UNIT IV

Classification: Decision Tree Induction, Bayesian Classification, Rule Based Classification, Classification by Back Propagation, Support Vector Machines, Lazy Learners, Model Evaluation and Selection, Techniques to improve Classification Accuracy

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. Many classification methods have been proposed by researchers in machine learning, pattern recognition, and statistics.

Why 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.

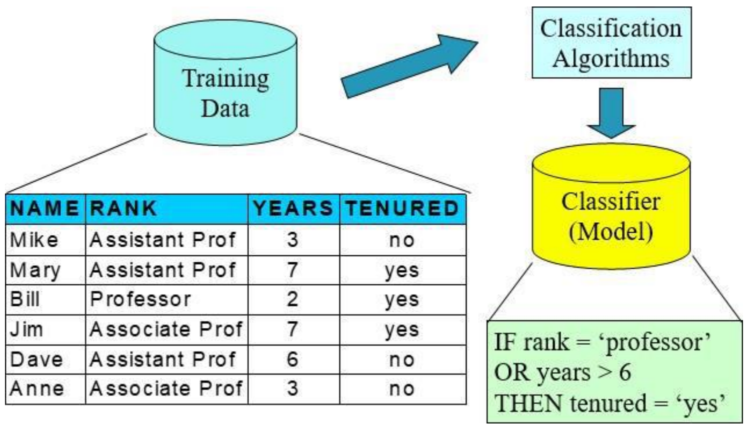
Suppose that the marketing manager wants to predict how much a given customer will spend during a sale at AllElectronics. This data analysis task is an example of **numeric prediction**, where the model constructed predicts a continuous-valued function, or ordered value, as opposed to a class label.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

Regression analysis is a statistical methodology that is most often used for numeric prediction; hence the two terms tend to be used synonymously, although other methods for numeric prediction exist. Classification and numeric prediction are the two major types of **prediction problems**.

General Approach for Classification:

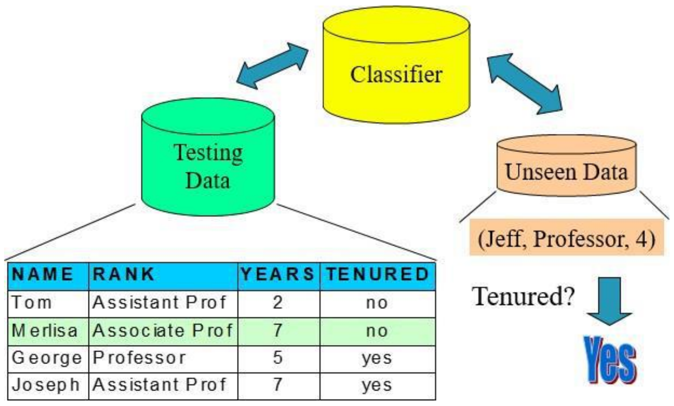
Data classification is a two-step process, consisting of a *learning step* (where a classification model is constructed) and a *classification step* (where the model is used to predict class labels for given data).

- 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.
- Each tuple/sample is assumed to belong to a predefined class, as determined by the class label attribute
- In the second step, 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.
- Accuracy rate is the percentage of test set samples that are correctly classified by the model



NAME	RANK	YEARS	TENURED
Mike	Assistant Prof	3	no
Mary	Assistant Prof	7	yes
Bill	Professor	2	yes
Jim	Associate Prof	7	yes
Dave	Assistant Prof	6	no
Anne	Associate Prof	3	no

Fig: Learning Step



NAME	RANK	YEARS	TENURED
Tom	Assistant Prof	2	no
Merlisa	Associate Prof	7	no
George	Professor	5	yes
Joseph	Assistant Prof	7	yes

Fig: Classification Step

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** (non leaf 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. Internal nodes are denoted by rectangles, and leaf nodes are denoted by ovals.

"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. Their representation of acquired knowledge in tree form is intuitive and generally easy to assimilate by humans. The learning and classification steps of decision tree induction are simple and fast.

Decision tree induction algorithms have been used for classification in many application areas such as medicine, manufacturing and production, financial analysis, astronomy, and molecular biology. Decision trees are the basis of several commercial rule induction systems.

During tree construction, *attribute selection measures* are used to select the attribute that best partitions the tuples into distinct classes. When decision trees are built, many of the branches may reflect noise or outliers in the training data. *Tree pruning* attempts to identify and remove such branches, with the goal of improving classification accuracy on unseen 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.

Decision Tree Algorithm:

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.

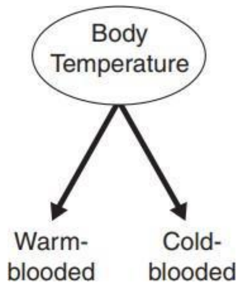
Method:

```
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 Dj be the set of data tuples in D satisfying outcome j; // a partition
12) if Dj 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(Dj , attribute list) to node N;
endfor
15) return N;
```

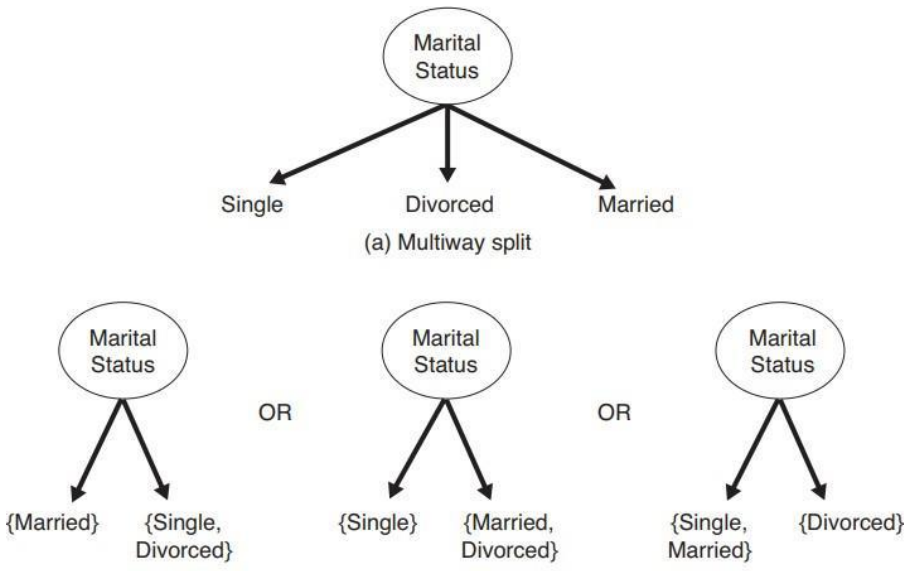
Methods for selecting best test conditions

Decision tree induction algorithms must provide a method for expressing an attribute test condition and its corresponding outcomes for different attribute types.

Binary Attributes: The test condition for a binary attribute generates two potential outcomes.



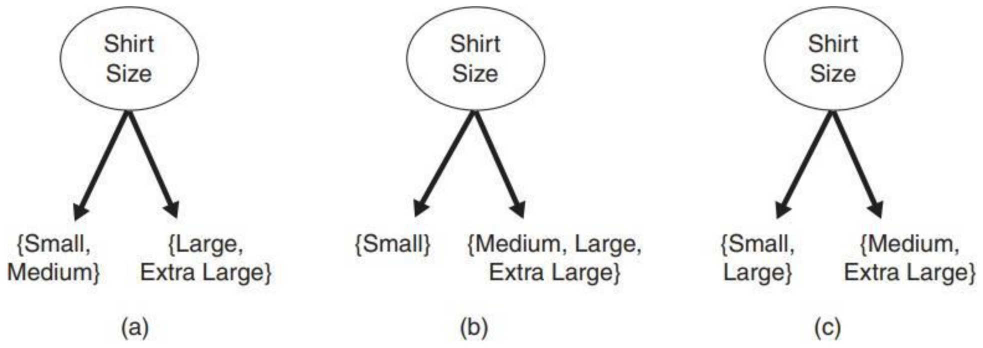
Nominal Attributes: These can have many values. These can be represented in two ways.



(a) Multiway split

(b) Binary split {by grouping attribute values}

Ordinal attributes: These can produce binary or multiway splits. The values can be grouped as long as the grouping does not violate the order property of attribute values.



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).
- Conceptually, the "best" splitting criterion is the one that most closely results in such a scenario. Attribute selection measures are also known as **splitting rules** because they determine how the tuples at a given node are to be split.
- The attribute selection measure provides a ranking for each attribute describing the given training tuples. The attribute having the best score for the 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 tree node created for partition D is labeled with the splitting criterion, branches are grown for each outcome of the criterion, and the tuples are partitioned accordingly.
- There are three popular attribute selection measures—*information gain*, *gain ratio*, and *Gini index*.

Information Gain

ID3 uses **information gain** as its attribute selection measure. Let node *N* represent or hold the tuples of partition *D*. The attribute with the highest information gain is chosen as the splitting attribute for node *N*. This attribute minimizes the information needed to classify the tuples in the resulting partitions and reflects the least randomness or "impurity" in these partitions. Such an approach minimizes the expected number of tests needed to classify a given tuple and guarantees that a simple (but not necessarily the simplest) tree is found.

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

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

Where $p_i$is the nonzero probability that an arbitrary tuple in *D* belongs to class $C_i$and is estimated by $|C_{i,D}|/|D|$. A log function to the base 2 is used, because the information is encoded in bits.*Info*(*D*) is also known as the **entropy** of *D*.

Information needed after using A to split D into V partitions.

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

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,

$$Gain(A) = Info(D) - Info_A(D).$$

The attribute *A* with the highest information gain, *Gain*(*A*), is chosen as the splittingattribute at node*N*. This is equivalent to saying that we want to partition on the attribute*A* that would do the "best classification," so that the amount of information still requiredto finish classifying the tuples is minimal.

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

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

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

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

Gini Index

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

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

Where $p_i$is the nonzero probability that an arbitrary tuple in *D* belongs to class $C_i$and is estimated by $|C_{i,D}|/|D|$ over *m* classes.

Note: The Gini index considers a binary split for each attribute.

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

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

- 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

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

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. Such methods typically use statistical measures to remove the least-reliable branches.
- Pruned trees tend to be smaller and less complex and, thus, easier to comprehend.
- They are usually faster and better at correctly classifying independent test data (i.e., of previously unseen tuples) than unpruned trees.

"How does tree pruning work?" There are two common approaches to tree pruning:

prepruning and *postpruning*.

- In the **prepruning** approach, a tree is "pruned" by halting its construction early. Upon halting, the node becomes a leaf. The leaf may hold the most frequent class among the subset tuples or the probability distribution of those tuples.
- If partitioning the tuples at a node would result in a split that falls below a prespecified threshold, then further partitioning of the given subset is halted. There are difficulties, however, in choosing an appropriate threshold.
- In the postpruning, which removes subtrees from a "fully grown" tree. A subtree at a given node is pruned by removing its branches and replacing it with a leaf. The leaf is labeled with the most frequent class among the subtree being replaced.

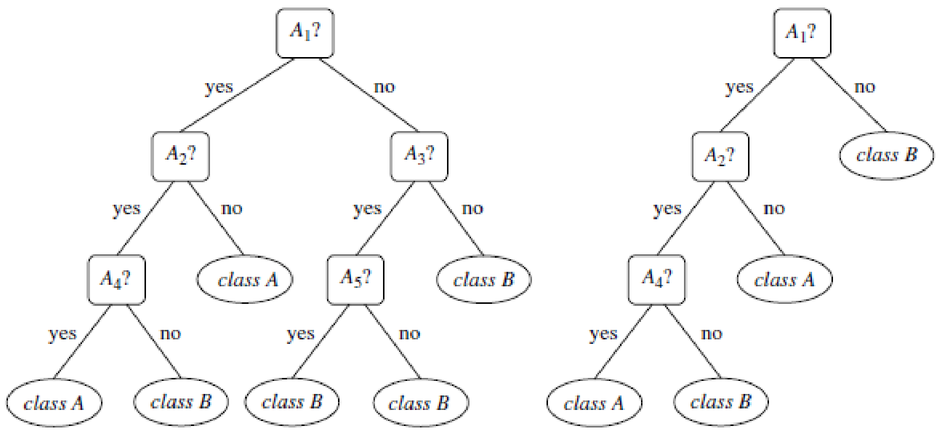


Fig: Unpruned and Pruned Trees

- The **cost complexity** pruning algorithm used in CART is an example of the postpruning approach.
- This approach considers the cost complexity of a tree to be a function of the number of leaves in the tree and the error rate of the tree (where the **error rate** is the percentage of tuples misclassified by the tree). It starts from the bottom of the tree.
- For each internal node, *N*, it computes the cost complexity of the subtree at *N*, and the cost complexity of the subtree at *N* if it were to be pruned (i.e., replaced by a leaf node).
- The two values are compared. If pruning the subtree at node *N* would result in a smaller cost complexity, then the subtree is pruned. Otherwise, it is kept.
- A **pruning set** of class-labeled tuples is used to estimate cost complexity.

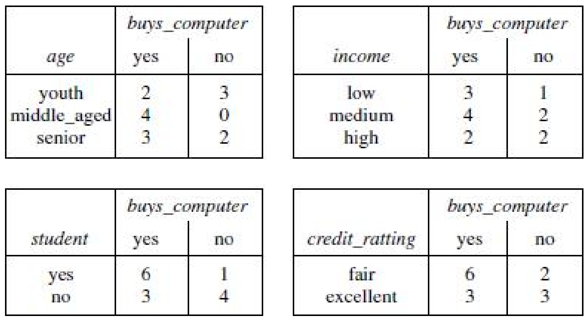
- This set is independent of the training set used to build the unpruned tree and of any test set usedfor accuracy estimation.
- The algorithm generates a set of progressively pruned trees. Ingeneral, the smallest decision tree that minimizes the cost complexity is preferred.
- C4.5 uses a method called **pessimistic pruning**, which is similar to the cost complexity method in that it also uses error rate estimates to make decisions regarding subtree pruning.

Scalability of 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. The pioneering decision tree algorithms that we have discussed so far have the restriction that the training tuples should reside *in memory*.

In data mining applications, very large training sets of millions of tuples are common. Most often, the training data will not fit in memory! Therefore, decision tree construction becomes inefficient due to swapping of the training tuples in and out of main and cache memories. More scalable approaches, capable of handling training data that are too large to fit in memory, are required. Earlier strategies to "save space" included discretizing continuous-valued attributes and sampling data at each node. These techniques, however, still assume that the training set can fit in memory.

Several scalable decision tree induction methods have been introduced in recent studies. Rain Forest, 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. Rain Forest also has techniques, however, for handling the case where the AVC-group does not fit in memory. Therefore, the method has high scalability for decision tree induction in very large data sets.



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

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

	buys_computer	
student	yes	no
yes	6	1
no	3	4

	buys_computer	
credit_ratting	yes	no
fair	6	2
excellent	3	3

Fig: AVC Sets for dataset

Example for Decision Tree construction and Classification Rules:

Construct Decision Tree for following dataset,

Age	income	Student	credit_rating	buys_computer
youth	high	No	fair	No
youth	high	No	excellent	No
middle_aged	high	No	fair	Yes
senior	medium	No	fair	Yes
senior	low	Yes	fair	Yes
senior	low	Yes	excellent	No
middle_aged	low	Yes	excellent	Yes
youth	medium	No	fair	No
youth	low	Yes	fair	Yes
senior	medium	Yes	fair	Yes
youth	medium	Yes	excellent	Yes
middle_aged	medium	No	excellent	Yes
middle_aged	high	Yes	fair	Yes
senior	medium	No	excellent	No

Solution:

Here the target class is buys_computer and values are yes, no. By using ID3 algorithm, we are constructing decision tree.

For ID3 Algorithm we have calculate Information gain attribute selection measure.

