UNIT-III

CLASSIFICATION

Classification and Prediction – Basic Concepts – Decision Tree Induction – Bayesian Classification – Rule Based Classification – Lazy Learners

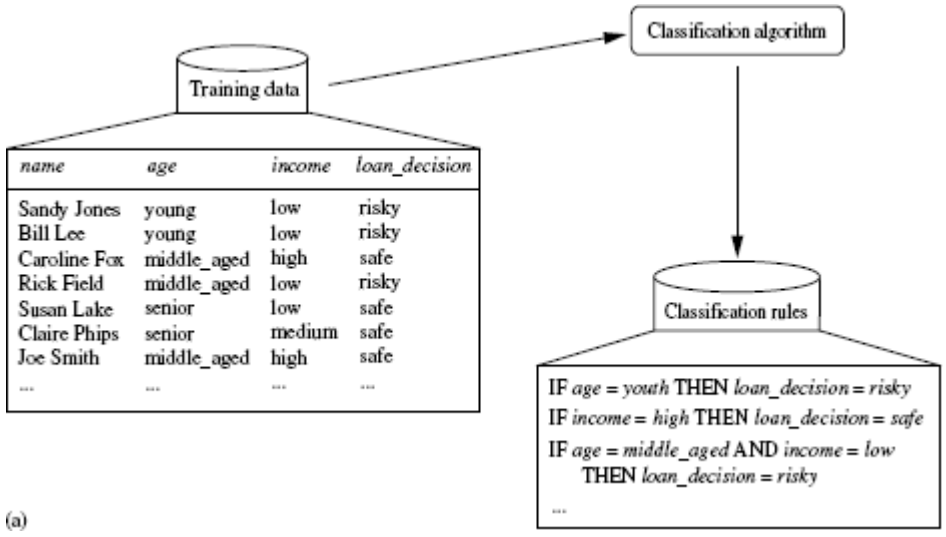
Classification and Prediction

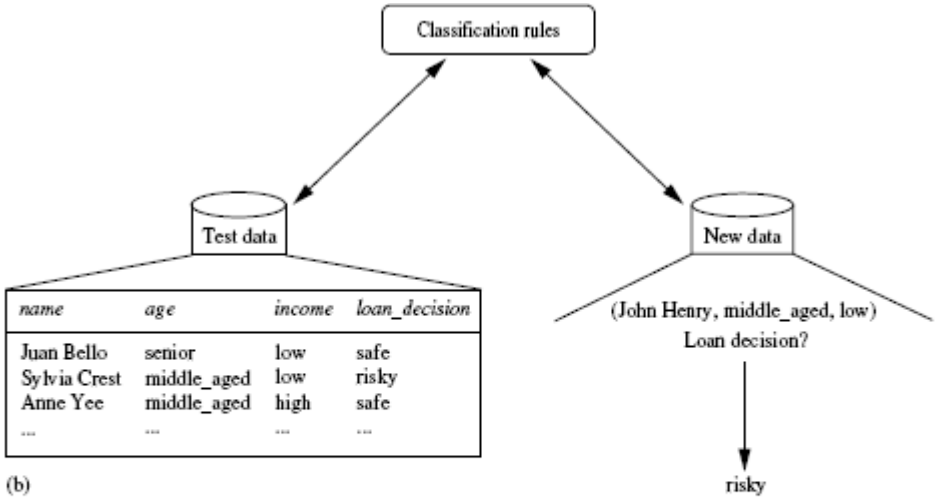
What Is Classification? What Is Prediction?

A bank loans officer needs analysis of her data in order 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 in order 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 *categorical labels*, such as ―safe‖ or ―risky‖ for the loan application data; ―yes‖ or ―no‖ for the marketing data; or ―treatment A,‖ ―treatment B,‖ or ―treatment C‖ for the medical data. These categories can be represented by discrete values, where the ordering among values has no meaning. For example, the values 1, 2, and 3 may be used to represent treatments A, B, and C, where there is no ordering implied among this group of treatment regimes.

Suppose that the marketing manager would like to predict how much a given cus tomer 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 categorical label. This model is a predictor

"How does classification work? Data classification is a two-step process, as shown for the loan application data of Figure 6.1. (The data are simplified for illustrative purposes. In reality, we may expect many more attributes to be considered.) 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.





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

Issues Regarding Classification and Prediction

- **Data cleaning**: This refers to the preprocessing of data in order to remove or reduce *noise* (by applying smoothing techniques, for example) and the treatment of *missing values* (e.g., by replacing a missing value with the most commonly occurring value for that attribute, or with the most probable value based on statistics). Although most classification algorithms have some mechanisms for handling noisy or missing data, this step can help reduce confusion during learning.

