C4.5 adopts a **class-based ordering scheme**. It groups together all rules for a single class, and then determines a ranking of these class rule sets. Within a rule set, the rules are not ordered.

Rule Induction Using a Sequential Covering Algorithm

IF-THEN rules can be extracted directly from the training data (i.e., without having to generate a decision tree first) using a **sequential covering algorithm**. The name comes from the notion that the rules are learned *sequentially* (one at a time), where each rule for a given class will ideally *cover* many of the class's tuples (and hopefully none of the tuples of other classes).



```
Algorithm: Sequential covering. Learn a set of IF-THEN rules for classification.
Input:
    ■ D, a data set of class-labeled tuples;
    ■ Att_vals, the set of all attributes and their possible values.
Output: A set of IF-THEN rules.
Method:
    Rule_set = {}; // initial set of rules learned is empty
    for each class c do
        repeat
            Rule = Learn_One_Rule(D, Att_vals, c);
            remove tuples covered by Rule from D;
            Rule_set = Rule_set + Rule; // add new rule to rule set
        until terminating condition;
    endfor
    return Rule_Set;
```

Figure 8.10 Basic sequential covering algorithm.

Source: Data Mining Concepts and Techniques, 3rd Edition, Han, Kamber and Pei

The general strategy is as follows. Rules are learned one at a time. Each time a rule is learned, the tuples covered by the rule are removed, and the process repeats on the remaining tuples. This sequential learning of rules is in contrast to decision tree induction. Because the path to each leaf in a decision tree corresponds to a rule, we can consider decision tree induction as learning a set of rules *simultaneously*.

"How are rules learned?" Typically, rules are grown in a *general-to-specific* manner (Figure 8.11). We can think of this as a beam search, where we start off with an empty rule and then gradually keep appending attribute tests to it. We append by adding the attribute test as a logical conjunct to the existing condition of the rule antecedent. Suppose our training set, *D*, consists of loan application data. Attributes regarding each

applicant include their age, income, education level, residence, credit rating, and the term of the loan. The classifying attribute is *loan decision*, which indicates whether a loan is accepted (considered safe) or rejected (considered risky). To learn a rule for the class "accept," we start off with the most general rule possible, that is, the condition of the rule antecedent is empty. The rule is

IF THEN *loan_decision* = *accept*.

We then consider each possible attribute test that may be added to the rule. These can be derived from the parameter *Att_vals*, which contains a list of attributes with their associated values. For example, for an attribute–value pair (*att*, *val)*, we can consider attribute tests such as *att* = *val*, *att* <= *val*, *att* > *val*, and so on. Typically, the training data will contain many attributes, each of which may have several possible values. Finding an optimal rule set becomes computationally explosive. Instead, *Learn One Rule* adopts a greedy depth-first strategy. Each time it is faced with adding a new attribute test (conjunct) to the current rule, it picks the one that most improves the rule quality, based on the training samples.

so that the current rule becomes

IF *income* = *high* THEN *loan_decision* = *accept*.

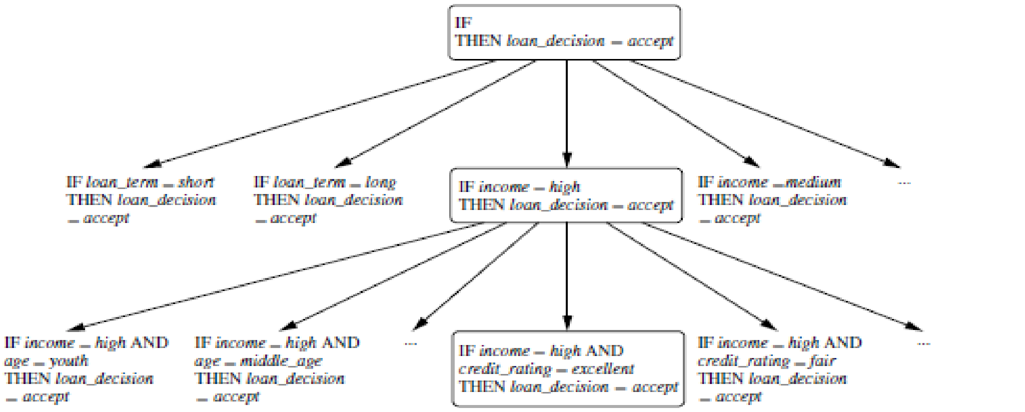

IF
THEN loan_decision = accept
IF loan_term = short THEN loan_decision = accept
IF loan_term = long THEN loan_decision = accept
IF income = high THEN loan_decision = accept
IF income = medium THEN loan_decision = accept
...
IF income = high AND age = youth THEN loan_decision = accept
IF income = high AND age = middle_age THEN loan_decision = accept
...
IF income = high AND credit_rating = excellent THEN loan_decision = accept
IF income = high AND credit_rating = fair THEN loan_decision = accept
...

Figure 8.11 A general-to-specific search through rule space.

Source: Data Mining Concepts and Techniques, 3rd Edition, Han, Kamber and Pei

Each time we add an attribute test to a rule, the resulting rule should cover relatively more of the “accept” tuples. During the next iteration, we again consider the possible attribute tests and end up selecting *credit_rating* = *excellent*. Our current rule grows to become

IF *income* = *high* AND *credit rating* = *excellent* THEN *loan decision* = *accept*.

The process repeats, where at each step we continue to greedily grow rules until the resulting rule meets an acceptable quality level.

Rule Quality Measures

Learn One Rule needs a measure of rule quality. Every time it considers an attribute test, it must check to see if appending such a test to the current rule’s condition will result in an improved rule.

Example 8.8 Choosing between two rules based on accuracy. Consider the two rules as illustrated in Figure 8.12. Both are for the class *loan decision* D *accept*. We use “*a*” to represent the tuples of class “*accept*” and “*r*” for the tuples of class “*reject*.” Rule *R*1 correctly classifies 38 of the 40 tuples it covers. Rule *R*2 covers only two tuples, which it correctly classifies. Their respective accuracies are 95% and 100%. Thus, *R*2 has greater accuracy than *R*1, but it is not the better rule because of its small coverage.

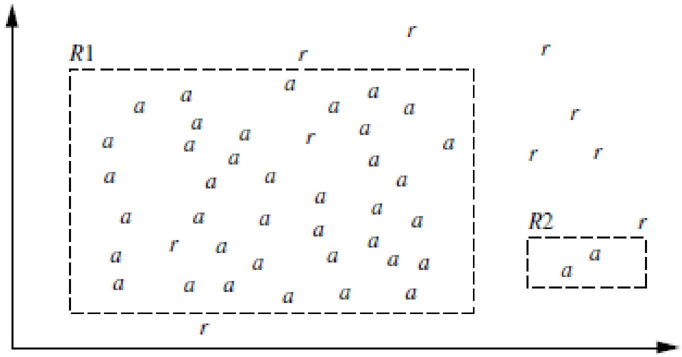


Figure 8.12 Rules for the class *loan_decision* = *accept*, showing *accept* (*a*) and *reject* (*r*) tuples.

Our current rule is *R*: IF *condition* THEN *class* = *c*. We want to see if logically ANDing a given attribute test to *condition* would result in a better rule.We call the new condition, *condition*0, where *R*0: IF *condition*0 THEN *class* = *c* is our potential new rule. In other words, we want to see if *R*0 is any better than *R*.

Another measure is based on information gain and was proposed in **FOIL** (First Order Inductive Learner), a sequential covering algorithm that learns first-order logic rules. Learning first-order rules is more complex because such rules contain variables, whereas the rules we are concerned with in this section are propositional (i.e., variablefree).