CLASS	P	buys_computer (yes)	9
	N	buys_computer (no)	5
		TOTAL	14

$$\text{Info}(D) = I(9,5) = -\frac{9}{14}\log_2\frac{9}{14} - \frac{5}{14}\log_2\frac{5}{14} = 0.940$$

Age	P	N	TOTAL	I(P,N)
youth	2	3	5	I(2,3
middle_aged	4	0	4	I(4,0)
senior	3	2	5	I(3,2)

$$I(2,3) = -\frac{2}{5}\log_2\frac{2}{5} - \frac{3}{5}\log_2\frac{3}{5} = 0.970$$

$$I(4,0) = -\frac{4}{4}\log_2\frac{4}{4} - \frac{0}{4}\log_2\frac{0}{4} = 0$$

$$I(3,2) = -\frac{3}{5}\log_2\frac{3}{5} - \frac{2}{5}\log_2\frac{2}{5} = 0.970$$

Age	P	N	TOTAL	I(P,N)	
youth	2	3	5	I(2,3	0.970
middle_aged	4	0	4	I(4,0)	0
senior	3	2	5	I(3,2)	0.970

$$\mathrm{Info}_{Age}(D) = \frac{5}{14} I(2,3) + \frac{4}{14} I(4,0) + \frac{5}{14} I(3,2) = 0.693$$

Gain(Age) = Info(D) – Info$_{Age}$(D)

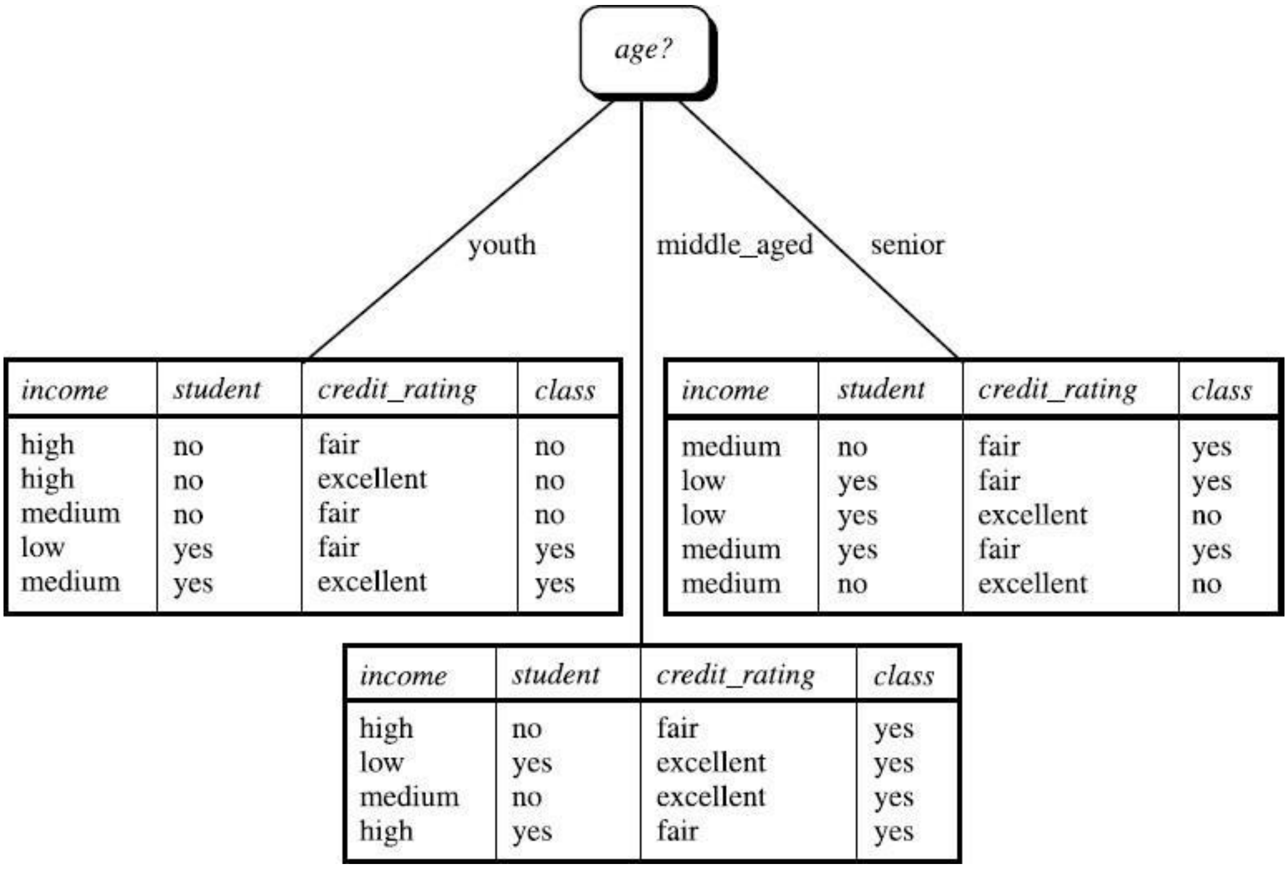
= 0.940 – 0693 = 0.247

Similarly,

Gain(Income) = 0.029 Gain (Student) = 0.151

Gain (credit_rating) = 0.048

Finally, *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

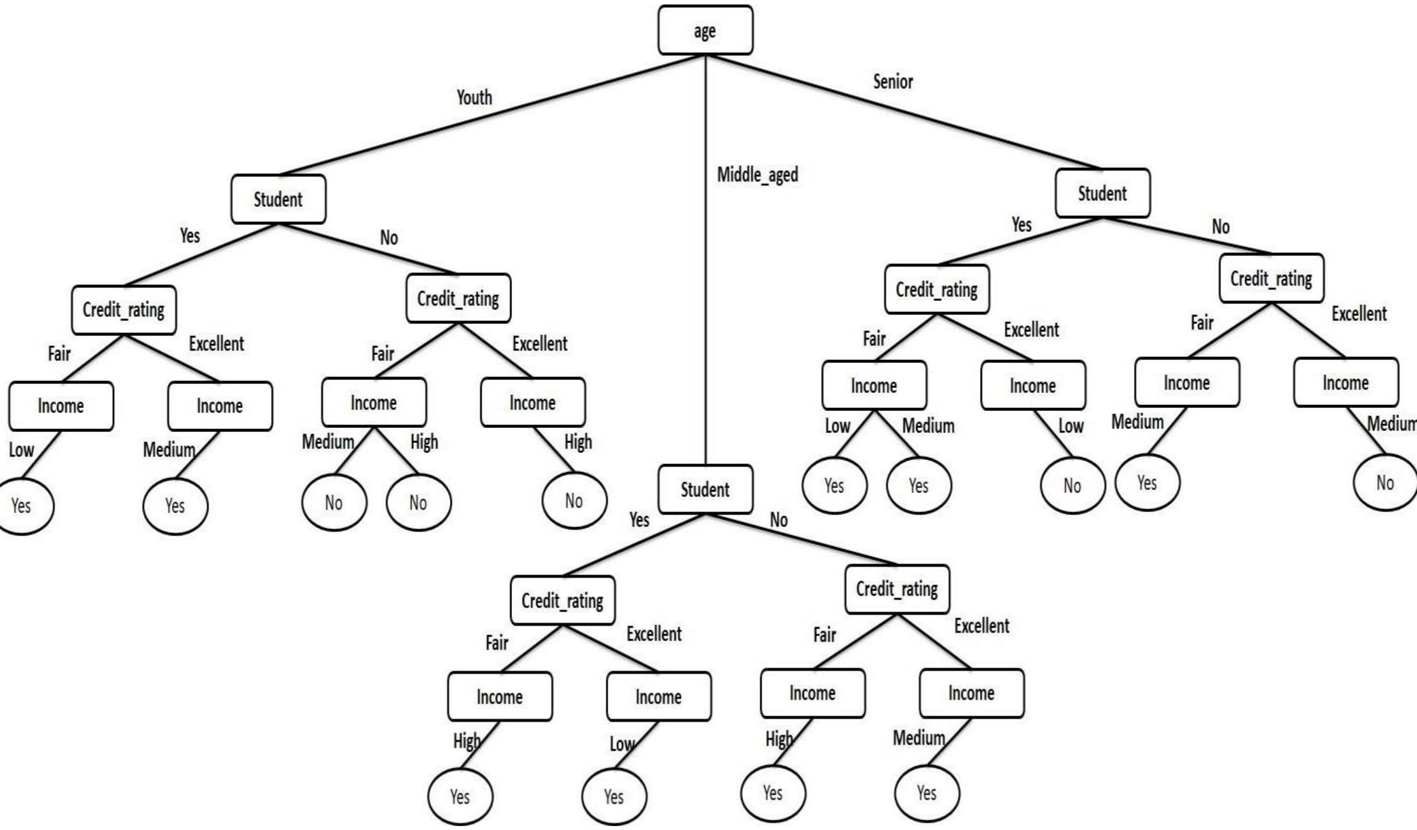


income	*student*	*credit_rating*	*class*
high	no	fair	no
high	no	excellent	no
medium	no	fair	no
low	yes	fair	yes
medium	yes	excellent	yes

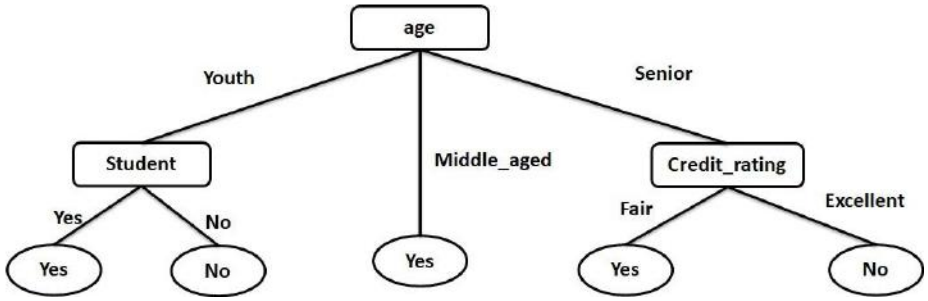
income	*student*	*credit_rating*	*class*
medium	no	fair	yes
low	yes	fair	yes
low	yes	excellent	no
medium	yes	fair	yes
medium	no	excellent	no

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

The Tree after splitting branches is



The Tree after Tree Pruning,



Finally, The Classification Rules are,

- **IF** age=Youth **AND** Student=Yes **THEN** buys_computer=Yes
- **IF** age=Middle_aged **THEN** buys_computer=Yes
- **IF** age=Senior **AND** Credit_rating=Fair **THEN** buys_computer=Yes

➢ 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.

Bayesian classification is based on Bayes' theorem, described next. Studies comparing classification algorithms have found a simple Bayesian classifier known as the ***naıveBayesian classifier*** to be comparable in performance with decision tree and selected neu- ral network classifiers. Bayesian classifiers have also exhibited high accuracy and speed when applied to large databases.

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*. It is made to simplify the computations involved and, in this sense, is considered "na¨ıve."

Section 8.3.1 reviews basic probability notation and Bayes' theorem. In Section 8.3.2 you will learn how to do na¨ıve Bayesian classification.

➢ 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 ***X*** be a data tuple. In Bayesian terms, ***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 ***X*** belongs to a specified class *C*. For classification problems, we want to determine *P(H* ***X****)*, the probability that the hypothesis *H* holds given the "evidence" or observed data tuple ***X***. In other words, we are looking for the probability that tuple ***X*** belongs to class *C*, given that we know the attribute description of ***X***.

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

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

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

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

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

Now that we have that out of the way, in the next section, we will look at how Bayes' theorem is used in the na¨ıve Bayesian classifier.

Naïve 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, $X = (x_1, x_2, \ldots, x_n)$, depicting n measurements made on the tuple from n attributes, respectively, $A_1, A_2, \ldots, A_n$.
2. Suppose that there are m classes, $C_1, C_2, \ldots, C_m$. Given a tuple, 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 C_i if and only if

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

Thus, we maximize $P(C_i|X)$. The class C_i for which $P(C_i|X)$ 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 $P(X|C_i)P(C_i)$ 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, $P(C_1) = P(C_2) = \cdots = P(C_m)$, and we would therefore maximize $P(X|C_i)$. Otherwise, we maximize $P(X|C_i)P(C_i)$. Note that the class prior probabilities may be estimated by $P(C_i) = |C_{i,D}| / |D|$, where $|C_{i,D}|$ is the number of training tuples of class C_i in D.

4. Given data sets with many attributes, it would be extremely computationally expensive to compute $P(X|C_i)$. To reduce computation in evaluating $P(X|C_i)$, 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,

$$P(X|C_i) = \prod_{k=1}^{n} P(x_k|C_i) \tag{8.12}$$

$$= P(x_1|C_i) \times P(x_2|C_i) \times \cdots \times P(x_n|C_i).$$

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

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

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

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

so that

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

These equations may appear daunting, but hold on! We need to compute μ_{C_i} and σ_{C_i}, which are the mean (i.e., average) and standard deviation, respectively, of the values of attribute A_k for training tuples of class C_i. We then plug these two quantities into Eq. (8.13), together with x_k, to estimate $P(x_k|C_i)$.

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)*. For a quick review of mean and standard deviation calculations, please see Section 2.2.

5. To predict the class label of X, $P(X|C_i)P(C_i)$ is evaluated for each class C_i. The classifier predicts that the class label of tuple X is the class C_i if and only if

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

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

"How effective are Bayesian classifiers?" Various empirical studies of this classifier in comparison to decision tree and neural network classifiers have found it to be comparable in some domains. In theory, Bayesian classifiers have the minimum error rate in comparison to all other classifiers. However, in practice this is not always the case, owing to inaccuracies in the assumptions made for its use, such as class-conditional independence, and the lack of available probability data.

Bayesian classifiers are also useful in that they provide a theoretical justification for other classifiers that do not explicitly use Bayes' theorem. For example, under certain assumptions, it can be shown that many neural network and curve-fitting algorithms output the *maximum posteriori* hypothesis, as does the na¨ıve Bayesian classifier.

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 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:

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

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

Using these probabilities, we obtain

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

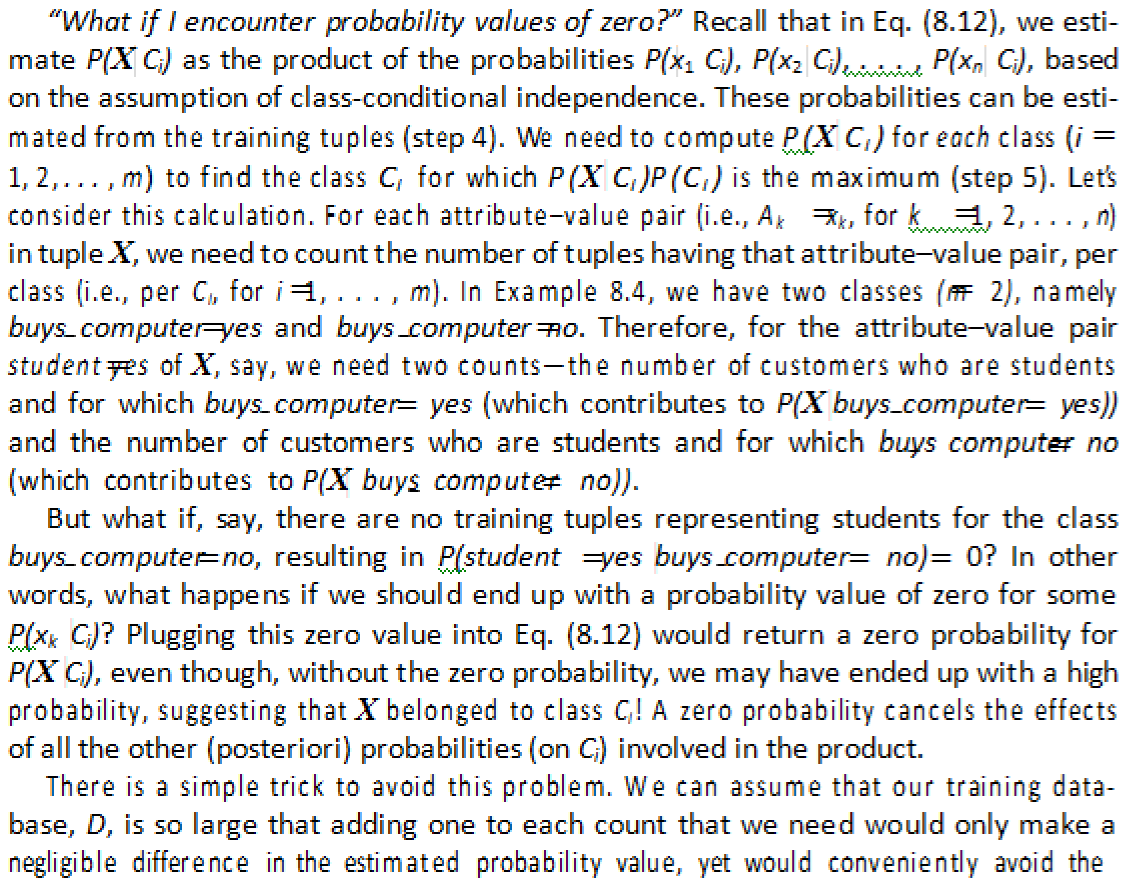
Similarly,

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

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 if I encounter probability values of zero?" Recall that in Eq. (8.12), we estimate $P(X|C_i)$ as the product of the probabilities $P(x_1|C_i)$, $P(x_2|C_i)$, ..., $P(x_n|C_i)$, based on the assumption of class-conditional independence. These probabilities can be estimated from the training tuples (step 4). We need to compute $P(X|C_i)$ for *each* class ($i = 1, 2, \ldots, m$) to find the class C_i for which $P(X|C_i)P(C_i)$ is the maximum (step 5). Let's consider this calculation. For each attribute–value pair (i.e., $A_k = x_k$, for $k = 1, 2, \ldots, n$) in tuple X, we need to count the number of tuples having that attribute–value pair, per class (i.e., per C_i, for $i = 1, \ldots, m$). In Example 8.4, we have two classes ($m = 2$), namely *buys_computer = yes* and *buys_computer = no*. Therefore, for the attribute–value pair *student = yes* of X, say, we need two counts—the number of customers who are students and for which *buys_computer = yes* (which contributes to $P(X|buys\_computer = yes)$) and the number of customers who are students and for which *buys_computer = no* (which contributes to $P(X|buys\_computer = no)$).