- **Relevance analysis**: Many of the attributes in the data may be *redundant*. Correlation analysis can be used to identify whether any two given attributes are statistically related. For example, a strong correlation between attributes A_1 and A_2 would suggest that one of the two could be removed from further analysis. A database may also contain *irrelevant* attributes. Attribute subset selection[4] can be used in these cases to find a reduced set of attributes such that the resulting probability distribution of the data classes is as close as possible to the original distribution obtained using all attributes. Hence, relevance analysis, in the form of correlation analysis and attribute subset selection, can be used to detect attributes that do not contribute to the classification or prediction task. Including such attributes may otherwise slow down, and possibly mislead, the learning step. Ideally, the time spent on relevance analysis, when added to the time spent on learning from the resulting ―reduced‖ attribute (or feature) subset, should be less than the time that would have been spent on learning from the original set of attributes. Hence, such analysis can help improve classification efficiency and scalability.

- **Data transformation and reduction:** The data may be transformed by normalization, particularly when neural networks or methods involving distance measurements are used in the learning step. Normalization involves scaling all values for a given attribute so that they fall within a small specified range, such as -1.0 to 1.0, or 0.0 to 1.0. In methods that use distance measurements, for example, this would prevent attributes with initially large ranges (like, say, *income*) from out weighing attributes with initially smaller ranges (such as binary attributes).

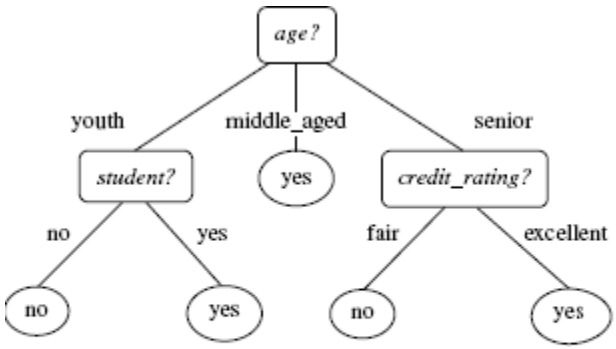
Comparing Classification and Prediction Methods

Classification and prediction methods can be compared and evaluated according to the following criteria:

- **Accuracy**
- **Speed**
- **Robustness**
- **Scalability**
- **Interpretability**

Classification by Decision Tree Induction (16 Mark Question)

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



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

A typical decision tree is shown in Figure. It represents the concept *buys computer*, that is, it predicts whether a customer at *AllElectronics* is likely to purchase a computer. Internal nodes are denoted by rectangles, and leaf nodes are denoted by ovals. Some decision tree algorithms produce only *binary* trees (where each internal node branches to exactly two other nodes), whereas others can produce non binary trees.

"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 hold s the class prediction for that tuple. Decision trees can easily be converted to classification rules.

Decision Tree Induction

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

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

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:

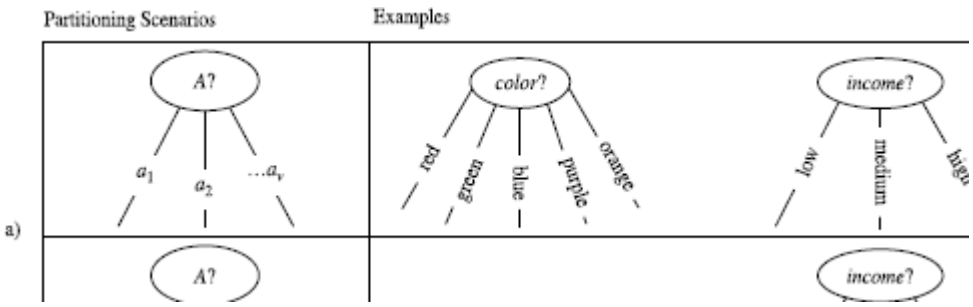
```
(1)  create a node N;
(2)  if tuples in D are all of the same class, C then
(3)      return N as a leaf node labeled with the class C;
(4)  if attribute_list is empty then
(5)      return N as a leaf node labeled with the majority class in D; // majority voting
(6)  apply Attribute_selection_method(D, 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;
```

Basic algorithm for inducing a decision tree from training tuples.

- The tree starts as a single node, *N*, representing the training tuples in *D* (step 1)
- If the tuples in *D* are all of the same class, then node *N* becomes a leaf and is labeled with that class (steps 2 and 3). Note that steps 4 and 5 are terminating conditions. All of the terminating conditions are explained at the end of the algorithm.
- Otherwise, the algorithm calls *Attribute selection method* to determine the splitting criterion. The splitting criterion tells us which attribute to test at node *N* by determining the ―best‖ way to separate or partition the tuples in *D* into individual classes(step 6). The splitting criterion also tells us which branches to grow from node *N* with respect to the outcomes of the chosen test. More specifically, the splitting criterion indicates the splitting attribute and may also indicate either a split-point or a splitting subset. The splitting criterion is determined so that, ideally, the resulting partitions at each branch are as ―pure‖ as possible. A partition is pure if all of the tuples in it belong to the same class. In other words, if we were to split up the tuples in *D* according to the mutually exclusive outcomes of the splitting criterion, we hope for the resulting partitions to be as pure as possible.
- The node *N* is labeled with the splitting criterion, which serves as a test at the node (step 7). A branch is grown from node *N* for each of the outcomes of the splitting criterion. The tuples in *D* are partitioned accordingly (steps 10 to 11). There are three possible scenarios, as

illustrated in Figure. Let A be the splitting attribute. A has v distinct values, {$a1$, $a2$, : : : , av}, based on the training data.$Info(D) = -\sum_{i=1}^{m} p_i \log_2(p_i)$,