FOIL assesses the information gained by extending *condition*0 as

$$FOIL\_Gain = pos' \times \left(\log_2 \frac{pos'}{pos' + neg'} - log_2 \frac{pos}{pos + neg}\right). \qquad (8.18)$$

Source: Data Mining Concepts and Techniques, 3rd Edition, Han, Kamber and Pei

It favors rules that have high accuracy and cover many positive tuples.

Rule Pruning

Learn One Rule does not employ a test set when evaluating rules. Assessments of rule quality as described previously are made with tuples from the original training data. These assessments are optimistic because the rules will likely overfit the data. That is, the rules may perform well on the training data, but less well on subsequent data. To compensate for this, we can prune the rules. rule is pruned by removing a conjunct

(attribute test). We choose to prune a rule, *R*, if the pruned version of *R* has greater quality, as assessed on an independent set of tuples.

FOIL uses a simple yet effective method. Given a rule, *R*,

$$FOIL\_Prune(R) = \frac{pos - neg}{pos + neg}, \qquad (8.20)$$

where *pos* and *neg* are the number of positive and negative tuples covered by *R*, respectively. This value will increase with the accuracy of *R* on a pruning set. Therefore, if the *FOIL_Prune* value is higher for the pruned version of *R*, then we prune *R*.

Model Evaluation and Selection

But what is accuracy? How can we estimate it? Are some measures of a classifier's accuracy more appropriate than others?How can we obtain a *reliable* accuracy estimate? Answers to these questions can find here What if we have more than one classifier and want to choose the "best" one? This is referred to as **model selection** (i.e., choosing one classifier over another).

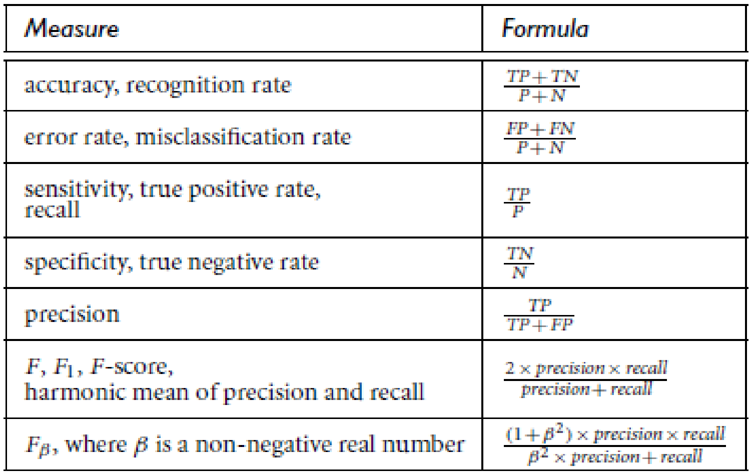
Metrics for Evaluating Classifier Performance

Here, measures for assessing how good or how "accurate" your classifier is at predicting the class label of tuples. They include accuracy (also known as recognition rate), sensitivity (or recall), specificity, precision,

*F*1, and F_β . Note that although accuracy is a specific measure, the word "accuracy" is also used as a general term to refer to a classifier's predictive abilities.

Using training data to derive a classifier and then estimate the accuracy of the resulting learned model can result in misleading overoptimistic estimates due to overspecialization of the learning algorithm to the data. Recall that we can talk in terms of **positive tuples** (tuples of the main class of interest) and **negative tuples** (all other tuples). Given two classes, for example, the positive tuples may be *buys computer* = *yes* while the negative tuples are *buys computer* = *no*. Suppose we use our classifier on a test set of labeled tuples. *P* is the number of positive tuples and *N* is the number of negative tuples. For each tuple, we compare the classifier's class label prediction with the tuple's known class label.

Source: Data Mining Concepts and Techniques, 3rd Edition, Han, Kamber and Pei



Measure	*Formula*
accuracy, recognition rate	$\frac{TP+TN}{P+N}$
error rate, misclassification rate	$\frac{FP+FN}{P+N}$
sensitivity, true positive rate, recall	$\frac{TP}{P}$
specificity, true negative rate	$\frac{TN}{N}$
precision	$\frac{TP}{TP+FP}$
F, F_1, F-score, harmonic mean of precision and recall	$\frac{2 \times precision \times recall}{precision + recall}$
F_β, where β is a non-negative real number	$\frac{(1+\beta^2) \times precision \times recall}{\beta^2 \times precision + recall}$

Figure 8.13 Evaluation measures. Note that some measures are known by more than one name. TP, TN, FP, P, N refer to the number of true positive, true negative, false positive, positive, and negative samples, respectively (see text).

True positives (*TP*): These refer to the positive tuples that were correctly labeled by the classifier. Let *TP* be the number of true positives.

True negatives (*TN*): These are the negative tuples that were correctly labeled by the classifier. Let *TN* be the number of true negatives.

False positives (*FP*): These are the negative tuples that were incorrectly labeled as positive (e.g., tuples of class *buys computer* = *no* for which the classifier predicted *buys computer* = *yes*). Let *FP* be the number of false positives.

False negatives (*FN*): These are the positive tuples that were mislabeled as negative (e.g., tuples of class *buys computer* = *yes* for which the classifier predicted *buys computer* = *no*). Let *FN* be the number of false negatives.

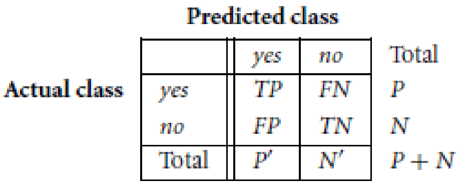
These terms are summarized in the **confusion matrix** of Figure 8.14.

The confusion matrix is a useful tool for analyzing how well your classifier can recognize tuples of different classes. *TP* and *TN* tell us when the classifier is getting things right, while *FP* and *FN* tell us when the classifier is getting things wrong (i.e., mislabeling). Given *m* classes (where *m* >= 2), a **confusion matrix** is a table of at least size *m* by *m*. An entry, *CMi,j* in the first *m* rows and *m* columns indicates the number of tuples of class *i* that were labeled by the classifier as class *j*. For a classifier to have good accuracy, ideally most of the tuples would be represented along the diagonal of the confusion matrix, from entry *CM*1,1 to entry *CMm,m*, with the rest of the entries being zero or close to zero. That is, ideally, *FP* and *FN* are around zero.

The **accuracy** of a classifier on a given test set is the percentage of test set tuples that are correctly classified by the classifier. That is,

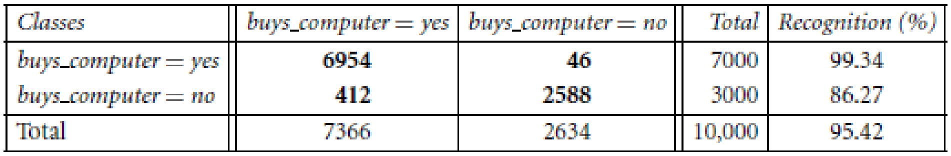
$$accuracy = \frac{TP+TN}{P+N}. \qquad (8.21)$$

Source: Data Mining Concepts and Techniques, 3rd Edition, Han, Kamber and Pei



		Predicted class		
		yes	*no*	Total
Actual class	*yes*	*TP*	*FN*	*P*
	no	*FP*	*TN*	*N*
	Total	P'	N'	$P+N$

Figure 8.14 Confusion matrix, shown with totals for positive and negative tuples.