But what if, say, there are no training tuples representing students for the class *buys_computer = no*, resulting in $P(student = yes|buys\_computer = no) = 0$? In other words, what happens if we should end up with a probability value of zero for some $P(x_k|C_i)$? Plugging this zero value into Eq. (8.12) would return a zero probability for $P(X|C_i)$, even though, without the zero probability, we may have ended up with a high probability, suggesting that X belonged to class C_i! A zero probability cancels the effects of all the other (posteriori) probabilities (on C_i) involved in the product.

There is a simple trick to avoid this problem. We can assume that our training database, D, is so large that adding one to each count that we need would only make a negligible difference in the estimated probability value, yet would conveniently avoid the

case of probability values of zero. This technique for probability estimation is known as the **Laplacian correction** or **Laplace estimator**, named after Pierre Laplace, a French mathematician who lived from 1749 to 1827. If we have, say, *q* counts to which we each add one, then we must remember to add *q* to the corresponding denominator used in the probability calculation.

➢ Rule-Based Classification

In this section, we look at rule-based classifiers, where the learned model is represented as a set of IF-THEN rules. We first examine how such rules are used for classification (Section 8.4.1). We then study ways in which they can be generated, either from a deci- sion tree (Section 8.4.2) or directly from the training data using a *sequential covering algorithm* (Section 8.4.3).

Using IF-THEN Rules for Classification

Rules are a good way of representing information or bits of knowledge. A **rule-based classifier** uses a set of IF-THEN rules for classification. An **IF-THEN** rule is an expres- sion 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 class-labeled data set, *D*, let n_{covers} be the number of tuples covered by *R*; $n_{correct}$ be the number of tuples correctly classified by *R*; and *D* 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 (i.e., their attribute values hold true for the rule's antecedent). For a rule's accuracy, we look at the tuples that it covers and see what percentage of them the rule can correctly classify.

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

Let's see how we can use rule-based classification to predict the class label of a given tuple,

X. 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. Alternatively, they may be sorted based on the misclassification cost per class. Within each class, the rules are not ordered—they don't have to be because they all predict the same class (and so there can be no class conflict!).
With **rule-based ordering**, the rules are organized into one long priority list, accord-ing 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**. With rule ordering, the triggering rule that appears earliest in the list has the highest priority, and so it gets to fire its class prediction. Any other rule that satisfies ***X*** is ignored. Most rule-based classification systems use a class-based rule-ordering strategy.
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***. The condition in the default rule is empty. In this way, the rule fires when no other rule is satisfied.
In the following sections, we examine how to build a rule-based classifier.

Rule Extraction from a Decision Tree

In Section 8.2, we learned how to build a decision tree classifier from a set of training data. Decision tree classifiers are a popular method of classification—it is easy to under- stand 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 rule- based 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).

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*.

Since we end up with one rule per leaf, the set of extracted rules is not much simpler than the corresponding decision tree! The extracted rules may be even more difficult to interpret than the original trees in some cases. As an example, Figure 8.7 showed decision trees that suffer from subtree repetition and replication. The resulting set of rules extracted can be large and difficult to follow, because some of the attribute tests may be irrelevant or redundant. So, the plot thickens. Although it is easy to extract rules from a decision tree, we may need to do some more work by pruning the resulting rule set.

"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 compen-

sate for the bias, resulting in a pessimistic estimate. In addition, any rule that does not contribute to the overall accuracy of the entire rule set can also be pruned.

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. C4.5 orders the class rule sets so as to minimize the number of *false-positive errors* (i.e., where a rule predicts a class, *C*, but the actual class is not *C*). The class rule set with the least number of false positives is examined first. Once pruning is complete, a final check is remove any duplicates. When choosing a default class, C4.5 does not choose the majority class, because this class will likely have many rules for its tuples. Instead, it selects the class that contains the most training tuples that were not covered by any rule.

➢ Classification by Backpropagation:

➢ Classification by Backpropagation

"*What is backpropagation?*" Backpropagation is a neural network learning algorithm. The neural networks field was originally kindled by psychologists and neurobiologists who sought to develop and test computational analogs of neurons. Roughly speaking, a neural network is a set of connected input/output units in which each connection has a weight associated with it. During the learning phase, the network learns by adjusting the weights so as to be able to predict the correct class label of the input tuples. Neural network learning is also referred to as **connectionist learning** due to the connections between units.

Neural networks involve long training times and are therefore more suitable for appli- cations where this is feasible. They require a number of parameters that are typically best determined empirically such as the network topology or "structure." Neural net- works have been criticized for their poor interpretability. For example, it is difficult for humans to interpret the symbolic meaning behind the learned weights and of "hidden units" in the network. These features initially made neural networks less desirable for data mining.

Advantages of neural networks, however, include their high tolerance of noisy data as well as their ability to classify patterns on which they have not been trained. They can be used when you may have little knowledge of the relationships between attributes and classes. They are well suited for continuous-valued inputs and outputs, unlike most decision tree algorithms. They have been successful on a wide array of real-world data, including handwritten character recognition, pathology and laboratory medicine, and training a computer to pronounce English text. Neural network algorithms are inher- ently parallel; parallelization techniques can be used to speed up the computation process. In addition, several techniques have been recently developed for rule extrac- tion from trained neural networks. These factors contribute to the usefulness of neural networks for classification and numeric prediction in data mining.

There are many different kinds of neural networks and neural network algorithms. The most popular neural network algorithm is *backpropagation*, which gained repute in the 1980s. In Section 9.2.1 you will learn about multilayer feed-forward net-works, the type of neural network on which the backpropagation algorithm performs. Section 9.2.2

discusses defining a network topology. The backpropagation algorithm is described in Section 9.2.3. Rule extraction from trained neural networks is discussed in Section 9.2.4.

A Multilayer Feed-Forward Neural Network

The backpropagation algorithm performs learning on a *multilayer feed-forward* neural network. It iteratively learns a set of weights for prediction of the class label of tuples.A **multilayer feed-forward** neural network consists of an *input layer*, one or more *hidden layers*, and an *output layer*. An example of a multilayer feed-forward network is shownin Figure 9.2.

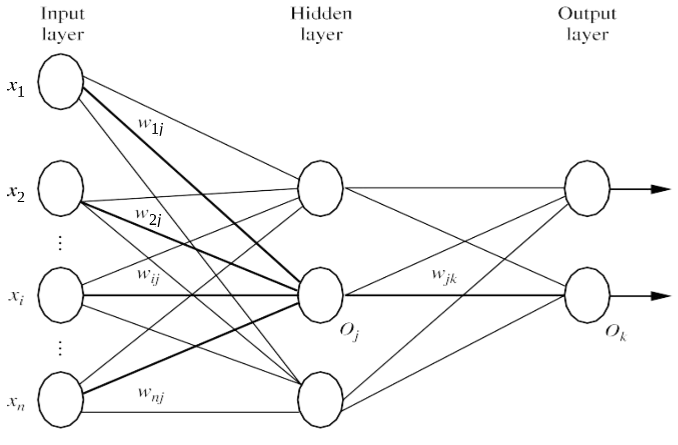


Figure 9.2 Multilayer feed-forward neural network.

Each layer is made up of units. The inputs to the network correspond to the attributes measured for each training tuple. The inputs are fed simultaneously into the units making up the **input layer**. These inputs pass through the input layer and are then weighted and fed simultaneously to a second layer of "neuronlike" units, known as a **hidden layer**. The outputs of the hidden layer units can be input to another hidden layer, and so on. The number of hidden layers is arbitrary, although in practice, usually only one is used. The weighted outputs of the last hidden layer are input to units making up the **output layer**, which emits the network's prediction for given tuples. The units in the input layer are called **input units**. The units in the hidden layers and output layer are sometimes referred to as **neurodes**, due to their symbolic biological basis, or as **output units**. The multilayer neural network shown in Figure 9.2 has two layers of output units. Therefore, we say that it is a **two-layer** neural network. (The input layer is not counted because it serves only to pass the input values to the next layer.) Similarly, a network containing two hidden layers is called a *three-layer* neural network, and so on. It is a feed-forward network since none of the weights cycles back to an input unit or to a previous layer's output unit. It is **fully connected** in that each unit provides input to each unit in the next forward layer.

Each output unit takes, as input, a weighted sum of the outputs from units in the previous layer (see Figure 9.4 later). It applies a nonlinear (activation) function to the weighted input. Multilayer feed-forward neural networks are able to model the class pre-diction as a nonlinear combination of the inputs. From a statistical point of view, theyperform nonlinear regression. *Multilayer feed-forward networks, given enough hidden units and enough training samples, can closely approximate any function.*

Defining a Network Topology

"*How can I design the neural network's topology?*" Before training can begin, the user must decide on the network topology by specifying the number of units in the input layer, the number of hidden layers (if more than one), the number of units in each hidden layer, and the number of units in the output layer.

Normalizing the input values for each attribute measured in the training tuples will help speed up the learning phase. Typically, input values are normalized so as to fall between 0.0 and 1.0. Discrete-valued attributes may be encoded such that there is one input unit per domain value. For example, if an attribute *A* has three possible or known values, namely $\{a_0, a_1, a_2\}$, then we may assign three input units to represent *A*. That is, we may have, say, I_0, I_1, I_2 as input units. Each unit is initialized to 0. If $A = a_0$, then I_0 is set to 1 and the rest are 0. If A a_1, then I_1 is set to 1 and the rest are 0, and so on.

Neural networks can be used for both classification (to predict the class label of a given tuple) and numeric prediction (to predict a continuous-valued output). For clas- sification, one output unit may be used to represent two classes (where the value 1 represents one class, and the value 0 represents the other). If there are more than two classes, then one output unit per class is used. (See Section 9.7.1 for more strategies on multiclass classification.)

There are no clear rules as to the "best" number of hidden layer units. Network design is a trial-and-error process and may affect the accuracy of the resulting trained net- work. The initial values of the weights may also affect the resulting accuracy. Once a network has been trained and its accuracy is not considered acceptable, it is common to repeat the training process with a different network topology or a different set of initial weights. Cross-validation techniques for accuracy estimation (described in Chapter 8) can be used to help decide when an acceptable network has been found. A number of automated techniques have been proposed that search for a "good" network structure. These typically use a hill-climbing approach that starts with an initial structure that is selectively modified.

Backpropagation

"*How does backpropagation work?*" Backpropagation learns by iteratively processing a data set of training tuples, comparing the network's prediction for each tuple with the actual known *target* value. The target value may be the known class label of the training tuple (for classification problems) or a continuous value (for numeric prediction). For each training tuple, the weights are modified so as to minimize the mean-squared error between the network's prediction and the actual target value. These modifications are made in the "backwards" direction (i.e., from the output layer) through each hidden layer down to the first hidden layer (hence the name *backpropagation*). Although it is not guaranteed, in general the weights will eventually converge, and the learning process stops. The algorithm is summarized in Figure 9.3. The steps involved are expressed in terms of inputs, outputs, and errors, and may seem awkward if this is your first look at neural network learning. However, once you become familiar with the process, you will see that each step is inherently simple. The steps are described next.

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

Input:

D, a data set consisting of the training tuples and their associated target values;

l, the learning rate;

network, a multilayer feed-forward network.

Output: A trained neural network.

Method:

```
(1)  Initialize all weights and biases in network;
(2)  while terminating condition is not satisfied {
(3)      for each training tuple X in D {
(4)          // Propagate the inputs forward:
(5)          for each input layer unit j {
(6)              $O_j = I_j$; // output of an input unit is its actual input value
(7)          for each hidden or output layer unit j {
(8)              $I_j = \sum_i w_{ij} O_i + \theta_j$; //compute the net input of unit j with respect to
                 the previous layer, i
(9)              $O_j = \frac{1}{1+e^{-I_j}}$; } // compute the output of each unit j
(10)         // Backpropagate the errors:
(11)         for each unit j in the output layer
(12)             $Err_j = O_j(1 - O_j)(T_j - O_j)$; // compute the error
(13)         for each unit j in the hidden layers, from the last to the first hidden layer
(14)             $Err_j = O_j(1 - O_j) \sum_k Err_k w_{jk}$; // compute the error with respect tothe
next higher layer, k
(15)         for each weight $w_{ij}$ in network {
(16)             $\Delta w_{ij} = (l) Err_j O_i$; // weight increment
(17)             $w_{ij} = w_{ij} + \Delta w_{ij}$; } // weight update
(18)         for each bias $\theta_j$ in network {
(19)             $\Delta\theta_j = (l) Err_j$; // bias increment
(20)             $\theta_j = \theta_j + \Delta\theta_j$; } // bias update
(21)         } }
```

Figure 9.3 Backpropagation algorithm.

Initialize the weights: The weights in the network are initialized to small random num- bers (e.g., ranging from 1.0 to 1.0, or 0.5 to 0.5). Each unit has a *bias* associated with it, as explained later. The biases are similarly initialized to small random numbers.

Each training tuple, ***X***, is processed by the following steps.

Propagate the inputs forward: First, the training tuple is fed to the network's input layer. The inputs pass through the input units, unchanged. That is, for an input unit, *j*,

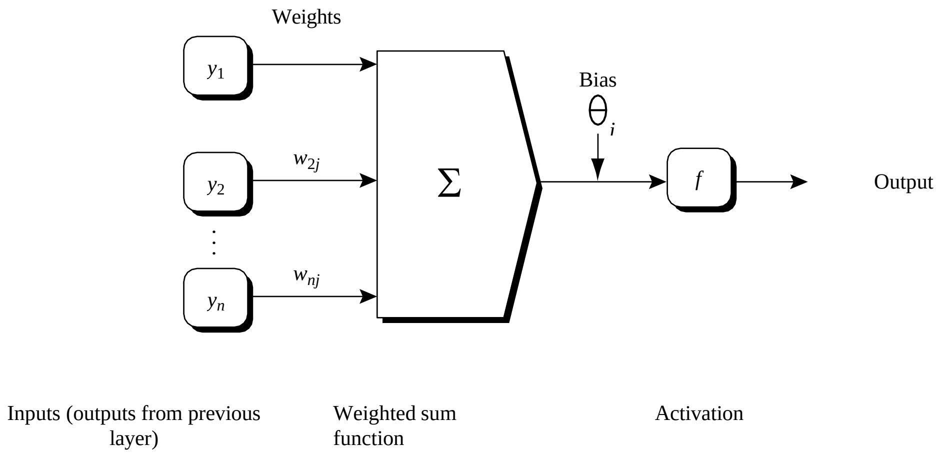


Figure 9.4 Hidden or output layer unit *j*: The inputs to unit *j* are outputs from the previous layer. These are multiplied by their corresponding weights to form a weighted sum, which is added to the bias associated with unit *j*. A nonlinear activation function is applied to the net input. (Forease of explanation, the inputs to unit *j* are labeled $y_1, y_2, \ldots, y_n$. If unit *j* were in the first hidden layer, then these inputs would correspond to the input tuple $(x_1, x_2, \ldots, x_n)$.)

its output, O_j, is equal to its input value, I_j. Next, the net input and output of each unit in the hidden and output layers are computed. The net input to a unit in the hidden or output layers is computed as a linear combination of its inputs. To help illustrate this point, a hidden layer or output layer unit is shown in Figure 9.4. Each such unit has a number of inputs to it that are, in fact, the outputs of the units connected to it in the previous layer. Each connection has a weight. To compute the net input to the unit, each input connected to the unit is multiplied by its corresponding weight, and this is summed. Given a unit, *j* in a hidden or output layer, the net input, I_j, to unit *j* is

$$I_j = \sum_i w_{ij} O_i + \theta_j, \qquad (9.4)$$

where w_{ij} is the weight of the connection from unit *i* in the previous layer to unit *j*; O_i is the output of unit *i* from the previous layer; and θ_j is the **bias** of the unit. The bias acts as a threshold in that it serves to vary the activity of the unit.

Each unit in the hidden and output layers takes its net input and then applies an **activation** function to it, as illustrated in Figure 9.4. The function symbolizes the activation of the neuron represented by the unit. The **logistic**, or **sigmoid**, function is used. Given the net input I_j to unit *j*, then O_j, the output of unit *j*, is computed as

$$O_j = \frac{1}{1+e^{-I_j}}. \qquad (9.5)$$

This function is also referred to as a *squashing function*, because it maps a large input domain onto the smaller range of 0 to 1. The logistic function is nonlinear and differentiable, allowing the backpropagation algorithm to model classification problems that are linearly inseparable.

We compute the output values, O_j, for each hidden layer, up to and including the output layer, which gives the network's prediction. In practice, it is a good idea to cache (i.e., save) the intermediate output values at each unit as they are required again later when backpropagating the error. This trick can substantially reduce the amount of computation required.

Backpropagate the error: The error is propagated backward by updating the weights and biases to reflect the error of the network's prediction. For a unit *j* in the output layer, the error Err_j is computed by

$$Err_j = O_j(1 - O_j)(T_j - O_j), \qquad (9.6)$$

where O_j is the actual output of unit *j*, and T_j is the known target value of the given training tuple. Note that $O_j(1\ O_j)$ is the derivative of the logistic function.
To compute the error of a hidden layer unit *j*, the weighted sum of the errors of the units connected to unit *j* in the next layer are considered. The error of a hidden layer unit *j* is

$$Err_j = O_j(1 - O_j) \sum_k Err_k w_{jk}, \qquad (9.7)$$

where w_{jk} is the weight of the connection from unit *j* to a unit *k* in the next higher layer, and Err_k is the error of unit *k*.
The weights and biases are updated to reflect the propagated errors. Weights are updated by the following equations, where Δw_{ij} is the change in weight w_{ij}:

$$\Delta w_{ij} = (l)Err_j O_i. \qquad (9.8)$$

$$w_{ij} = w_{ij} + \Delta w_{ij}. \qquad (9.9)$$

"*What is* l *in Eq. (9.8)?*" The variable *l* is the **learning rate**, a constant typically having a value between 0.0 and 1.0. Backpropagation learns using a gradient descent method to search for a set of weights that fits the training data so as to minimize the mean- squared distance between the network's class prediction and the known target value of the tuples.[1] The learning rate helps avoid getting stuck at a local minimum in decision space (i.e., where the weights appear to converge, but are not the optimum solution) and encourages finding the global minimum. If the learning rate is too small, then learning will occur at a very slow pace. If the learning rate is too large, then oscillation between inadequate solutions may occur. A rule of thumb is to set the learning rate to 1/*t*, where
t is the number of iterations through the training set so far.
Biases are updated by the following equations, where $\Delta\theta_j$ is the change in bias θ_j:

$$\Delta\vartheta_j = (l)Err_j. \qquad (9.10)$$

$$\vartheta_j = \vartheta_j + \Delta\vartheta_j. \qquad (9.11)$$

Note that here we are updating the weights and biases after the presentation of each tuple. This is referred to as **case updating**. Alternatively, the weight and bias increments could be accumulated in variables, so that the weights and biases are updated after all the tuples in the training set have been presented. This latter strategy is called **epoch updating**, where one iteration through the training set is an **epoch**. In the ory, the mathematical derivation of backpropagation employs epoch updating, yet in practice, case updating is more common because it tends to yield more accurate results.

Terminating condition: Training stops when

All Δw_{ij} in the previous epoch are so small as to be below some specified threshold, or

The percentage of tuples misclassified in the previous epoch is below some threshold, or

A prespecified number of epochs has expired.

In practice, several hundreds of thousands of epochs may be required before the weights will converge.
"*How efficient is backpropagation?*" The computational efficiency depends on the time spent training the network. Given D tuples and w weights, each epoch requires $O(|D| \times w)$ time. However, in the worst-case scenario, the number of epochs can be exponential in n, the number of inputs. In practice, the time required for the networks to converge is highly variable. A number of techniques exist that help speed up the training time. For example, a technique known as *simulated annealing* can be used, which also ensures convergence to a global optimum.

Example 9.1 Sample calculations for learning by the backpropagation algorithm. Figure 9.5 shows a multilayer feed-forward neural network. Let the learning rate be 0.9. The initial weight and bias values of the network are given in Table 9.1, along with the first training tuple, $\boldsymbol{X} = (1, 0, 1)$, with a class label of 1.

This example shows the calculations for backpropagation, given the first training tuple, ***X***. The tuple is fed into the network, and the net input and output of each unit is computed. These values are shown in Table 9.2. The error of each unit is computed and propagated backward. The error values are shown in Table 9.3. The weight and bias updates are shown in Table 9.4. ■

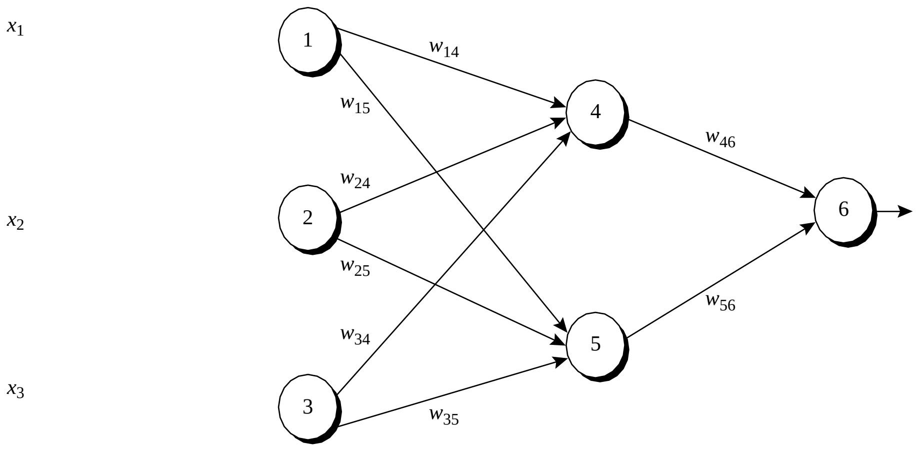


Figure 9.5 Example of a multilayer feed-forward neural network.

Table 9.1 Initial Input, Weight, and Bias Values

x_1	x_2	x_3	w_{14}	w_{15}	w_{24}	w_{25}	w_{34}	w_{35}	w_{46}	w_{56}	ϑ_4	ϑ_5	ϑ_6
1	0	1	0.2	- 0.3	0.4	0.1	- 0.5	0.2	- 0.3	- 0.2	- 0.4	0.2	0.1

Table 9.2 Net Input and Output Calculations

Unit, j	*Net Input, I_j*	*Output, O_j*
4	0.2 + 0 - 0.5 - 0.4 = -0.7	$1/(1 + e^{0.7}) = 0.332$
5	- 0.3 + 0 + 0.2 + 0.2 = 0.1	$1/(1 + e^{-0.1}) = 0.525$
6	(-0.3)(0.332) - (0.2)(0.525) + 0.1 = -0.105	$1/(1 + e^{0.105}) = 0.474$

Table 9.3 Calculation of the Error at Each Node

Unit, j	*Err_j*
6	(0.474)(1 - 0.474)(1 - 0.474) = 0.1311
5	(0.525)(1 - 0.525)(0.1311)(- 0.2) = -0.0065
4	(0.332)(1 - 0.332)(0.1311)(- 0.3) = -0.0087

Table 9.4 Calculations for Weight and Bias Updating

Weight or Bias	New Value
w_{46}	$-0.3 + (0.9)(0.1311)(0.332) = -0.261$
w_{56}	$-0.2 + (0.9)(0.1311)(0.525) = -0.138$
w_{14}	$0.2 + (0.9)(-0.0087)(1) = 0.192$
w_{15}	$-0.3 + (0.9)(-0.0065)(1) = -0.306$
w_{24}	$0.4 + (0.9)(-0.0087)(0) = 0.4$
w_{25}	$0.1 + (0.9)(-0.0065)(0) = 0.1$
w_{34}	$-0.5 + (0.9)(-0.0087)(1) = -0.508$
w_{35}	$0.2 + (0.9)(-0.0065)(1) = 0.194$
θ_6	$0.1 + (0.9)(0.1311) = 0.218$
θ_5	$0.2 + (0.9)(-0.0065) = 0.194$
θ_4	$-0.4 + (0.9)(-0.0087) = -0.408$

"*How can we classify an unknown tuple using a trained network?*" To classify an unknown tuple, ***X***, the tuple is input to the trained network, and the net input and output of each unit are computed. (There is no need for computation and/or backpropagation of the error.) If there is one output node per class, then the output node with the highest value determines the predicted class label for ***X***. If there is only one output node, then output values greater than or equal to 0.5 may be considered as belonging to the positive class, while values less than 0.5 may be considered negative.
Several variations and alternatives to the backpropagation algorithm have been proposed for classification in neural networks. These may involve the dynamic adjustment of the network topology and of the learning rate or other parameters, or the use of different error functions.

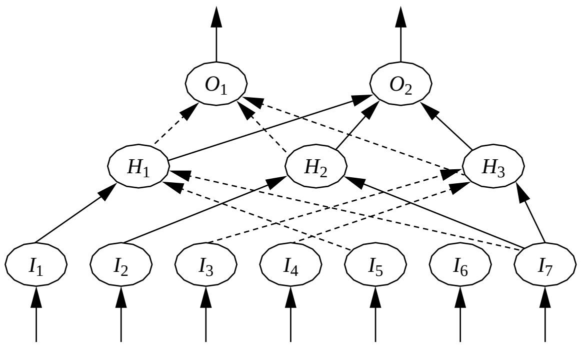
Inside the Black Box: Backpropagation and Interpretability

"Neural networks are like a black box. How can I 'understand' what the backpropagation network has learned?" A major disadvantage of neural networks lies in their knowledge representation. Acquired knowledge in the form of a network of units connected by weighted links is difficult for humans to interpret. This factor has motivated research in extracting the knowledge embedded in trained neural networks and in representing that knowledge symbolically. Methods include extracting rules from networks and sensitivity analysis.

Various algorithms for rule extraction have been proposed. The methods typically impose restrictions regarding procedures used in training the given neural network, the network topology, and the discretization of input values.
Fully connected networks are difficult to articulate. Hence, often the first step in extracting rules from neural networks is **network pruning**. This consists of simplifying the network structure by removing weighted links that have the least effect on the trained network. For example, a weighted link may be deleted if such removal

does not result ina decrease in the classification accuracy of the network.

Once the trained network has been pruned, some approaches will then perform link, unit, or activation value clustering. In one method, for example, clustering is used to find the set of common activation values for each hidden unit in a given trained two- layer neural network (Figure 9.6). The combinations of these activation values for eachhidden unit are analyzed. Rules are derived relating combinations of activation values



Identify sets of common activation values for each hidden node, H_i:
for H_1: (–1,0,1)
for H_2: (0,1)
for H_3: (–1,0.24,1)

Derive rules relating common activation values with output nodes, O_j:
IF ($H_2=0$ AND $H_3=-1$) OR
($H_1=-1$ AND $H_2=1$ AND $H_3=-1$) OR
($H_1=-1$ AND $H_2=0$ AND $H_3=0.24$)
THEN $O_1=1$, $O_2=0$
ELSE $O_1=0$, $O_2=1$

Derive rules relating input nodes, I_j, to output nodes, O_j:
IF ($I_2=0$ AND $I_7=0$) THEN $H_2=0$
IF ($I_4=1$ AND $I_6=1$) THEN $H_3=-1$
IF ($I_5=0$) THEN $H_3=-1$

Obtain rules relating inputs and output classes:
IF ($I_2=0$ AND $I_7=0$ AND $I_4=1$ AND $I_6=1$) THEN class = 1
IF ($I_2=0$ AND $I_7=0$ AND $I_5=0$) THEN class = 1

Figure 9.6 Rules can be extracted from training neural networks. *Source:* Adapted from Lu, Setiono, andLiu [LSL95].

with corresponding output unit values. Similarly, the sets of input values and activation values are studied to derive rules describing the relationship between the input layer and the hidden "layer units"? Finally, the two sets of rules may be combined to form IF-THEN rules. Other algorithms may derive rules of other forms, including *M*-of-*N* rules (where *M* out of a given *N* conditions in the rule antecedent must be true for the rule consequent to be applied), decision trees with *M*-of-*N* tests, fuzzy rules, and finite automata.

Sensitivity analysis is used to assess the impact that a given input variable has on a network output. The input to the variable is varied while the remaining input variables are fixed at some value. Meanwhile, changes in the network output are monitored. The knowledge gained from this analysis form can be represented in rules such as "*IF X decreases* 5*% THEN Y increases* 8*%*."

➢ Support Vector Machines

In this section, we study **support vector machines (SVMs)**, a method for the classifi- cation of both linear and nonlinear data. In a nutshell, an **SVM** is an algorithm that works as follows. It uses a nonlinear mapping to transform the original training data into a higher dimension. Within this new dimension, it searches for the linear opti- mal separating hyperplane (i.e., a "decision boundary" separating the tuples of one class from another). With an appropriate nonlinear mapping to a sufficiently high dimen- sion, data from two classes can always be separated by a hyperplane. The SVM finds this hyperplane using *support vectors* ("essential" training tuples) and *margins* (defined by the support vectors). We will delve more into these new concepts later.

"*I've heard that SVMs have attracted a great deal of attention lately. Why?*" The first paper on support vector machines was presented in 1992 by Vladimir Vapnik and col- leagues Bernhard Boser and Isabelle Guyon, although the groundwork for SVMs has been around since the 1960s (including early work by Vapnik and Alexei Chervonenkis on statistical learning theory). Although the training time of even the fastest SVMs can be extremely slow, they are highly accurate, owing to their ability to model com- plex nonlinear decision boundaries. They are much less prone to overfitting than other methods. The support vectors found also provide a compact description of the learned model. SVMs can be used for numeric prediction as well as classification. They have been applied to a number of areas, including handwritten digit recognition, object recognition, and speaker identification, as well as benchmark time-series prediction tests.