Classes	*buys_computer = yes*	*buys_computer = no*	*Total*	*Recognition (%)*
buys_computer = yes	**6954**	**46**	7000	99.34
buys_computer = no	**412**	**2588**	3000	86.27
Total	7366	2634	10,000	95.42

Figure 8.15 Confusion matrix for the classes *buys_computer = yes* and *buys_computer = no*, where an entry in row *i* and column *j* shows the number of tuples of class *i* that were labeled by the classifier as class *j*. Ideally, the nondiagonal entries should be zero or close to zero.

In the pattern recognition literature, this is also referred to as the overall **recognition rate** of the classifier, that is, it reflects how well the classifier recognizes tuples of the various classes. An example of a confusion matrix for the two classes *buys computer* = *yes* (positive) and *buys computer* = *no* (negative) is given in Figure 8.15.

error rate or **misclassification rate** of a classifier, *M*, which is simply 1- *accuracy(M)*, where *accuracy(M)* is the accuracy of *M*. This also can be computed as

$$error\ rate = \frac{FP + FN}{P + N}. \qquad (8.22)$$

If we were to use the training set (instead of a test set) to estimate the error rate of a model, this quantity is known as the **resubstitution error**. This error estimate is optimistic of the true error rate (and similarly, the corresponding accuracy estimate is optimistic) because the model is not tested on any samples that it has not already seen.

We now consider the **class imbalance problem**, where the main class of interest is rare. That is, the data set distribution reflects a significant majority of the negative class and a minority positive class. For example, in fraud detection applications, the class of interest (or positive class) is *"fraud,"* which occurs much less frequently than the negative *"nonfraudulant"* class. In medical data, there may be a rare class, such as *"cancer."* Suppose that you have trained a classifier to classify medical data tuples, where the class label attribute is *"cancer"* and the possible class values are *"yes"* and *"no."* An accuracy rate of, say, 97% may make the classifier seem quite accurate, but what if only, say, 3% of the training tuples are actually cancer? Clearly, an accuracy rate of 97% may not be acceptable—the classifier could be correctly labeling only the noncancer tuples, for instance, and misclassifying all the cancer tuples. Instead, we need other measures, which access how well the classifier can recognize the positive tuples (*cancer* = *yes*) and how well it can recognize the negative tuples (*cancer* = *no*).

The **sensitivity** and **specificity** measures can be used, respectively, for this purpose. Sensitivity is also referred to as the *true positive* (*recognition*) *rate* (i.e., the proportion of positive tuples that are correctly identified), while specificity is the *true negative rate* (i.e., the proportion of negative tuples that are correctly identified). These measures are defined as

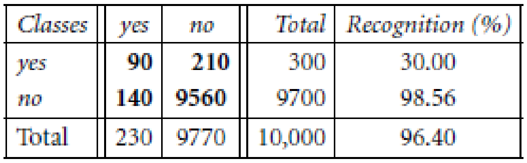
$$sensitivity = \frac{TP}{P} \tag{8.23}$$

$$specificity = \frac{TN}{N}. \tag{8.24}$$

It can be shown that accuracy is a function of sensitivity and specificity:

$$accuracy = sensitivity\frac{P}{(P+N)} + specificity\frac{N}{(P+N)}. \tag{8.25}$$

Example 8.9 Sensitivity and specificity. Figure 8.16 shows a confusion matrix for medical data where the class values are *yes* and *no* for a class label attribute, *cancer*. The sensitivity of the classifier is 90/300 = 30.00%. The specificity is 9560/9700= 98.56%. The classifier's overall accuracy is 9650/10,000= 96.50%. Thus, we note that although the classifier has a high accuracy, it's ability to correctly label the positive (rare) class is poor given its low sensitivity. It has high specificity, meaning that it can accurately recognize negative tuples.



Classes	*yes*	*no*	*Total*	*Recognition (%)*
yes	**90**	**210**	300	30.00
no	**140**	**9560**	9700	98.56
Total	230	9770	10,000	96.40

Figure 8.16 Confusion matrix for the classes *cancer* = *yes* and *cancer* = *no*.

The *precision* and *recall* measures are also widely used in classification. **Precision** can be thought of as a measure of *exactness* (i.e., what percentage of tuples labeled as positive are actually such), whereas **recall** is a measure of *completeness* (what percentage of positive tuples are labeled as such). If recall seems familiar, that's because it is the same as sensitivity (or the *true positive rate*). These measures can be computed as

$$precision = \frac{TP}{TP+FP} \tag{8.26}$$

$$recall = \frac{TP}{TP+FN} = \frac{TP}{P}. \tag{8.27}$$

Example 8.10 Precision and recall. The precision of the classifier in Figure 8.16 for the *yes* class is 90/230 = 39.13%. The recall is 90/300= 30.00%.

Source: Data Mining Concepts and Techniques, 3rd Edition, Han, Kamber and Pei

A perfect precision score of 1.0 for a class *C* means that every tuple that the classifier labeled as belonging to class *C* does indeed belong to class *C*. However, it does not tell us anything about the number of class *C* tuples that the classifier mislabeled. A perfect recall score of 1.0 for *C* means that every item from class *C* was labeled as such, but it does not tell us how many other tuples were incorrectly labeled as belonging to class *C*. There tends to be an inverse relationship between precision and recall, where it is possible to increase one at the cost of reducing the other. For example, our medical classifier may achieve high precision by labeling all cancer tuples that present a certain way as *cancer*, but may have low recall if it mislabels many other instances of *cancer* tuples. Precision and recall scores are typically used together, where precision values are compared for a fixed value of recall, or vice versa. For example, we may compare precision values at a recall value of, say, 0.75.

An alternative way to use precision and recall is to combine them into a single measure. This is the approach of the *F* measure (also known as the *F*1 score or *F*-score) and the F_β measure. They are defined as

$$F = \frac{2 \times precision \times recall}{precision + recall} \tag{8.28}$$

$$F_\beta = \frac{(1+\beta^2) \times precision \times recall}{\beta^2 \times precision + recall}, \tag{8.29}$$

where β is a non-negative real number. The *F* measure is the *harmonic mean* of precision and recall (the proof of which is left as an exercise). It gives equal weight to precision and recall. The F_β measure is a weighted measure of precision and recall. It assigns β times as much weight to recall as to precision. Commonly used F_β measures are *F*2 (which weights recall twice as much as precision) and *F*0.5 (which weights precision twice as much as recall).

"Are there other cases where accuracy may not be appropriate?" In classification problems, it is commonly assumed that all tuples are uniquely classifiable, that is, that each training tuple can belong to only one class. Yet, owing to the wide diversity of data in large databases, it is not always reasonable to assume that all tuples are uniquely classifiable. Rather, it is more probable to assume that each tuple may belong to more than one class. How then can the accuracy of classifiers on large databases be measured? The accuracy measure is not appropriate, because it does not take into account the possibility of tuples belonging to more than one class.

In addition to accuracy-based measures, classifiers can also be compared with respect to the following additional aspects:

Speed: This refers to the computational costs involved in generating and using the given classifier.

Robustness: This is the ability of the classifier to make correct predictions given noisy data or data with missing values. Robustness is typically assessed with a series of synthetic data sets representing increasing degrees of noise and missing values.

Scalability: This refers to the ability to construct the classifier efficiently given large amounts of data. Scalability is typically assessed with a series of data sets of increasing size.

Interpretability: This refers to the level of understanding and insight that is provided by the classifier or predictor. Interpretability is subjective and therefore more difficult to assess.

Source: Data Mining Concepts and Techniques, 3rd Edition, Han, Kamber and Pei

In summary, we have presented several evaluation measures. The accuracy measure works best when the data classes are fairly evenly distributed. Other measures, such as sensitivity (or recall), specificity, precision, *F*, and F_β, are better suited to the class imbalance problem, where the main class of interest is rare.

Holdout Method and Random Subsampling

The **holdout** method is what we have alluded to so far in our discussions about accuracy. In this method, the given data are randomly partitioned into two independent sets, a *training set* and a *test set*. Typically, two-thirds of the data are allocated to the training set, and the remaining one-third is allocated to the test set. The training set is used to derive the model. The model's accuracy is then estimated with the test set

Random subsampling is a variation of the holdout method in which the holdout method is repeated *k* times. The overall accuracy estimate is taken as the average of the accuracies obtained from each iteration.

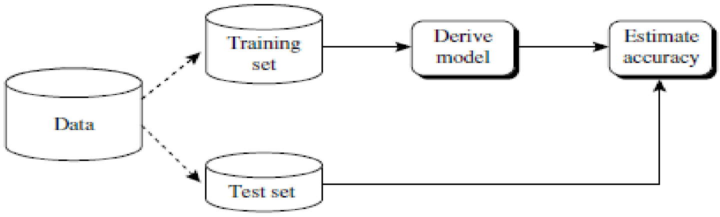


Figure 8.17 Estimating accuracy with the holdout method.

Cross-Validation

In ***k*-fold cross-validation**, the initial data are randomly partitioned into *k* mutually exclusive subsets or "folds," *D*1, *D*2, ... , *Dk*, each of approximately equal size. Training and testing is performed *k* times. In iteration *i*, partition *Di* is reserved as the test set, and the remaining partitions are collectively used to train the model. That is, in the first iteration, subsets *D*2, ... , *Dk* collectively serve as the training set to obtain a first model, which is tested on *D*1; the second iteration is trained on subsets *D*1, *D*3, ... , *Dk* and tested on *D*2; and so on. Unlike the holdout and random subsampling methods, here each sample is used the same number of times for training and once for testing. For classification, the accuracy estimate is the overall number of correct classifications from the *k* iterations, divided by the total number of tuples in the initial data.

Leave-one-out is a special case of *k*-fold cross-validation where *k* is set to the number of initial tuples. That is, only one sample is "left out" at a time for the test set. In **stratified cross-validation**, the folds are stratified so that the class distribution of the tuples in each fold is approximately the same as that in the initial data.

In general, stratified 10-fold cross-validation is recommended for estimating accuracy (even if computation power allows using more folds) due to its relatively low bias and variance.

Source: Data Mining Concepts and Techniques, 3rd Edition, Han, Kamber and Pei

Bootstrap

the **bootstrap method** samples the given training tuples uniformly *with replacement*. That is, each time a tuple is selected, it is equally likely to be selected again and re-added to the training set. For instance, imagine a machine that randomly selects tuples for our training set. In *sampling with replacement*, the machine is allowed to select the same tuple more than once.

There are several bootstrap methods. A commonly used one is the **.632 bootstrap**, which works as follows. Suppose we are given a data set of *d* tuples. The data set is sampled *d* times, with replacement, resulting in a *bootstrap sample* or training set of *d* samples. It is very likely that some of the original data tuples will occur more than once in this sample. The data tuples that did not make it into the training set end up forming the test set. Suppose we were to try this out several times. As it turns out, on average, 63.2% of the original data tuples will end up in the bootstrap sample, and the remaining 36.8% will form the test set (hence, the name, .632 bootstrap).

"Where does the figure, 63.2%, come from?" Each tuple has a probability of 1/*d* of being selected, so the probability of not being chosen is $(1-1/d)$. We have to select *d* times, so the probability that a tuple will not be chosen during this whole time is $(1-1/d)^d$. If *d* is large, the probability approaches e^{-1} = 0.368. Thus, 36.8% of tuples will not be selected for training and thereby end up in the test set, and the remaining 63.2% will form the training set.

We can repeat the sampling procedure *k* times, where in each iteration, we use the current test set to obtain an accuracy estimate of the model obtained from the current bootstrap sample. The overall accuracy of the model, *M*, is then estimated as

$$Acc(M) = \frac{1}{k}\sum_{i=1}^{k}(0.632 \times Acc(M_i)_{test\_set} + 0.368 \times Acc(M_i)_{train\_set}), \qquad (8.30)$$

where *Acc(Mi)test set* is the accuracy of the model obtained with bootstrap sample *i* when it is applied to test set *i*. *Acc(Mi)train set* is the accuracy of the model obtained with bootstrap sample *i* when it is applied to the original set of data tuples. Bootstrapping tends to be overly optimistic. It works best with small data sets.

Model Selection Using Statistical Tests of Significance

Suppose that we have generated two classification models, *M*1 and *M*2, from our data. We have performed 10-fold cross-validation to obtain a mean error rate8 for each. How can we determine which model is best? It may seem intuitive to select the model with the lowest error rate; however, the mean error rates are just *estimates* of error on the true population of future data cases. There can be considerable variance between error rates within any given 10-fold cross-validation experiment. Although the mean error rates obtained for *M*1 and *M*2 may appear different, that difference may not be statistically significant. What if any difference between the two may just be attributed to chance?

What do we need to perform the statistical test? Suppose that for each model, we did 10-fold cross-validation, say, 10 times, each time using a different 10-fold data partitioning. Each partitioning is independently drawn. We can average the 10 error rates obtained each for*M*1 and *M*2, respectively,

Source: Data Mining Concepts and Techniques, 3rd Edition, Han, Kamber and Pei

to obtain the mean error rate for each model. For a given model, the individual error rates calculated in the cross-validations may be considered as different, independent samples from a probability distribution. In general, they follow a *t-distribution with* k -1 *degrees of freedom* where, here, $k = 10$. (This distribution looks very similar to a normal, or Gaussian, distribution even though the functions defining the two are quite different. Both are unimodal, symmetric, and bellshaped.) This allows us to do hypothesis testing where the significance test used is the ***t*-test**, or **Student's *t*-test**. Our hypothesis is that the two models are the same, or in other words, that the difference in mean error rate between the two is zero. If we can reject this hypothesis (referred to as the *null hypothesis*), then we can conclude that the difference between the two models is statistically significant, in which case we can select the model with the lower error rate.

In data mining practice, we may often employ a single test set, that is, the same test set can be used for both $M1$ and $M2$. In such cases, we do a **pairwise comparison** of the two models *for each* 10-fold cross-validation round. That is, for the ith round of 10-fold cross-validation, the same cross-validation partitioning is used to obtain an error rate for $M1$ and for $M2$.

To determine whether$M1$ and$M2$ are significantly different, we compute t and select a **significance level**, *sig*. In practice, a significance level of 5% or 1% is typically used. We then consult a table for the *t-distribution*

However, because the t-distribution is symmetric, typically only the upper percentage points of the distribution are shown. Therefore, we look up the table value for $z = sig/2$, which in this case is 0.025, where z is also referred to as a **confidence limit**. If $t > z$ or $t < -z$, then our value of t lies in the rejection region, within the distribution's tails. This means that we can reject the null hypothesis that the means of $M1$ and $M2$ are the same and conclude that there is a statistically significant difference between the two models. Otherwise, if we cannot reject the null hypothesis, we conclude that any difference between $M1$ and $M2$ can be attributed to chance.