The Case When the Data Are Linearly Separable

To explain the mystery of SVMs, let's first look at the simplest case—a two-class prob-lem where the classes are linearly separable. Let the data set D be given as ($\boldsymbol{X}_1$, y_1), ($\boldsymbol{X}_2$, y_2), . . . , ($\boldsymbol{X}_{|D|}$, $y_{|D|}$), where $\boldsymbol{X}_i$ is the set of training tuples with associated class labels, y_i. Each y_i can take one of two values, either +1 or −1 (i.e., $y_i \in \{+1, -1\}$),

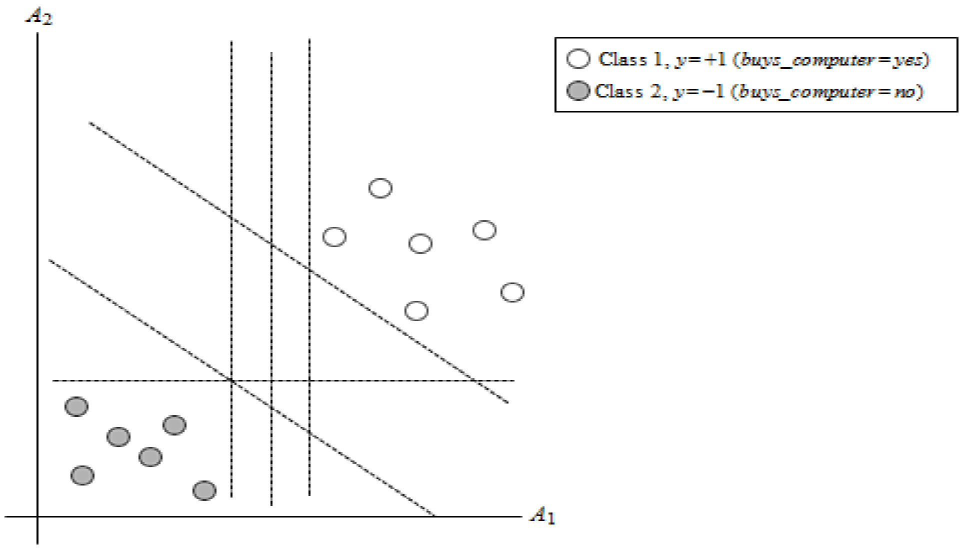
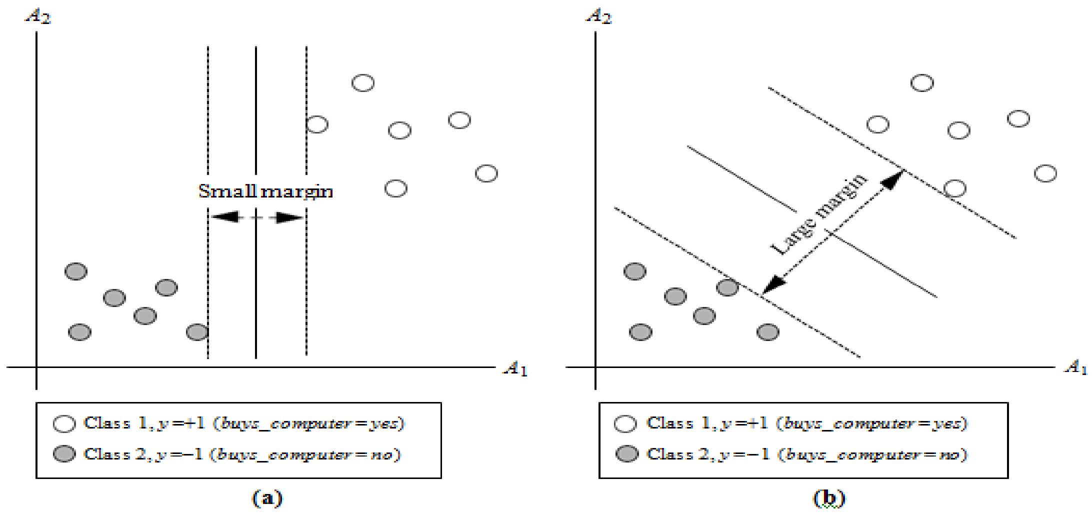

A_2
A_1
○ Class 1, $y = +1$ (*buys_computer* = *yes*)
● Class 2, $y = -1$ (*buys_computer* = *no*)

Figure 9.7 The 2-D training data are linearly separable. There are an infinite number of possible separating hyperplanes or "decision boundaries," some of which are shown here as dashed lines. Which one is best?

corresponding to the classes *buys computer* = *yes* and *buys computer* = *no*, respectively. To aid in visualization, let's consider an example based on two input attributes, A_1 and A_2, as shown in Figure 9.7. From the graph, we see that the 2-D data are **linearly separa- ble** (or "linear," for short), because a straight line can be drawn to separate all the tuples of class 1 from all the tuples of class 1. + −

There are an infinite number of separating lines that could be drawn. We want to find the "best" one, that is, one that (we hope) will have the minimum classification error on previously unseen tuples. How can we find this best line? Note that if our data were 3-D (i.e., with three attributes), we would want to find the best separating *plane*. Generalizing to *n* dimensions, we want to find the best *hyperplane*. We will use "hyperplane" to refer to the decision boundary that we are seeking, regardless of the number of input attributes. So, in other words, how can we find the best hyperplane?

An SVM approaches this problem by searching for the **maximum marginal hyper- plane**. Consider Figure 9.8, which shows two possible separating hyperplanes and their associated margins. Before we get into the definition of margins, let's take an intuitive look at this figure. Both hyperplanes can correctly classify all the given data tuples. Intuitively, however, we expect the hyperplane with the larger margin to be more accurate at classifying future data tuples than the hyperplane with the smaller margin. This is why (during the learning or training phase) the SVM searches for the hyperplane with the largest margin, that is, the *maximum marginal hyperplane* (MMH). The associated margin gives the largest separation between classes.



A_2
Small margin
A_1
○ Class 1, $y=+1$ (buys_computer = yes)
● Class 2, $y=-1$ (buys_computer = no)
(a)
A_2
Large margin
A_1
○ Class 1, $y=+1$ (buys_computer = yes)
● Class 2, $y=-1$ (buys_computer = no)
(b)

Figure 9.8 Here we see just two possible separating hyperplanes and their associated margins. Whichone is better? The one with the larger margin (b) should have greater generalization accuracy.

Getting to an informal definition of **margin**, we can say that the shortest distance from a hyperplane to one side of its margin is equal to the shortest distance from the hyperplane to the other side of its margin, where the "sides" of the margin are parallel to the hyperplane. When dealing with the MMH, this distance is, in fact, the shortest distance from the MMH to the closest training tuple of either class.
A separating hyperplane can be written as

$$\boldsymbol{W} \cdot \boldsymbol{X} + b = 0, \tag{9.12}$$

where $\boldsymbol{W}$ is a weight vector, namely, $\boldsymbol{W}$ $w_1, w_2, \ldots, w_n$; n is the number of attributes; and b is a scalar, often referred to as a bias. To aid in visualization, let's consider two inputattributes, A_1 and A_2, as in Figure 9.8(b). Training tuples are 2-D (e.g., $\boldsymbol{X} = (x_1, x_2)$), where x_1 and x_2 are the values of attributes A_1 and A_2, respectively, for $\boldsymbol{X}$. If we think of b as an additional weight, w_0, we can rewrite Eq. (9.12) as

$$w_0 + w_1x_1 + w_2x_2 = 0. \tag{9.13}$$

Thus, any point that lies above the separating hyperplane satisfies

$$w_0 + w_1x_1 + w_2x_2 > 0. \tag{9.14}$$

Similarly, any point that lies below the separating hyperplane satisfies

$$w_0 + w_1x_1 + w_2x_2 < 0. \tag{9.15}$$

The weights can be adjusted so that the hyperplanes defining the "sides" of the margin can be written as

$$H_1 : w_0 + w_1x_1 + w_2x_2 \geq 1 \quad \text{for } y_i = +1, \tag{9.16}$$

$$H_2 : w_0 + w_1x_1 + w_2x_2 \leq -1 \quad \text{for } y_i = -1. \tag{9.17}$$

That is, any tuple that falls on or above H_1 belongs to class +1, and any tuple that falls on or below H_2 belongs to class 1. Combining the two inequalities of Eqs. (9.16) and (9.17), we get

$$y_i(w_0 + w_1x_1 + w_2x_2) \geq 1, \ \forall i. \tag{9.18}$$

Any training tuples that fall on hyperplanes H_1 or H_2 (i.e., the "sides" defining the margin) satisfy Eq. (9.18) and are called **support vectors**. That is, they are equally close to the (separating) MMH. In Figure 9.9, the support vectors are shown encircled with a thicker border. Essentially, the support vectors are the most difficult tuples to classify and give the most information regarding classification.

From this, we can obtain a formula for the size of the maximal margin. The distance from the separating hyperplane to any point on H_1 is $\frac{1}{||W||}$, where $||W||$ is the Euclidean norm of W, that is, $\sqrt{W \cdot W}$.[2] By definition, this is equal to the distance from any point on H_2 to the separating hyperplane. Therefore, the maximal margin is $\frac{2}{||W||}$.

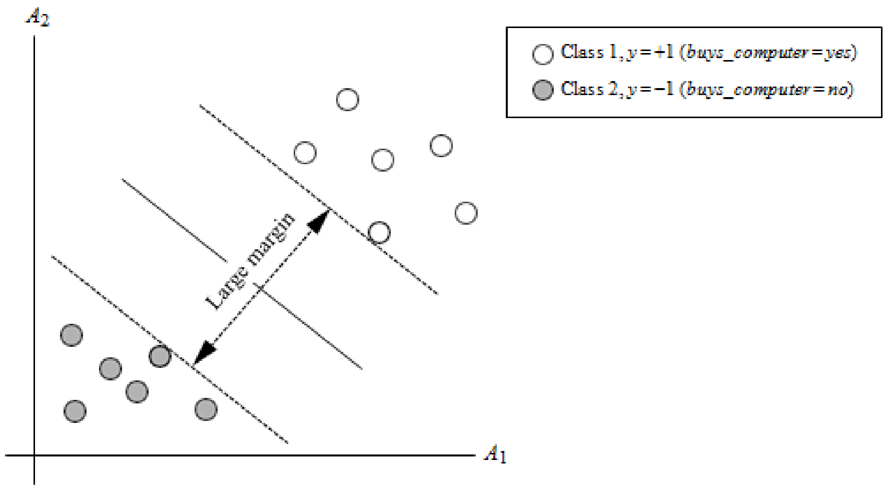

A_2
○ Class 1, $y = +1$ (*buys_computer = yes*)
● Class 2, $y = -1$ (*buys_computer = no*)
Large margin
A_1

Figure 9.9 Support vectors. The SVM finds the maximum separating hyperplane, that is, the one with maximum distance between the nearest training tuples. The support vectors are shown with a thicker border.

“*So, how does an SVM find the MMH and the support vectors?*” Using some “fancy math tricks,” we can rewrite Eq. (9.18) so that it becomes what is known as a constrained (convex) quadratic optimization problem. Such fancy math tricks are beyond the scope of this book. Advanced readers may be interested to note that the tricks involve rewriting Eq. (9.18) using a Lagrangian formulation and then solving for the solution using Karush-Kuhn-Tucker (KKT) conditions. Details can be found in the bibliographic notes at the end of this chapter (Section 9.10).

If the data are small (say, less than 2000 training tuples), any optimization software package for solving constrained convex quadratic problems can then be used to find the support vectors and MMH. For larger data, special and more efficient algorithms for training SVMs can be used instead, the details of which exceed the scope of this book. Once we’ve found the support vectors and MMH (note that the support vectors define the MMH!), we have a trained support vector machine. The MMH is a linear class boundary, and so the corresponding SVM can be used to classify linearly separable data. We refer to such a trained SVM as a *linear SVM*.

“*Once I’ve got a trained support vector machine, how do I use it to classify test (i.e., new) tuples?*” Based on the Lagrangian formulation mentioned before, the MMH can be rewritten as the decision boundary

$$d(\boldsymbol{X}^T) = \sum_{i=1}^{l} y_i \alpha_i \boldsymbol{X}_i \boldsymbol{X}^T + b_0, \qquad (9.19)$$

where y_i is the class label of support vector $\boldsymbol{X}_i$; $\boldsymbol{X}^T$ is a test tuple; α_i and b_0 are numeric parameters that were determined automatically by the optimization or SVM algorithm noted before; and l is the number of support vectors.

Interested readers may note that the α_i are Lagrangian multipliers. For linearly separable data, the support vectors are a subset of the actual training tuples (although there will be a slight twist regarding this when dealing with nonlinearly separable data, as we shall see in the following).

Given a test tuple, $\boldsymbol{X}^T$, we plug it into Eq. (9.19), and then check to see the sign of the result. This tells us on which side of the hyperplane the test tuple falls. If the sign is positive, then $\boldsymbol{X}^T$ falls on or above the MMH, and so the SVM predicts that $\boldsymbol{X}^T$ belongs to class 1 (representing *buys computer yes*, in our case). If the sign is negative, then $\boldsymbol{X}^T$ falls on or below the MMH and the class prediction is 1 (representing *buys computer no*).

Notice that the Lagrangian formulation of our problem (Eq. 9.19) contains a dot product between support vector $\boldsymbol{X}_i$ and test tuple $\boldsymbol{X}^T$. This will prove very useful for finding the MMH and support vectors for the case when the given data are nonlinearly separable, as described further in the next section.

Before we move on to the nonlinear case, there are two more important things to note. The complexity of the learned classifier is characterized by the number of support vectors rather than the dimensionality of the data. Hence, SVMs tend to be less prone to overfitting than some other methods. The support vectors are the essential or critical training tuples—they lie closest to the decision boundary (MMH). If all other training tuples were removed and training were repeated, the same separating hyperplane would be found. Furthermore, the number of support vectors found can be used to compute an (upper) bound on the expected error rate of the SVM classifier, which is independent of the data dimensionality. An SVM with a small number of support vectors can have good generalization, even when the dimensionality of the data is high.

The Case When the Data Are Linearly Inseparable

In Section 9.3.1 we learned about linear SVMs for classifying linearly separable data, but what if the data are not linearly separable, as in Figure 9.10? In such cases, no straight line can be found that would separate the classes. The linear SVMs we studied would not be able to find a feasible solution here. Now what?

The good news is that the approach described for linear SVMs can be extended to create *nonlinear SVMs* for the classification of *linearly inseparable data* (also called *non- linearly separable data*, or *nonlinear data* for short). Such SVMs are capable of finding nonlinear decision boundaries (i.e., nonlinear hypersurfaces) in input space.

"*So*," you may ask, "*how can we extend the linear approach?*" We obtain a nonlinear SVM by extending the approach for linear SVMs as follows. There are two main steps. In the first step, we transform the original input data into a higher dimensional space using a nonlinear mapping. Several common nonlinear mappings can be used in this step, as we will further describe next. Once the data have been transformed into the new higher space, the second step searches for a linear separating hyperplane in the new space. We again end up with a quadratic optimization problem that can be solved using the linear SVM formulation. The maximal marginal hyperplane found in the new space corresponds to a nonlinear separating hypersurface in the original space.

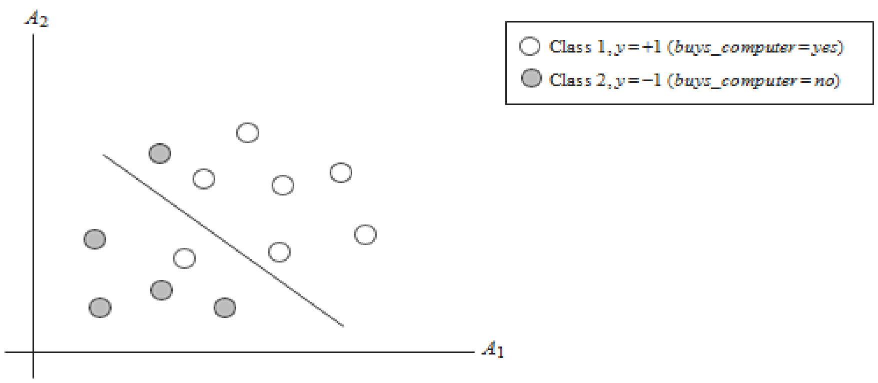


Figure 9.10 A simple 2-D case showing linearly inseparable data. Unlike the linear separable data of Figure 9.7, here it is not possible to draw a straight line to separate the classes. Instead, the decision boundary is nonlinear

Example 9.2 Nonlinear transformation of original input data into a higher dimensional space. Consider the following example. A 3-D input vector $\boldsymbol{X} = (x_1, x_2, x_3)$ is mapped intoa 6-D space, $\boldsymbol{Z}$, using the mappings $\varphi_1(\boldsymbol{X})$ x_1, $\varphi_2(\boldsymbol{X})$ x_2, $\varphi_3(\boldsymbol{X}) = x_3$, $\varphi_4(\boldsymbol{X}) (x_1)^2$, $\varphi_5(\boldsymbol{X}) =$ x_1x_2, and $\varphi_6(\boldsymbol{X})$ x_1x_3. A decision hyperplane in the new space is $d(\boldsymbol{Z})$ $\boldsymbol{WZ}$ b, where $\boldsymbol{W}$ and $\boldsymbol{Z}$ are vectors. This is linear. We solve for $\boldsymbol{W}$ and$b$ and then substitute back so that the linear decision hyperplane in the new ($\boldsymbol{Z}$)space corresponds to a nonlinear second-order polynomial in the original 3-D inputspace:

$$d(\boldsymbol{Z}) = w_1x_1 + w_2x_2 + w_3x_3 + w_4(x_1)^2 + w_5x_1x_2 + w_6x_1x_3 + b$$

$$= w_1z_1 + w_2z_2 + w_3z_3 + w_4z_4 + w_5z_5 + w_6z_6 + b.$$

■

But there are some problems. First, how do we choose the nonlinear mapping to a higher dimensional space? Second, the computation involved will be costly. Refer to Eq. (9.19) for the classification of a test tuple, $\boldsymbol{X^T}$. Given the test tuple, we have to com- pute its dot product with every one of the support vectors.[3] In training, we have to compute a similar dot product several times in order to find the MMH. This is espe- cially expensive. Hence, the dot product computation required is very heavy and costly. We need another trick!