If two test sets are available instead of a single test set, then a nonpaired version of the t -test is used, where the variance between the means of the two models is estimated as

$$var(M_1 - M_2) = \sqrt{\frac{var(M_1)}{k_1} + \frac{var(M_2)}{k_2}}, \qquad (8.33)$$

and $k1$ and $k2$ are the number of cross-validation samples (in our case, 10-fold crossvalidation rounds) used for $M1$ and $M2$, respectively. This is also known as the **two sample *t*-test**.

Comparing Classifiers Based on Cost–Benefit and ROC Curves

The true positives, true negatives, false positives, and false negatives are also useful in assessing the **costs and benefits** (or risks and gains) associated with a classification model. The cost associated with a false negative (such as incorrectly predicting that a cancerous patient is not cancerous) is far greater than those of a false positive (incorrectly yet conservatively labeling a noncancerous patient as cancerous). In such cases, we can outweigh one type of error over another by assigning a different cost to each. These costs may consider the danger to the patient, financial costs of resulting therapies, and other hospital costs. Similarly, the benefits associated with a true positive decision may be different than those of a true negative. Up to now, to compute classifier accuracy, we

Source: Data Mining Concepts and Techniques, 3rd Edition, Han, Kamber and Pei

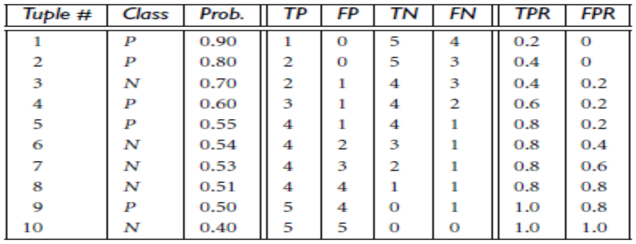
have assumed equal costs and essentially divided the sum of true positives and true negatives by the total number of test tuples.

Receiver operating characteristic curves are a useful visual tool for comparing two classification models. ROC curves come from signal detection theory that was developed during World War II for the analysis of radar images. An ROC curve for a given model shows the trade-off between the *true positive rate* (*TPR*) and the *false positive rate* (*FPR*).10 Given a test set and a model, *TPR* is the proportion of positive (or "yes") tuples that are correctly labeled by the model; *FPR* is the proportion of negative (or "no") tuples that are mislabeled as positive. Given that *TP*, *FP*, *P*, and *N* are the number of true positive, false positive, positive, and negative tuples, respectively, fromSection 8.5.1 we know that *TPR*= *TP/ P* , which is sensitivity. Furthermore, *FPR* = *FP/N* , which is 1-*specificity*.

For a two-class problem, an ROC curve allows us to visualize the trade-off between the rate at which the model can accurately recognize positive cases versus the rate at which it mistakenly identifies negative cases as positive for different portions of the test set. Any increase in *TPR* occurs at the cost of an increase in *FPR*. The area under the ROC curve is a measure of the accuracy of the model.

To plot an ROC curve for a given classification model, *M*, the model must be able to return a probability of the predicted class for each test tuple. With this information, we rank and sort the tuples so that the tuple that is most likely to belong to the positive or "yes" class appears at the top of the list, and the tuple that is least likely to belong to the positive class lands at the bottomof the list.

Example 8.11 Plotting an ROC curve. Figure 8.18 shows the probability value (column 3) returned by a probabilistic classifier for each of the 10 tuples in a test set, sorted by decreasing probability order. Column 1 is merely a tuple identification number, which aids in our explanation. Column 2 is the actual class label of the tuple. There are five positive tuples and five negative tuples, thus *P* = 5 and *N* = 5. As we examine the known class label of each tuple, we can determine the values of the remaining columns, *TP*, *FP*, *TN*, *FN*, *TPR*, and *FPR*. We start with tuple 1, which has the highest probability score, and take that score as our threshold, that is, *t* = 0.9. Thus, the classifier considers tuple 1 to be positive, and all the other tuples are considered negative. Since the actual class label of tuple 1 is positive, we have a true positive, hence *TP* = 1 and *FP* = 0. Among the



Tuple #	*Class*	*Prob.*	*TP*	*FP*	*TN*	*FN*	*TPR*	*FPR*
1	*P*	0.90	1	0	5	4	0.2	0
2	*P*	0.80	2	0	5	3	0.4	0
3	*N*	0.70	2	1	4	3	0.4	0.2
4	*P*	0.60	3	1	4	2	0.6	0.2
5	*P*	0.55	4	1	4	1	0.8	0.2
6	*N*	0.54	4	2	3	1	0.8	0.4
7	*N*	0.53	4	3	2	1	0.8	0.6
8	*N*	0.51	4	4	1	1	0.8	0.8
9	*P*	0.50	5	4	0	1	1.0	0.8
10	*N*	0.40	5	5	0	0	1.0	1.0

Figure 8.18 Tuples sorted by decreasing score, where the score is the value returned by a probabilistic classifier.

Source: Data Mining Concepts and Techniques, 3rd Edition, Han, Kamber and Pei

remaining nine tuples, which are all classified as negative, five actually are negative (thus, *TN* = 5). The remaining four are all actually positive, thus, *FN* = 4. We can therefore compute *TPR* = *TP/P*= 1/5= 0.2, while *FPR* = 0. Thus, we have the point (0.2,0) for the ROC curve.

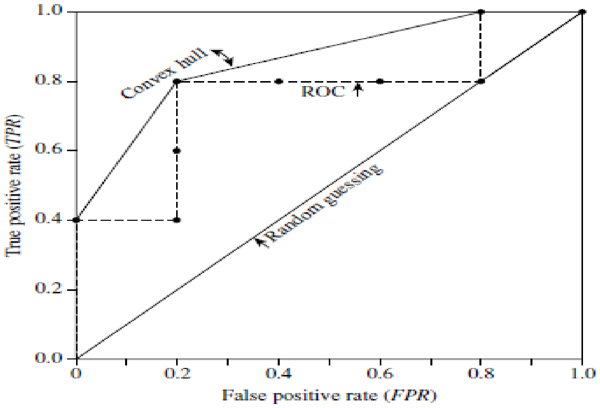


Figure 8.19 ROC curve for the data in Figure 8.18.

Next, threshold *t* is set to 0.8, the probability value for tuple 2, so this tuple is now also considered positive, while tuples 3 through 10 are considered negative. The actual class label of tuple 2 is positive, thus now *TP* = 2. The rest of the row can easily be computed, resulting in the point (0.4,0). Next, we examine the class label of tuple 3 and let *t* be 0.7, the probability value returned by the classifier for that tuple. Thus, tuple 3 is considered positive, yet its actual label is negative, and so it is a false positive. Thus, *TP* stays the same and *FP* increments so that *FP* = 1. The rest of the values in the row can also be easily computed, yielding the point (0.4, 0.2). The resulting ROC graph, from examining each tuple, is the jagged line shown in Figure 8.19.

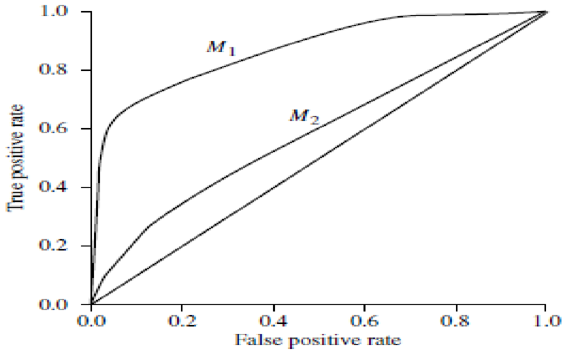


Figure 8.20 ROC curves of two classification models, M_1 and M_2. The diagonal shows where, for every true positive, we are equally likely to encounter a false positive. The closer an ROC curve is to the diagonal line, the less accurate the model is. Thus, $M1$ is more accurate here.

The diagonal line representing random guessing is also shown. Thus, the closer the ROC curve of a model is to the diagonal line, the less accurate the model. If the model is really good, initially we are more likely to encounter true positives as we move down the ranked list. Thus, the curve moves steeply up from zero. Later, as we start to encounter fewer and fewer true positives, and more and more false positives, the curve eases off and becomes more horizontal.

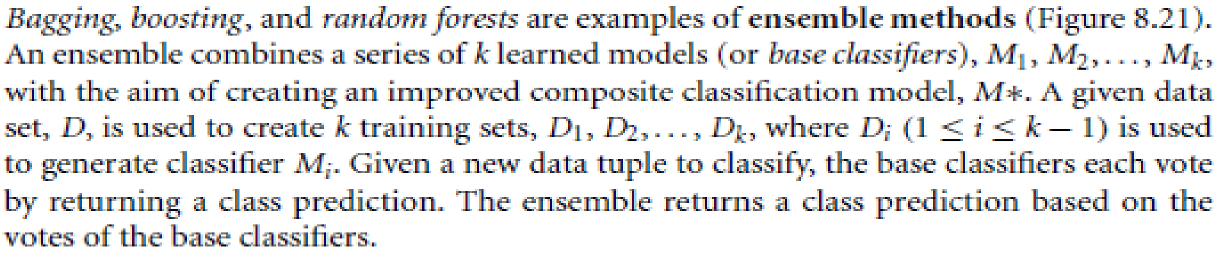
Unit-IV

Techniques to Improve Classification Accuracy

In this section, you will learn some tricks for increasing classification accuracy. We focus on *ensemble methods*. An ensemble for classification is a composite model, made up of a combination of classifiers. The individual classifiers vote, and a class label prediction is returned by the ensemble based on the collection of votes. Ensembles tend to be more accurate than their component classifiers.

Traditional learning models assume that the data classes are well distributed. In many real-world data domains, however, the data are class-imbalanced, where the main class of interest is represented by only a few tuples. This is known as the *class imbalance problem*.

Introducing Ensemble Methods



Bagging, *boosting*, and *random forests* are examples of **ensemble methods** (Figure 8.21). An ensemble combines a series of k learned models (or *base classifiers*), $M_1, M_2, \ldots, M_k$, with the aim of creating an improved composite classification model, $M*$. A given data set, D, is used to create k training sets, $D_1, D_2, \ldots, D_k$, where D_i $(1 \leq i \leq k - 1)$ is used to generate classifier M_i. Given a new data tuple to classify, the base classifiers each vote by returning a class prediction. The ensemble returns a class prediction based on the votes of the base classifiers.

An ensemble tends to be more accurate than its base classifiers. For example, consider an ensemble that performs majority voting. That is, given a tuple ***X*** to classify, it collects the class label predictions returned fromthe base classifiers and outputs the class in majority. The base classifiers may make mistakes, but the ensemble will misclassify ***X*** only if over half of the base classifiers are in error. Ensembles yield better results when there is significant diversity among the models. That is, ideally, there is little correlation among classifiers. The classifiers should also perform better than random guessing. Each base classifier can be allocated to a different CPU and so ensemble methods are parallelizable.

To help illustrate the power of an ensemble, consider a simple two-class problem described by two attributes, $x1$ and $x2$. The problem has a linear decision boundary. Figure 8.22(a) shows the decision boundary of a decision tree classifier on the problem. Figure 8.22(b) shows the decision boundary of an ensemble of decision tree classifiers on the same problem. Although the ensemble's decision boundary is still piecewise constant, it has a finer resolution and is better than that of a single tree.

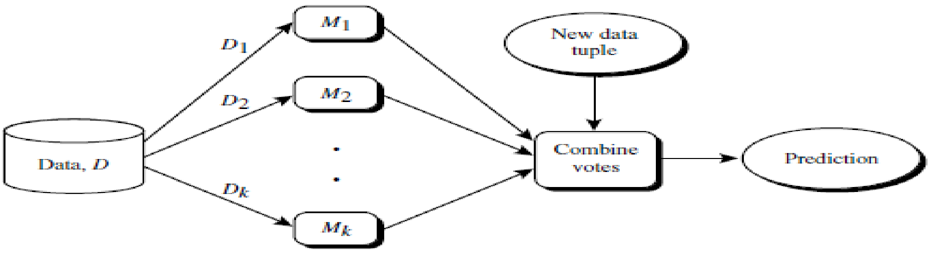


Figure 8.21 Increasing classifier accuracy: Ensemble methods generate a set of classification models, $M_1, M_2, \ldots, M_k$. Given a new data tuple to classify, each classifier "votes" for the class label of that tuple. The ensemble combines the votes to return a class prediction.

Source: Data Mining Concepts and Techniques, 3rd Edition, Han, Kamber and Pei

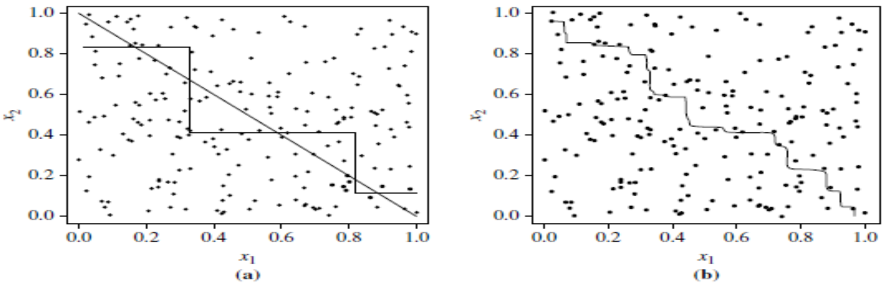
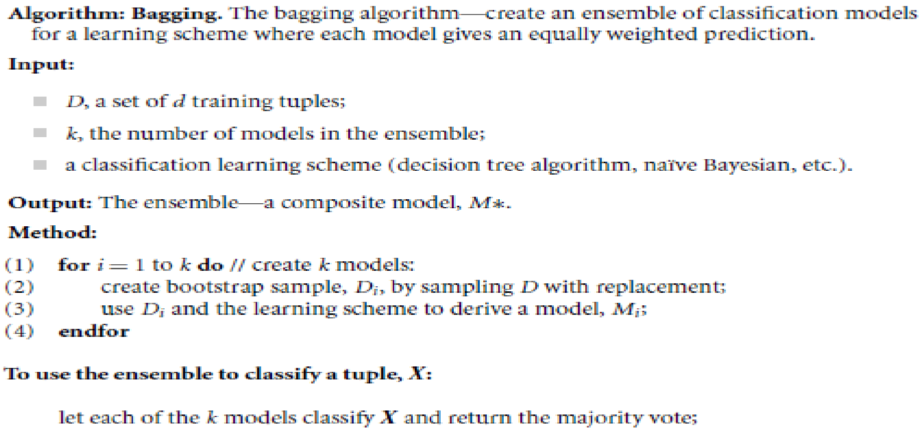


Figure 8.22 Decision boundary by (a) a single decision tree and (b) an ensemble of decision trees for a linearly separable problem (i.e., where the actual decision boundary is a straight line). The decision tree struggles with approximating a linear boundary. The decision boundary of the ensemble is closer to the true boundary. *Source:* From Seni and Elder [SE10]. © 2010 Morgan & Claypool Publishers; used with permission.