Luckily, we can use another math trick. It so happens that in solving the quadratic optimization problem of the linear SVM (i.e., when searching for a linear SVM in the new higher dimensional space), the training tuples appear only in the form of dot prod- ucts, $\varphi(\boldsymbol{X}_i)\ \varphi(\boldsymbol{X}_j)$, where $\varphi(\boldsymbol{X})$ is simply the nonlinear mapping function applied to transform the training tuples. Instead of computing the dot product on the transformed data tuples, it turns out that it is mathematically equivalent to instead apply a *kernel function*, $K(\boldsymbol{X}_i, \boldsymbol{X}_j)$, to the original input data. That is,

$$K(\boldsymbol{X}_i, \boldsymbol{X}_j) = \varphi(\boldsymbol{X}_i) \cdot \varphi(\boldsymbol{X}_j). \tag{9.20}$$

In other words, everywhere that $\varphi(\boldsymbol{X}_i)\ \varphi(\boldsymbol{X}_j)$ appears in the training algorithm, we can replace it with $K(\boldsymbol{X}_i, \boldsymbol{X}_j)$. In this way, all calculations are made in the original input space, which is of potentially much lower dimensionality! We can safely avoid the mapping—it turns out that we don't even have to know what the mapping is! We will talk more later about what kinds of functions can be used as kernel functions for this problem.

After applying this trick, we can then proceed to find a maximal separating hyper-plane. The procedure is similar to that described in Section 9.3.1, although it involves placing a user-specified upper bound, C, on the Lagrange multipliers, α_i. This upper bound is best determined experimentally.

"What are some of the kernel functions that could be used?" Properties of the kinds of kernel functions that could be used to replace the dot product scenario just described have been studied. Three admissible kernel functions are

Polynomial kernel of degree *h*: $K(X_i, X_j) = (X_i \cdot X_j + 1)^h$

Gaussian radial basis function kernel: $K(X_i, X_j) = e^{-\|X_i - X_j\|^2/2\sigma^2}$

Sigmoid kernel: $K(X_i, X_j) = \tanh(\kappa X_i \cdot X_j - \delta)$

Each of these results in a different nonlinear classifier in (the original) input space. Neural network aficionados will be interested to note that the resulting decision hyper- planes found for nonlinear SVMs are the same type as those found by other well-known neural network classifiers. For instance, an SVM with a Gaussian radial basis func- tion (RBF) gives the same decision hyper plane as a type of neural network known as a radial basis function network. An SVM with a sigmoid kernel is equivalent to a simple two-layer neural network known as a multilayer perceptron (with no hidden layers).
There are no golden rules for determining which admissible kernel will result in the most accurate SVM. In practice, the kernel chosen does not generally make a large difference in resulting accuracy. SVM training always finds a global solution, unlike neural networks, such as backpropagation, where many local minima usually exist (Section 9.2.3).
So far, we have described linear and nonlinear SVMs for binary (i.e., two-class) clas- sification. SVM classifiers can be combined for the multiclass case. See Section 9.7.1 for some strategies, such as training one classifier per class and the use of error-correcting codes.
A major research goal regarding SVMs is to improve the speed in training and testing so that SVMs may become a more feasible option for very large data sets (e.g., millions of support vectors). Other issues include determining the best kernel for a given data set and finding more efficient methods for the multiclass case.

➢ Lazy Learners (or Learning from Your Neighbors)

The classification methods discussed so far in this book—decision tree induction, Bayesian classification, rule-based classification, classification by backpropagation, support vector machines, and classification based on association rule mining—are all examples of *eager learners*. **Eager learners**, when given a set of training tuples, will construct a generalization (i.e., classification) model before receiving new (e.g., test) tuples to classify. We can think of the learned model as being ready and eager to classify previously unseen tuples.
Imagine a contrasting lazy approach, in which the learner instead waits until the last minute before doing any model construction to classify a given test tuple. That is, when given a training tuple, a **lazy learner** simply stores it (or does only a little minor processing) and waits until it is given a test tuple. Only when it sees the test tuple does it perform generalization to classify the tuple based on its similarity to the stored train- ing tuples. Unlike eager learning methods, lazy learners do less work when a training tuple is presented and more work when making a classification or numeric prediction. Because lazy learners store the training tuples or "instances," they are also referred to as **instance-based learners**, even though all learning is essentially based on instances.
When making a classification or numeric prediction, lazy learners can be computationally

expensive. They require efficient storage techniques and are well suited to implementation on parallel hardware. They offer little explanation or insight into the data's structure. Lazy learners, however, naturally support incremental learning. They are able to model complex decision spaces having hyperpolygonal shapes that may not be as easily describable by other learning algorithms (such as hyperrectangular shapes modeled by decision trees). In this section, we look at two examples of lazy learners: *k-nearest-neighbor classifiers* (Section 9.5.1) and *case-based reasoning classifiers* (Section 9.5.2).

k-Nearest-Neighbor Classifiers

The *k*-nearest-neighbor method was first described in the early 1950s. The method is labor intensive when given large training sets, and did not gain popularity until the 1960s when increased computing power became available. It has since been widely used in the area of pattern recognition.

Nearest-neighbor classifiers are based on learning by analogy, that is, by comparing a given test tuple with training tuples that are similar to it. The training tuples are described by *n* attributes. Each tuple represents a point in an *n*-dimensional space. In this way, all the training tuples are stored in an *n*-dimensional pattern space. When given an unknown tuple, a ***k*-nearest-neighbor classifier** searches the pattern space for the *k* training tuples that are closest to the unknown tuple. These *k* training tuples are the *k* "nearest neighbors" of the unknown tuple.

"Closeness" is defined in terms of a distance metric, such as Euclidean distance. The Euclidean distance between two points or tuples, say, $\boldsymbol{X}_1 = (x_{11}, x_{12}, \ldots, x_{1n})$ and $\boldsymbol{X}_2 = (x_{21}, x_{22}, \ldots, x_{2n})$, is

$$dist(\boldsymbol{X}_1, \boldsymbol{X}_2) = \sqrt{\sum_{i=1}^{n} (x_{1i} - x_{2i})^2}. \qquad (9.22)$$

In other words, for each numeric attribute, we take the difference between the corresponding values of that attribute in tuple $\boldsymbol{X}_1$ and in tuple $\boldsymbol{X}_2$, square this difference, and accumulate it. The square root is taken of the total accumulated distance count. Typically, we normalize the values of each attribute before using Eq. (9.22). This helps prevent attributes with initially large ranges (e.g., *income*) from outweighing attributes with initially smaller ranges (e.g., binary attributes). Min-max normalization, for exam-ple, can be used to transform a value *v* of a numeric attribute *A* to v^{r} in the range [0, 1] by computing

$$v' = \frac{v - min_A}{max_A - min_A}, \qquad (9.23)$$

where *min$_A$* and *max$_A$* are the minimum and maximum values of attribute *A*. Chapter 3describes other methods for data normalization as a form of data transformation.
For *k*-nearest-neighbor classification, the unknown tuple is assigned the most common class among its *k*-nearest neighbors. When *k* 1, the unknown tuple is assigned the class of the training tuple that is closest to it in pattern space. Nearest-neighbor classifiers can also be used for numeric prediction, that is, to return a real-valued prediction for a given unknown tuple. In this case, the classifier returns the average value of the real-valued labels associated with the *k*-nearest neighbors of the unknown tuple.
"*But how can distance be computed for attributes that are not numeric, but nominal (or categorical) such as color?*" The previous discussion assumes that the attributes used to describe the tuples are all numeric. For nominal attributes, a simple method is to compare the corresponding value of the attribute in tuple $\boldsymbol{X}_1$ with that in tuple $\boldsymbol{X}_2$. If the two are identical (e.g., tuples $\boldsymbol{X}_1$ and $\boldsymbol{X}_2$ both have the color blue), then the difference between the two is taken as 0. If the two are different (e.g., tuple $\boldsymbol{X}_1$ is blue but tuple $\boldsymbol{X}_2$ is red), then the difference is considered to be 1. Other methods may incorporate more sophisticated schemes for differential grading (e.g., where a larger difference score is assigned, say, for blue and white than for blue and black).
"*What about missing values?*" In general, if the value of a given attribute *A* is missing in tuple $\boldsymbol{X}_1$ and/or in tuple $\boldsymbol{X}_2$, we assume the maximum possible difference. Suppose that each of the attributes has been mapped to the range [0, 1]. For nominal attributes, we take the difference value to be 1 if either one or both of the corresponding values of *A* are missing.

If *A* is numeric and missing from both tuples $\boldsymbol{X}_1$ and $\boldsymbol{X}_2$, then the difference is also taken to be 1. If only one value is missing and the other (which we will call v') is present and normalized, then we can take the difference to be either 1 v' or 0 v' (i.e., 1 v' or v'), whichever is greater.

"*How can I determine a good value for k, the number of neighbors?*" This can be determined experimentally. Starting with *k* 1, we use a test set to estimate the error rate of the classifier. This process can be repeated each time by incrementing *k* to allow for one more neighbor. The *k* value that gives the minimum error rate may be selected. In general, the larger the number of training tuples, the larger the value of *k* will be (so that classification and numeric prediction decisions can be based on a larger portion of the stored tuples). As the number of training tuples approaches infinity and *k* = 1, the error rate can be no worse than twice the Bayes error rate (the latter being the theoretical minimum). If *k* also approaches infinity, the error rate approaches the Bayes error rate. Nearest-neighbor classifiers use distance-based comparisons that intrinsically assign equal weight to each attribute. They therefore can suffer from poor accuracy when given noisy or irrelevant attributes. The method, however, has been modified to incorporate attribute weighting and the pruning of noisy data tuples. The choice of a distance metric can be critical. The Manhattan (city block) distance (Section 2.4.4), or other distance measurements, may also be used.

Nearest-neighbor classifiers can be extremely slow when classifying test tuples. If D is a training database of $|D|$ tuples and $k = 1$, then $O(|D|)$ comparisons are required to classify a given test tuple. By presorting and arranging the stored tuples into search trees, the number of comparisons can be reduced to $O(\log(|D|))$. Parallel implementation can reduce the running time to a constant, that is, $O(1)$, which is independent of $|D|$.

Other techniques to speed up classification time include the use of *partial distance* calculations and *editing* the stored tuples. In the **partial distance** method, we compute the distance based on a subset of the n attributes. If this distance exceeds a threshold, then further computation for the given stored tuple is halted, and the process moves on to the next stored tuple. The **editing** method removes training tuples that prove useless. This method is also referred to as **pruning** or **condensing** because it reduces the total number of tuples stored.

Case-Based Reasoning

Case-based reasoning (CBR) classifiers use a database of problem solutions to solve new problems. Unlike nearest-neighbor classifiers, which store training tuples as points in Euclidean space, CBR stores the tuples or "cases" for problem solving as complex symbolic descriptions. Business applications of CBR include problem resolution for customer service help desks, where cases describe product-related diagnostic problems. CBR has also been applied to areas such as engineering and law, where cases are either technical designs or legal rulings, respectively. Medical education is another area for CBR, where patient case histories and treatments are used to help diagnose and treat new patients.

When given a new case to classify, a case-based reasoner will first check if an identical training case exists. If one is found, then the accompanying solution to that case is returned. If no identical case is found, then the case-based reasoner will search for training cases having components that are similar to those of the new case. Conceptually, these training cases may be considered as neighbors of the new case. If cases are represented as graphs, this involves searching for subgraphs that are similar to sub- graphs within the new case. The case-based reasoner tries to combine the solutions of the neighboring training cases to propose a solution for the new case. If incompatibilities arise with the individual solutions, then backtracking to search for other solutions may be necessary. The case-based reasoner may employ background knowledge and problem-solving strategies to propose a feasible combined solution.

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

➢ Other Classification Methods

In this section, we give a brief description of several other classification methods, including genetic algorithms (Section 9.6.1), rough set approach (Section 9.6.2), and fuzzy set approaches (Section 9.6.3). In general, these methods are less commonly used for classification in commercial data mining systems than the methods described earlier in this book. However, these methods show their strength in certain applications, and hence it is worthwhile to include them here.

Genetic Algorithms

Genetic algorithms attempt to incorporate ideas of natural evolution. In general, genetic learning starts as follows. An initial **population** is created consisting of randomly generated rules. Each rule can be represented by a string of bits. As a simple example, suppose that samples in a given training set are described by two Boolean attributes, A_1 and A_2, and that there are two classes, C_1 and C_2. The rule "*IF* A_1 *AND NOT* A_2 *THEN* C_2" can be encoded as the bit string "100," where the two leftmost bits represent attributes A_1 and A_2, respectively, and the rightmost bit represents the class. Similarly, the rule "*IF NOT* A_1 *AND NOT* A_2 *THEN* C_1" can be encoded as "001." If an attribute has k values, where $k > 2$, then k bits may be used to encode the attribute's values. Classes can be encoded in a similar fashion.

Based on the notion of survival of the fittest, a new population is formed to consist of the *fittest* rules in the current population, as well as *offspring* of these rules. Typically, the **fitness** of a rule is assessed by its classification accuracy on a set of training samples. Offspring are created by applying genetic operators such as crossover and mutation.

In **crossover**, substrings from pairs of rules are swapped to form new pairs of rules. In **mutation**, randomly selected bits in a rule's string are inverted.

The process of generating new populations based on prior populations of rules continues until a population, P, evolves where each rule in P satisfies a prespecified fitness threshold.

Genetic algorithms are easily parallelizable and have been used for classification as well as other optimization problems. In data mining, they may be used to evaluate the fitness of other algorithms.

Rough Set Approach

Rough set theory can be used for classification to discover structural relationships within imprecise or noisy data. It applies to discrete-valued attributes. Continuous-valued attributes must therefore be discretized before its use.

Rough set theory is based on the establishment of **equivalence classes** within the given training data. All the data tuples forming an equivalence class are indiscernible, that is, the samples are identical with respect to the attributes describing the data. Given real-world data, it is common that some classes cannot be distinguished in terms of the available attributes. Rough sets can be used to approximately or "roughly" define such classes. A rough set definition for a given class, *C*, is approximated by two sets—a **lower approximation** of *C* and an **upper approximation** of *C*. The lower approximation of *C* consists of all the data tuples that, based on the knowledge of the attributes, are certain to belong to *C* without ambiguity. The upper approximation of *C* consists of all the tuples that, based on the knowledge of the attributes, cannot be described as not belonging to
C. The lower and upper approximations for a class *C* are shown in Figure 9.14, where each rectangular region represents an equivalence class. Decision rules can be generated for each class. Typically, a decision table is used to represent the rules.