Bagging

We now take an intuitive look at how bagging works as a method of increasing accuracy. Suppose that you are a patient and would like to have a diagnosis made based on your symptoms. Instead of asking one doctor, you may choose to ask several. If a certain diagnosis occurs more than any other, you may choose this as the final or best diagnosis. That is, the final diagnosis is made based on a majority vote, where each doctor gets an equal vote. Now replace each doctor by a classifier, and you have the basic idea behind bagging. Intuitively, a majority vote made by a large group of doctors may be more reliable than a majority vote made by a small group.

Given a set, *D*, of *d* tuples, **bagging** works as follows. For iteration *i* (*i*= 1, 2,.... , *k),* a training set, *Di* , of *d* tuples is sampled with replacement from the original set of tuples, *D*. Note that the term *bagging* stands for *bootstrap aggregation*. Each training set is a bootstrap sample, as described in Section 8.5.4. Because sampling with replacement is used, some of the original tuples of *D* may not be included in *Di* , whereas others may occur more than once. A classifier model, *Mi* , is learned for each training set, *Di* . To classify an unknown tuple, ***X***, each classifier, *Mi* , returns its class prediction, which counts as one vote. The bagged classifier, *M**, counts the votes and assigns the class with the most votes to ***X***. Bagging can be applied to the prediction of continuous values by taking the average value of each prediction for a given test tuple.



Algorithm: Bagging. The bagging algorithm—create an ensemble of classification models for a learning scheme where each model gives an equally weighted prediction.

Input:

- *D*, a set of *d* training tuples;
- *k*, the number of models in the ensemble;
- a classification learning scheme (decision tree algorithm, naïve Bayesian, etc.).

Output: The ensemble—a composite model, *M**.

Method:

```
for $i = 1$ to $k$ do // create $k$ models:
    create bootstrap sample, $D_i$, by sampling $D$ with replacement;
    use $D_i$ and the learning scheme to derive a model, $M_i$;
endfor
```

To use the ensemble to classify a tuple, *X*:

let each of the *k* models classify ***X*** and return the majority vote;

Figure 8.23 Bagging.

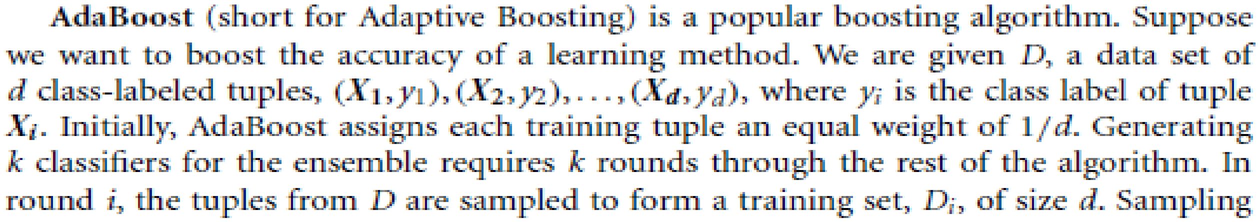
Source: Data Mining Concepts and Techniques, 3rd Edition, Han, Kamber and Pei

It will not be considerably worse and is more robust to the effects of noisy data and overfitting. The increased accuracy occurs because the composite model reduces the variance of the individual classifiers.

Boosting and AdaBoost

As in the previous section, suppose that as a patient, you have certain symptoms. Instead of consulting one doctor, you choose to consult several. Suppose you assign weights to the value or worth of each doctor's diagnosis, based on the accuracies of previous diagnoses they have made. The final diagnosis is then a combination of the weighted diagnoses. This is the essence behind boosting.

In **boosting**, weights are also assigned to each training tuple. A series of *k* classifiers is iteratively learned. After a classifier, *Mi* , is learned, the weights are updated to allow the subsequent classifier,*Mi*C1, to "pay more attention" to the training tuples that were misclassified by *Mi* . The final boosted classifier, *M**, combines the votes of each individual classifier, where the weight of each classifier's vote is a function of its accuracy.



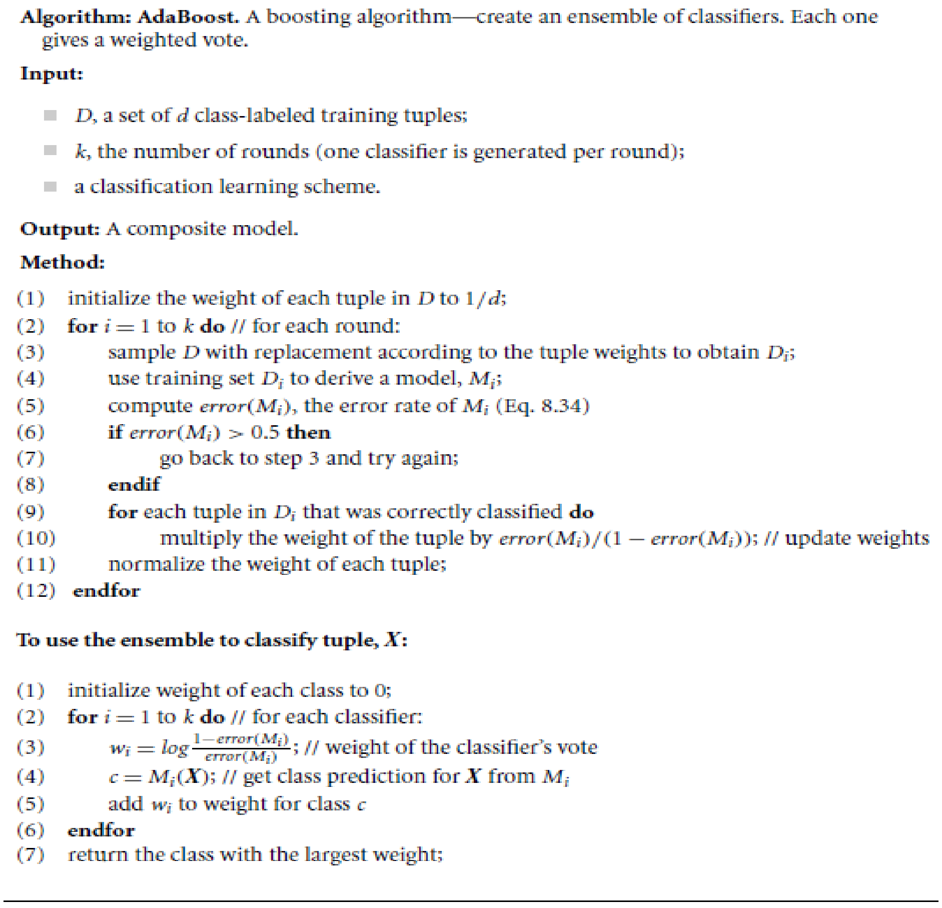
AdaBoost (short for Adaptive Boosting) is a popular boosting algorithm. Suppose we want to boost the accuracy of a learning method. We are given D, a data set of d class-labeled tuples, $(X_1, y_1), (X_2, y_2), \ldots, (X_d, y_d)$, where y_i is the class label of tuple X_i. Initially, AdaBoost assigns each training tuple an equal weight of $1/d$. Generating k classifiers for the ensemble requires k rounds through the rest of the algorithm. In round i, the tuples from D are sampled to form a training set, D_i, of size d. Sampling

with replacement is used—the same tuple may be selected more than once. Each tuple's chance of being selected is based on its weight. A classifier model, *Mi* , is derived from the training tuples of *Di* . Its error is then calculated using *Di* as a test set. The weights of the training tuples are then adjusted according to how they were classified.