Rough sets can also be used for attribute subset selection (or feature reduction, where attributes that do not contribute to the classification of the given training data can be identified and removed) and relevance analysis (where the contribution or significance of each attribute is assessed with respect to the classification task). The problem of find- ing the minimal subsets (**reducts**) of attributes that can describe all the concepts in the given data set is NP-hard. However, algorithms to reduce the computation intensity have been proposed. In one method, for example, a **discernibility matrix** is used that stores the differences between attribute values for each pair of data tuples. Rather than

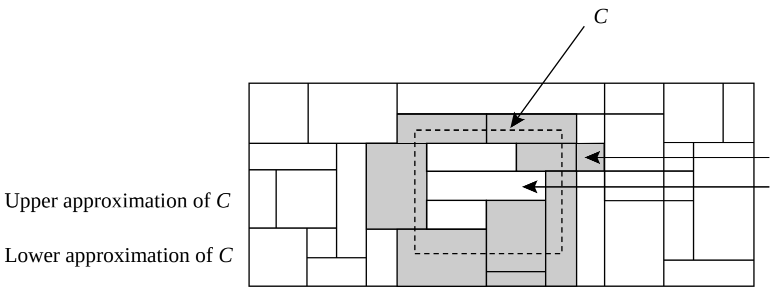


Figure 9.14 A rough set approximation of class *C*'s set of tuples using lower and upper approximationsets of *C*. The rectangular regions represent equivalence classes.

searching on the entire training set, the matrix is instead searched to detect redundant attributes.

Fuzzy Set Approaches

Rule-based systems for classification have the disadvantage that they involve sharp cut-offs for continuous attributes. For example, consider the following rule for customer credit application approval. The rule essentially says that applications for customers who have had a job for two or more years and who have a high income (i.e., of at least
$50,000) are approved:

IF (years employed ≥ 2) AND (income ≥ 50,000) THEN credit = approved. (9.24)

By Rule (9.24), a customer who has had a job for at least two years will receive credit if her income is, say, $50,000, but not if it is $49,000. Such harsh thresholding may seem unfair.
Instead, we can discretize *income* into categories (e.g., *low income, medium income, high income*) and then apply **fuzzy logic** to allow "fuzzy" thresholds or boundaries to be defined for each category (Figure 9.15). Rather than having a precise cutoff between categories, fuzzy logic uses truth values between 0.0 and 1.0 to represent the degree of membership that a certain value has in a given category. Each category then represents a **fuzzy set**. Hence, with fuzzy logic, we can capture the notion that an income of $49,000 is, more or less, high, although not as high as an income of $50,000. Fuzzy logic systems typically provide graphical tools to assist users in converting attribute values to fuzzy truth values.
Fuzzy set theory is also known as **possibility theory**. It was proposed by Lotfi Zadeh in 1965 as an alternative to traditional two-value logic and probability theory. It lets us work at a high abstraction level and offers a means for dealing with imprecise data

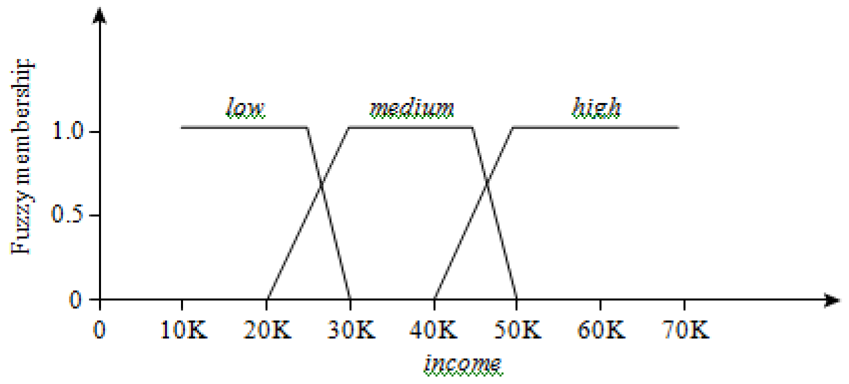


Figure 9.15 Fuzzy truth values for *income*, representing the degree of membership of *income* values with respect to the categories {*low, medium, high*}. Each category represents a fuzzy set. Note that a given income value, *x*, can have membership in more than one fuzzy set. The membership values of *x* in each fuzzy set do not have to total to 1.

measurement. Most important, fuzzy set theory allows us to deal with vague or inexact facts. For example, being a member of a set of high incomes is inexact (e.g., if \$50,000 is high, then what about \$49,000? or \$48,000?) Unlike the notion of traditional "crisp" sets where an element belongs to either a set *S* or its complement, in fuzzy set theory, elements can belong to more than one fuzzy set. For example, the income value \$49,000 belongs to both the *medium* and *high* fuzzy sets, but to differing degrees. Using fuzzy set notation and following Figure 9.15, this can be shown as

$$m_{medium\ income}(\$49{,}000) = 0.15 \text{ and } m_{high\ income}(\$49{,}000) = 0.96,$$

where *m* denotes the membership function, that is operating on the fuzzy sets of *medium income* and *high income*, respectively. In fuzzy set theory, membership val- ues for a given element, *x* (e.g., for \$49,000), do not have to sum to 1. This is unlike traditional probability theory, which is constrained by a summation axiom.

Fuzzy set theory is useful for data mining systems performing rule-based classification. It provides operations for combining fuzzy measurements. Suppose that in addition to the fuzzy sets for *income*, we defined the fuzzy sets *junior employee* and *senior employee* for the attribute *years employed*. Suppose also that we have a rule that, say, tests *high income* and *senior employee* in the rule antecedent (IF part) for a given employee, *x*. If these two fuzzy measures are ANDed together, the minimum of their measure is taken as the measure of the rule. In other words,

$$m_{(high\ income\ AND\ senior\ employee)}(x) = min(m_{high\ income}(x), m_{senior\ employee}(x)).$$

This is akin to saying that a chain is as strong as its weakest link. If the two measures are ORed, the maximum of their measure is taken as the measure of the rule. In other words,

$$m_{(high\ income\ OR\ senior\ employee)}(x) = max(m_{high\ income}(x), m_{senior\ employee}(x)).$$

Intuitively, this is like saying that a rope is as strong as its strongest strand.
Given a tuple to classify, more than one fuzzy rule may apply. Each applicable rule contributes a vote for membership in the categories. Typically, the truth values for each predicted category are summed, and these sums are combined. Several procedures exist for translating the resulting fuzzy output into a *defuzzified* or crisp value that is returned by the system.
Fuzzy logic systems have been used in numerous areas for classification, including market research, finance, health care, and environmental engineering.

➢ Model Evaluation and Selection

Now that you may have built a classification model, there may be many questions going through your mind. For example, suppose you used data from previous sales to build a classifier to predict customer purchasing behavior. You would like an estimate of how accurately the classifier can predict the purchasing behavior of future customers, that is, future customer data on which the classifier has not been trained. You may even have tried different methods to build more than one classifier and now wish to compare their accuracy. 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? These questions are addressed in this section.
Section 8.5.1 describes various evaluation metrics for the predictive accuracy of a classifier. Holdout and random subsampling (Section 8.5.2), cross-validation (Section 8.5.3), and bootstrap methods (Section 8.5.4) are common techniques for assessing accuracy, based on randomly sampled partitions of the given data. 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). The last two sections address this issue. Section 8.5.5 discusses how to use tests of statistical significance to assess whether the difference in accuracy between two classifiers is due to chance. Section 8.5.6 presents how to compare classifiers based on cost–benefit and receiver operating characteristic (ROC) curves.

Metrics for Evaluating Classifier Performance

This section presents measures for assessing how good or how "accurate" your classifier is at predicting the class label of tuples. We will consider the case of where the class tuples are more or less evenly distributed, as well as the case where classes are unbalanced (e.g., where an important class of interest is rare such as in medical tests). The classifier evaluation measures presented in this section are summarized in Figure 8.13. 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 over-specialization of the learning algorithm to the data. (We will say more on this in a moment!) Instead, it is better to measure the classifier's accuracy on a *test set* consisting of class-labeled tuples that were not used to train the model.

Before we discuss the various measures, we need to become comfortable with some terminology. Recall that we can talk in terms of **positive tuples** (tuples of the main class of interest) and **negative tuples** (all other tuples).[6] Given two classes, for example, the positive tuples may be *buys computer* = *yes* while the negative tuples are

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).

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.
There are **four additional terms** we need to know that are the "building blocks" used in computing many evaluation measures. Understanding them will make it easy to grasp the meaning of the various measures.

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 neg- ative (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.,

Actual class	**Predicted class**		
	yes	*no*	Total
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.

mislabeling). Given *m* classes (where *m* ≥ 2), a **confusion matrix** is a table of at least size *m* by *m*. An entry, $CM_{i,j}$ 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 $CM_{m,m}$, with the rest of the entries being zero or close to zero. That is, ideally, *FP* and *FN* are around zero.

The table may have additional rows or columns to provide totals. For example, in the confusion matrix of Figure 8.14, P and N are shown. In addition, P' is the number of tuples that were labeled as positive $(TP + FP)$ and N' is the number of tuples that were labeled as negative $(TN + FN)$. The total number of tuples is $TP + TN + FP + FN$, or $P + N$, or $P' + N'$. Note that although the confusion matrix shown is for a binary classification problem, confusion matrices can be easily drawn for multiple classes in a similar manner.

Now let's look at the evaluation measures, starting with accuracy. 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,

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

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. Totals are shown, as well as the recognition rates per class and overall. By glancing at a confusion matrix, it is easy to see if the corresponding classifier is confusing two classes.

For example, we see that it mislabeled 412 *"no"* tuples as *"yes."* Accuracy is most effective when the class distribution is relatively balanced.

We can also speak of the **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

$$sensitivity = \frac{TP}{P} \qquad (8.23)$$

$$specificity = \frac{TN}{N}. \qquad (8.24)$$

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

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

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

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*.

of the classifier is $\frac{90}{300}$ = 30.00%. The specificity is $\frac{9560}{9700}$ = 98.56%. The classifier's over- all accuracy is $\frac{9650}{10,000}$ 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 sen- sitivity. It has high specificity, meaning that it can accurately recognize negative tuples. Techniques for handling class-imbalanced data are given in Section 8.6.5. ■

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

$$precision = \frac{TP}{TP + FP} \quad (8.26)$$

$$recall = \frac{TP}{TP + FN} = \frac{TP}{P}. \quad (8.27)$$

Example 8.10 Precision and recall. The precision of the classifier in Figure 8.16 for the *yes* class is $\frac{90}{230}$ = 39.13%. The recall is $\frac{90}{300}$ = 30.00%, which is the same calculation for sensitivity in Example 8.9. ■

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 mea- sure. 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

$$F = \frac{2 \times precision \times recall}{precision + recall} \qquad (8.28)$$

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

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.

Rather than returning a class label, it is useful to return a probability class distribution. Accuracy measures may then use a **second guess** heuristic, whereby a class prediction is judged as correct if it agrees with the first or second most probable class. Although this does take into consideration, to some degree, the nonunique classification of tuples, it is not a complete solution.

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. Decision trees and classification rules can be easy to interpret, yet their interpretability may diminish the more they become complex. We discuss some work in this area, such as the extraction of classification rules from a "black box" neural network classifier called backpropagation, in Chapter 9.

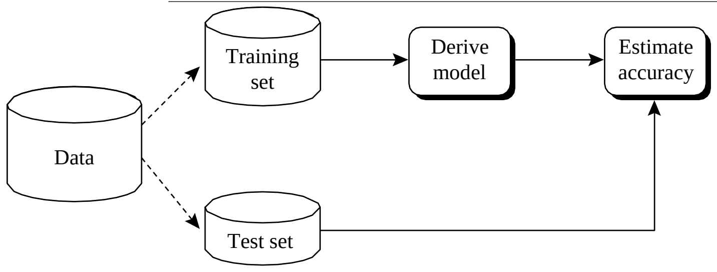


Figure 8.17 Estimating accuracy with the holdout method.

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. The remaining subsections focus on obtaining reliable classifier accuracy estimates.

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 (Figure 8.17). The estimate is pessimistic because only a portion of the initial data is used to derive the model.
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.

Cross-Validation

In ***k*-fold cross-validation**, the initial data are randomly partitioned into k mutually exclusive subsets or "folds," $D_1, D_2, \dots, D_k$, each of approximately equal size. Training and testing is performed k times. In iteration i, partition D_i is reserved as the test set, and the remaining partitions are collectively used to train the model. That is, in the first iteration, subsets $D_2, \dots, D_k$ collectively serve as the training set to obtain a first model, which is tested on D_1; the second iteration is trained on subsets $D_1, D_3, \dots, D_k$ 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.

Bootstrap

Unlike the accuracy estimation methods just mentioned, 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.[7] 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 $Acc(M_i)_{test\ \ set}$ is the accuracy of the model obtained with bootstrap sample *i* when it is applied to test set *i*. $Acc(M_i)_{train\ \ set}$ is the accuracy of the model obtained with boot- strap 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 rate[8] 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? This section addresses these questions.

To determine if there is any "real" difference in the mean error rates of two models, we need to employ a *test of statistical significance*. In addition, we want to obtain some confidence limits for our mean error rates so that we can make statements like, *"Any observed mean will not vary by two standard errors 95% of the time for future samples"* or *"One model is better than th±e other by a margin of error of 4%."*

±

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, 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 gen- eral, 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 bell- shaped.) 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 M_1 and M_2. In such cases, we do a **pairwise comparison** of the two models *for each* 10-fold cross-validation round. That is, for the *i*th round of 10-fold cross-validation, the same cross-validation partitioning is used to obtain an error rate for M_1 and for M_2. Let $err(M_1)_i$ (or $err(M_2)_i$) be the error rate of model M_1 (or M_2) on round *i*. The error rates for M_1 are averaged to ob~~tain~~ a mean error rate for ~~M_1,~~ denoted $err(M_1)$. Similarly, we can obtain $err(M_2)$. The variance of the difference between the two models is denoted $var(M_1$ $M_2)$. The *t* -test computes the *t-statistic with k* 1 *degrees of freedom* for *k* samples. In our example we have *k* 10 since, here, the *k* samples are our error rates obtained from ten 10-fold cross-validations for each model. The t-statistic for pairwise comparison is computed as follows:

$$t = \frac{\overline{err}(M_1) - \overline{err}(M_2)}{\sqrt{var(M_1 - M_2)/k}} \qquad (8.31)$$

where

$$var(M_1 - M_2) = \frac{1}{k}\sum_{i=1}^{k}\left[err(M_1)_i - err(M_2)_i - (\overline{err}(M_1) - \overline{err}(M_2))\right]^2 . \qquad (8.32)$$

To determine whether M_1 and M_2 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*, available in standard textbooks on statistics. This table is usually shown arranged by degrees of freedom as rows and significance levels as columns. Suppose we want to ascertain whether the difference between M_1 and M_2 is significantly different for 95% of the population, that is, *sig* 5% or 0.05. We need to find the t -distribution value corresponding to k 1 degrees of freedom (or 9 degrees of freedom for our example) from the table. 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 M_1 and M_2 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 M_1 and M_2 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 k_1 and k_2 are the number of cross-validation samples (in our case, 10-fold cross-validation rounds) used for M_1 and M_2, respectively. This is also known as the **two sample *t*-test**.[9] When consulting the table of t -distribution, the number of degrees of freedom used is taken as the minimum number of degrees of the two models.

Comparing Classifiers Based on Cost–Benefit and ROC Curves

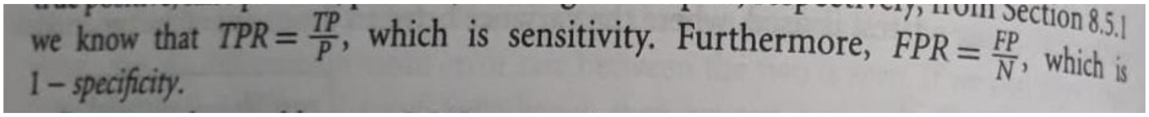
The true positives, true negatives, false positives, and false negatives are also useful in assessing the **costs and benefits** (or risks and gains) associated with a classification

[9]This test was used in sampling cubes for OLAP-based mining in Chapter 5.

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

Alternatively, we can incorporate costs and benefits by instead computing the average cost (or benefit) per decision. Other applications involving cost–benefit analysis include loan application decisions and target marketing mailouts. For example, the cost of loan- ing to a defaulter greatly exceeds that of the lost business incurred by denying a loan to a nondefaulter. Similarly, in an application that tries to identify households that are likely to respond to mailouts of certain promotional material, the cost of mailouts to numerous households that do not respond may outweigh the cost of lost business from not mailing to households that would have responded. Other costs to consider in the overall analysis include the costs to collect the data and to develop the classification tool.

Receiver operating characteristic curves are a useful visual tool for comparing two classification models. ROC curves come from signal detection theory that was deve- loped 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, from Section 8.5.1


respectively, from Section 8.5.1
we know that $TPR = \frac{TP}{P}$, which is sensitivity. Furthermore, $FPR = \frac{FP}{N}$, 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 bottom of the list. Na¨ıve Bayesian (Section 8.3) and backpropagation (Section 9.2) classifiers return a class probability distribution for each prediction and, therefore, are appropriate, although other classifiers, such as decision tree classifiers (Section 8.2), can easily be modified

to return class probability predictions. Let the value that a probabilistic classifier returns for a given tuple ***X*** be *f* (***X***) [0, 1]. For a binary problem, a threshold *t* is typically≥selected so that tuples where *f* (***X***) *t* are considered positive and all the other tuples are considered negative. Note that the number of true positives and the number of false positives are both functions of *t* , so that we could write *TP(t)* and *FP(t)*. Both are monotonic descending functions.
We first describe the general idea behind plotting an ROC curve, and then follow up with an example. The vertical axis of an ROC curve represents *TPR*. The horizontal axis represents *FPR*. To plot an ROC curve for *M*, we begin as follows. Starting at the bottom left corner (where *TPR* *FPR* 0), we check the tuple's actual class label at the top of the list. If we have a true positive (i.e., a positive tuple that was correctly classified), then *TP* and thus *TPR* increase. On the graph, we move up and plot a point. If, instead, the model classifies a negative tuple as positive, we have a false positive, and so both *FP* and *FPR* increase. On the graph, we move right and plot a point. This process is repeated for each of the test tuples in ranked order, each time moving up on the graph for a true positive or toward the right for a false positive.

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 probabilisticclassifier.

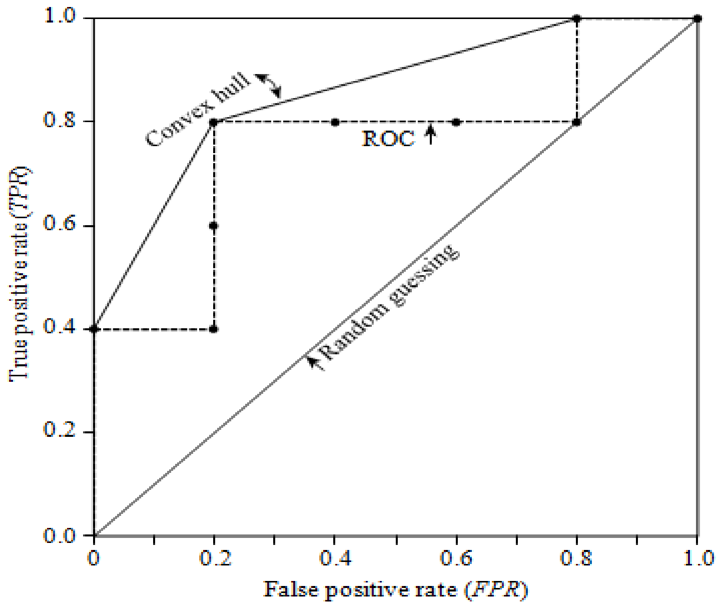


Figure 8.19 ROC curve for the data in Figure 8.18.

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 $TPR = \frac{TP}{P} = \frac{1}{5} = 0.2$, while $FPR = 0$. Thus, we have the point $(0.2, 0)$ for the

ROC curve.

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.

There are many methods to obtain a curve out of these points, the most common of which is to use a convex hull. The plot also shows a diagonal line where for every true positive of such a model, we are just as likely to encounter a false positive. For comparison, this line represents random guessing. ■

Figure 8.20 shows the ROC curves of two classification models. 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,

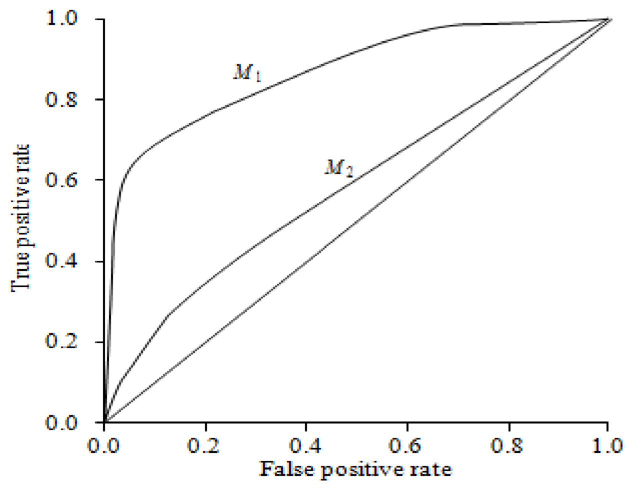


Figure 8.20 ROC curves of two classification models, M_1 and M_2. The diagonal shows where, for every true positive, we are equally likely to encounter a false positive. The closer an ROC curve isto the diagonal line, the less accurate the model is. Thus, *M*1 is more accurate here.

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. To assess the accuracy of a model, we can measure the area under the curve. Several software packages are able to perform such calculation. The closer the area is to 0.5, the less accurate the corresponding model is. A model with perfect accuracy will have an area of 1.0.

➢ 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. We start off in Section 8.6.1 by introducing ensemble methods in general. Bagging (Section 8.6.2), boosting (Section 8.6.3), and random forests (Section 8.6.4) are popular ensemble methods.

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*. We also study techniques for improving the classification accuracy of class-imbalanced data. These are presented in Section 8.6.5.

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, . . . , 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, . . . , 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, con- sider an ensemble that performs majority voting. That is, given a tuple ***X*** to classify, it collects the class label predictions returned from the 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, x_1 and x_2. 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.

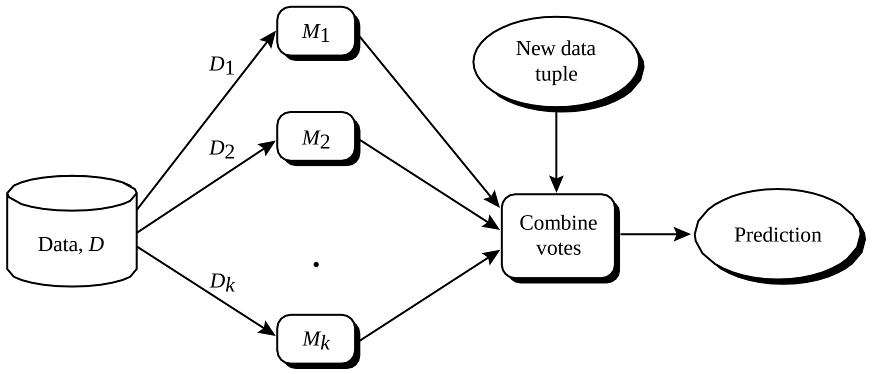


Figure 8.21 Increasing classifier accuracy: Ensemble methods generate a set of classification models, M_1, M_2, . . . , M_k. Given a new data tuple to classify, each classifier "votes" for the class label of that tuple. The ensemble combines the votes to return a class prediction.

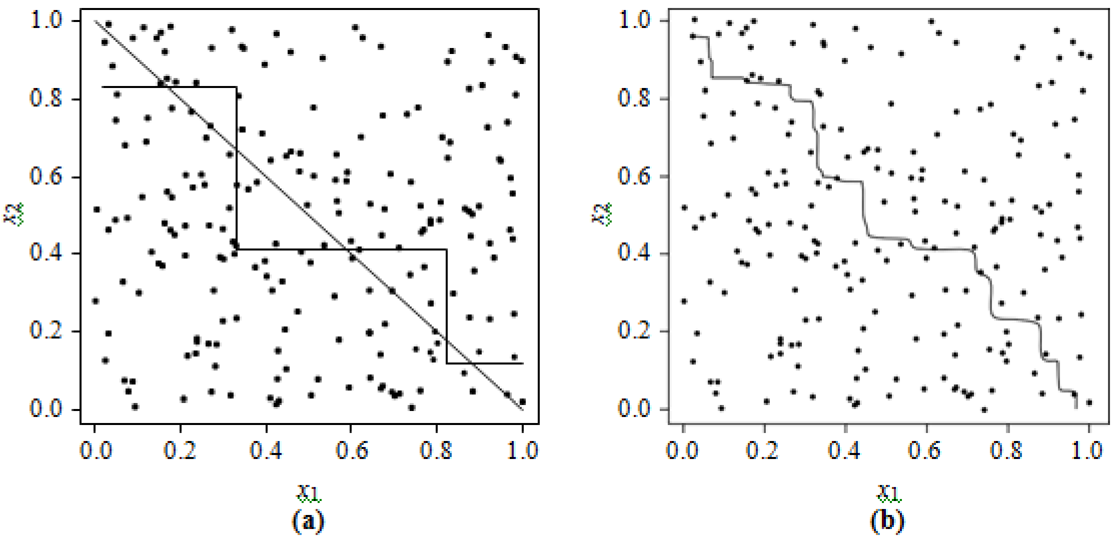


Figure 8.22 Decision boundary by (a) a single decision tree and (b) an ensemble of decision trees for a linearly separable problem (i.e., where the actual decision boundary is a straight line). The decision tree struggles with approximating a linear boundary. The decision boundary of the ensemble is 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* $(i = 1, 2, \ldots, k)$, a training set, D_i, 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 D_i, whereas others may occur more than once. A classifier model, M_i, is learned for each training set, D_i. To classify an unknown tuple, ***X***, each classifier, M_i, 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. The algorithm is summarized in Figure 8.23.

The bagged classifier often has significantly greater accuracy than a single classifier derived from *D*, the original training data. It will not be considerably worse and is more

Algorithm: Bagging. The bagging algorithm—create an ensemble of classification modelsfor 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, na¨ıve Bayesian, etc.).

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

Method:

```
(1)   for $i$ = 1 to $k$ do // create $k$ models:
(2)       create bootstrap sample, $D_i$, by sampling $D$ with replacement;
(3)       use $D_i$ and the learning scheme to derive a model, $M_i$;
(4)   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.

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

We now look at the ensemble method of boosting. 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 doc- tor'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, M_i, is learned, the weights are updated to allow the subsequent classifier, M_{i+1}, to "pay more attention" to the training tuples that were mis- classified by M_i. 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, $(\mathbf{X}_1, y_1), (\mathbf{X}_2, y_2), \ldots, (\mathbf{X}_d, y_d)$, where y_i is the class label of tuple $\mathbf{X}_i$. 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, M_i, is derived from the training tuples of D_i. Its error is then calculated using D_i 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. The algorithm is summarized in Figure 8.24.

Now, let's look at some of the math that's involved in the algorithm. To compute the error rate of model M_i, we sum the weights of each of the tuples in D_i that M_i misclassified. That is,

$$error(M_i) = \sum_{j=1}^{d} w_j \times err(\mathbf{X}_j), \qquad (8.34)$$

where $err(\mathbf{X}_j)$ is the misclassification error of tuple $\mathbf{X}_j$: If the tuple was misclassified, then $err(\mathbf{X}_j)$ is 1; otherwise, it is 0. If the performance of classifier M_i is so poor that its error exceeds 0.5, then we abandon it. Instead, we try again by generating a new D_i training set, from which we derive a new M_i.

The error rate of M_i affects how the weights of the training tuples are updated. If a tuple in round i was correctly classified, its weight is multiplied by $error(M_i)/(1\ error(M_i))$. Once the weights of āll the correctly classified tuples are updated, the weights for all tuples (including the misclassified ones) are normalized so that their sum remains the same as it was before. To normalize a weight, we multiply it by the sum of the old weights, divided by the sum of the new weights. As a result, the weights of mis-classified tuples are increased and the weights of correctly classified tuples are decreased, as described before.

*"Once boosting is complete, how is the ensemble of classifiers used to predict the class label of a tuple, **X**?"* Unlike bagging, where each classifier was assigned an equal vote, boosting assigns a weight to each classifier's vote, based on how well the classifier performed. The lower a classifier's error rate, the more accurate it is, and therefore, the higher its weight for voting should be. The weight of classifier M_i's vote is

$$\log \frac{1 - error(M_i)}{error(M_i)}. \qquad (8.35)$$

For each class, c, we sum the weights of each classifier that assigned class c to $\mathbf{X}$. The class with the highest sum is the "winner" and is returned as the class prediction for tuple $\mathbf{X}$. *"How does boosting compare with bagging?"* Because of the way boosting focuses on the misclassified tuples, it risks overfitting the resulting composite model to such data.

Algorithm: AdaBoost. A boosting algorithm—create an ensemble of classifiers. Each onegives 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)       $c = M_i(\boldsymbol{X})$; // get class prediction for $\boldsymbol{X}$ 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.

Therefore, sometimes the resulting "boosted" model may be less accurate than a single model derived from the 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.

Random Forests

We now present another ensemble method called **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, D_i, of *d* tuples is sampled with replacement from *D*. That is, each D_i is a bootstrap sample of *D* (Section 8.5.4), so that some tuples may occur more than once in D_i, 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 avail- able attributes. To construct a decision tree classifier, M_i, 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 num-ber 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. Typically, up to $log_2 d$ 1 are chosen. (An interesting empirical observation was that using a single random input attribute may result in good accuracy that is often higher than when using several attributes.) Because random forests consider many fewer attributes for each split, they are efficient on very large databases. They can be faster than either bagging or boosting. Random forests give internal estimates of variable importance.

Improving Classification Accuracy of Class-Imbalanced Data

In this section, we revisit the *class imbalance problem*. In particular, we study approaches to improving the 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. Earlier parts of this chapter presented ways of addressing the class imbalance problem. Although the accuracy measure assumes that the cost of classes are equal, alternative evaluation metrics can be used that consider the different types of classifications. Section 8.5.1, for example, presented senstivity or recall (the true positive rate) and *specificity* (the true negative rate), which help to assess how well a classifier can predict the class label of imbalanced data. Additional relevant measures discussed include F_1 and F_β. Section 8.5.6 showed how ROC cūrves plot *sensitivity* versus 1 *specificity* (i.e., the false positive rate). Such curves can provide insight when studying the performance of classifiers on 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) under-sampling, (3) threshold moving, and (4) ensemble techniques. The first three do not involve any changes to the construction of the classification model. That is, over sampling and under sampling change the distribution of tuples in the training set; threshold moving affects how the model makes decisions when classifying new data. Ensemble methods follow the techniques described in Sections 8.6.2 through 8.6.4. For ease of explanation, we describe these general approaches with respect to the two-class imbalance data problem, where the higher-cost classes are rarer than the lower-cost classes.

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 pos- itive 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 (just like in Section 8.5.6, where we discussed how to construct ROC curves). That is, for an input tuple, $\boldsymbol{X}$, such a classifier returns as output a mapping, $f(\boldsymbol{X})$ [0, 1]. Rather than manipulating the training tuples, this method returns a clas- sification decision based on the output values. In $\geq$ the simplest approach, tuples for which $f(\boldsymbol{X})$ t , for some threshold, t , are considered positive, while all other tuples are con- sidered negative. Other approaches may involve manipulating the outputs by weighting. In general, threshold moving moves the threshold, t , so that the rare class tuples are eas-ier to classify (and hence, there is less chance of costly false negative errors). Examples of such classifiers include na¨ıve Bayesian classifiers (Section 8.3) and neural network clas- sifiers like backpropagation (Section 9.2). The threshold-moving method, although not as popular as over- and undersampling, is simple and has shown some success for the two-class-imbalanced data.

Ensemble methods (Sections 8.6.2 through 8.6.4) 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. These methods work relatively well for the class imbalance problem on two-class tasks. Threshold-moving and ensemble methods were empirically observed to outper- form oversampling and undersampling. Threshold moving works well even on data sets that are extremely imbalanced. The class imbalance problem on multiclass tasks is much more difficult, where oversampling and threshold moving are less effective.

Although threshold-moving and ensemble methods show promise, finding a solution for the multiclass imbalance problem remains an area of future work.