If a tuple was incorrectly classified, its weight is increased. If a tuple was correctly classified, its weight is decreased. A tuple's weight reflects how difficult it is to classify— the higher the weight, the more often it has been misclassified. These weights will be used to generate the training samples for the classifier of the next round. The basic idea is that when we build a classifier, we want it to focus more on the misclassified tuples of the previous round. Some classifiers may be better at classifying some "difficult" tuples than others. In this way, we build a series of classifiers that complement each other.

Therefore, sometimes the resulting "boosted" model may be less accurate than a single model derived fromthe same data. Bagging is less susceptible to model overfitting.While both can significantly improve accuracy in comparison to a single model, boosting tends to achieve greater accuracy.

Source: Data Mining Concepts and Techniques, 3rd Edition, Han, Kamber and Pei



Algorithm: AdaBoost. A boosting algorithm—create an ensemble of classifiers. Each one gives a weighted vote.

Input:

- D, a set of d class-labeled training tuples;
- k, the number of rounds (one classifier is generated per round);
- a classification learning scheme.

Output: A composite model.

Method:

```
(1)  initialize the weight of each tuple in $D$ to $1/d$;
(2)  for $i = 1$ to $k$ do // for each round:
(3)      sample $D$ with replacement according to the tuple weights to obtain $D_i$;
(4)      use training set $D_i$ to derive a model, $M_i$;
(5)      compute $error(M_i)$, the error rate of $M_i$ (Eq. 8.34)
(6)      if $error(M_i) > 0.5$ then
(7)          go back to step 3 and try again;
(8)      endif
(9)      for each tuple in $D_i$ that was correctly classified do
(10)         multiply the weight of the tuple by $error(M_i)/(1 - error(M_i))$; // update weights
(11)     normalize the weight of each tuple;
(12) endfor
```

To use the ensemble to classify tuple, X:

```
(1)  initialize weight of each class to 0;
(2)  for $i = 1$ to $k$ do // for each classifier:
(3)      $w_i = log\frac{1-error(M_i)}{error(M_i)}$; // weight of the classifier's vote
(4)      $c = M_i(X)$; // get class prediction for $X$ from $M_i$
(5)      add $w_i$ to weight for class $c$
(6)  endfor
(7)  return the class with the largest weight;
```

Figure 8.24 AdaBoost, a boosting algorithm.

Random Forests

Imagine that each of the classifiers in the ensemble is a *decision tree* classifier so that the collection of classifiers is a "forest." The individual decision trees are generated using a random selection of attributes at each node to determine the split. More formally, each tree depends on the values of a random vector sampled independently and with the same distribution for all trees in the forest. During classification, each tree votes and the most popular class is returned.

Random forests can be built using bagging (Section 8.6.2) in tandem with random attribute selection. A training set, *D*, of *d* tuples is given. The general procedure to generate *k* decision trees for the ensemble is as follows. For each iteration, *i (i*= 1, 2,... , *k*), a training set, *Di* , of *d* tuples is sampled with replacement from *D*. That is, each *Di* is a bootstrap sample of *D* (Section 8.5.4), so that some tuples may occur more than once

in *Di* , while others may be excluded. Let *F* be the number of attributes to be used to determine the split at each node, where *F* is much smaller than the number of available attributes. To construct a decision tree classifier, *Mi* , randomly select, at each node, *F* attributes as candidates for the split at the node. The CART methodology is used to grow the trees. The trees are grown to maximum size and are not pruned. Random forests formed this way, with *random input selection*, are called Forest-RI.

Another form of random forest, called Forest-RC, uses *random linear combinations* of the input attributes. Instead of randomly selecting a subset of the attributes, it creates new attributes (or features) that are a linear combination of the existing attributes. That is, an attribute is generated by specifying L, the number of original attributes to be combined. At a given node, L attributes are randomly selected and added together with coefficients that are uniform random numbers on [-1, 1]. F linear combinations are generated, and a search is made over these for the best split. This form of random forest is useful when there are only a few attributes available, so as to reduce the correlation between individual classifiers.

Random forests are comparable in accuracy to AdaBoost, yet are more robust to errors and outliers. The generalization error for a forest converges as long as the number of trees in the forest is large. Thus, overfitting is not a problem. The accuracy of a random forest depends on the strength of the individual classifiers and a measure of the dependence between them. The ideal is to maintain the strength of individual classifiers without increasing their correlation. Random forests are insensitive to the number of attributes selected for consideration at each split.

Improving Classification Accuracy of Class-Imbalanced Data

Given two-class data, the data are class-imbalanced if the main class of interest (the positive class) is represented by only a few tuples, while the majority of tuples represent the negative class. For multiclass-imbalanced data, the data distribution of each class differs substantially where, again, the main class or classes of interest are rare. The class imbalance problem is closely related to cost-sensitive learning, wherein the costs of errors, per class, are not equal. In medical diagnosis, for example, it is much more costly to falsely diagnose a cancerous patient as healthy (a false negative) than to misdiagnose a healthy patient as having cancer (a false positive). A false negative error could lead to the loss of life and therefore is much more expensive than a false positive error. Other applications involving class-imbalanced data include fraud detection, the detection of oil spills from satellite radar images, and fault monitoring.

Traditional classification algorithms aim to minimize the number of errors made during classification. They assume that the costs of false positive and false negative errors are equal. By assuming a balanced distribution of classes and equal error costs, they are therefore not suitable for class-imbalanced data.

In this section, we look at general approaches for *improving* the classification accuracy of class-imbalanced data. These approaches include (1) oversampling, (2) undersampling, (3) threshold moving, and (4) ensemble techniques. The first three do not involve any changes to the construction of the classification model. That is, oversampling and undersampling change the distribution of tuples in the training set; threshold moving affects how the model makes decisions when classifying new data.

Both oversampling and undersampling change the training data distribution so that the rare (positive) class is well represented. **Oversampling** works by resampling the positive tuples so that the resulting training set contains an equal number of positive and negative tuples. **Undersampling** works by decreasing the number of negative tuples. It randomly eliminates tuples from the majority (negative) class until there are an equal number of positive and negative tuples.

Example 8.12 Oversampling and undersampling. Suppose the original training set contains 100 positive and 1000 negative tuples. In oversampling, we replicate tuples of the rarer class to form a new training set containing 1000 positive tuples and 1000 negative tuples. In undersampling, we randomly eliminate negative tuples so that the new training set contains 100 positive tuples and 100 negative tuples.

Several variations to oversampling and undersampling exist. They may vary, for instance, in how tuples are added or eliminated. For example, the SMOTE algorithm uses oversampling where synthetic tuples are added, which are “close to” the given positive tuples in tuple space.

The **threshold-moving** approach to the class imbalance problem does not involve any sampling. It applies to classifiers that, given an input tuple, return a continuous output value. That is, for an input tuple, ***X***, such a classifier returns as output a mapping, $f(X) \rightarrow [0,1].$. Rather than manipulating the training tuples, this method returns a classification decision based on the output values.

Ensemble methods have also been applied to the class imbalance problem. The individual classifiers making up the ensemble may include versions of the approaches described here such as oversampling and threshold moving.

Source: Data Mining Concepts and Techniques, 3rd Edition, Han, Kamber and Pei