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



Attribute Selection Measures

An attribute selection measure is a heuristic for selecting the splitting criterion that ―best‖ separates a given data partition, *D*, of class-labeled training tuples into individual classes. If we were to split *D* into smaller partitions according to the outcomes of the splitting criterion, ideally each partition would be pure (i.e., all of the tuples that fall into a given partition would belong to the same class). Conceptually, the ―best‖ splitting criterion is the one that most closely results in such a scenario. Attribute selection measures are also known as splitting rules because they determine how the tuples at a given nod e 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 measure6 is chosen as the *splitting attribute* for the given tuples. If the splitting attribute is continuous-valued or if we are restricted to binary trees then, respectively, either a *split point* or a *splitting subset* must also be determined as part of the splitting criterion. The tree node created for partition *D* is labeled with the splitting criterion, branches are grown for each outcome of the criterion, and the tuples are partitioned accordingly. This section des cribes three popular attribute selection measures—*information gain*, *gain ratio*, and *gini inde*

Information gain

ID3 uses information gain as its attribute selection measure.

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,

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

In other words, *Gain(A)* tells us how much would be gained by branching on *A*. It is the expected reduction in the information requirement caused by knowing the value of *A*. The attribute *A* with the highest information gain, (*Gain*(*A*)), is chosen as the splitting attribute at node *N*.

Example 6.1 Induction of a decision tree using information gain.

Table 6.1 presents a training set, *D*, of class-labeled tuples randomly selected from the *AllElectronics* customer database. (The data are adapted from [Qui86]. In this example, each attribute is discrete-valued. Continuous-valued attributes have been generalized.) The class label attribute, *buys computer*, has two distinct values (namely, {*yes*, *no*}); therefore, there are two distinct classes (that is, $m = 2$). Let class $C1$ correspond to *yes* and class $C2$ correspond to *no*. There are nine tuples of class *yes* and five tuples of class *no*. A (root) node *N* is created for the tuples in *D*. To find the splitting criterion for these tuples, we must compute the information gain of each attribute. We first use Equation (6.1) to compute the expected information needed to classify a tuple in *D*:

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

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

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

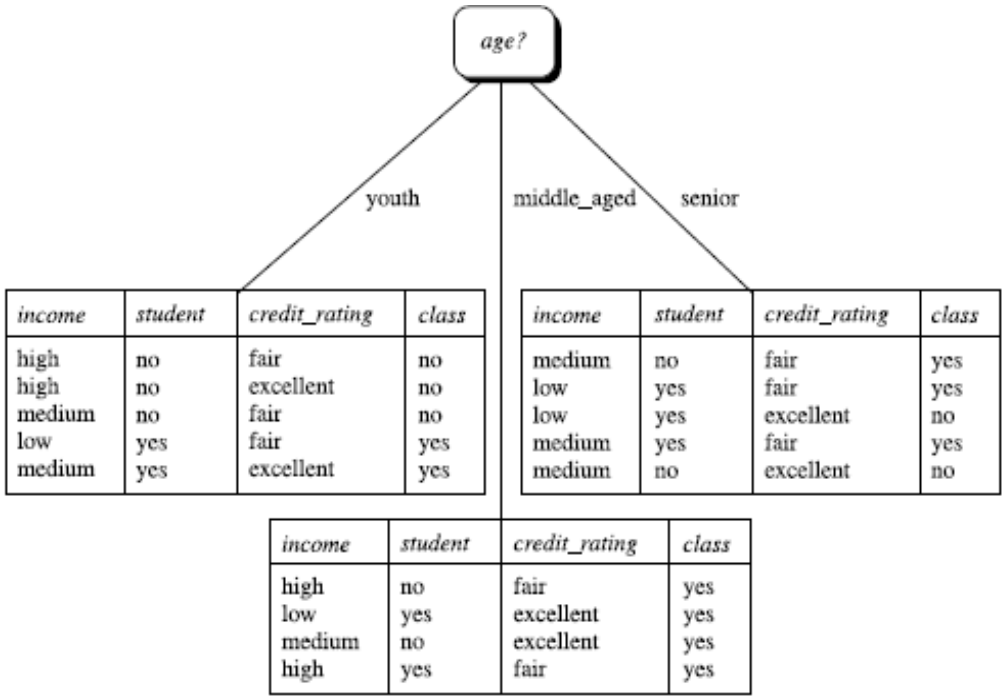
The expected information needed to classify a tuple in *D* if the tuples are partitioned according to *age* is

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

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

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

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



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

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

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

Bayesian Classification (16 mark Question)

"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 below. Studies comparing classification algorithms have found a simple Bayesian classifier known as the *naïve Bayesian classifier* to be comparable in performance with decision tree and selected neural network classifiers. Bayesian classifiers have also exhibited high accu racy and speed when applied to large databases.

1) 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*j***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***.

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

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

2) Naïve Bayesian Classification

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

equally lik
mize $P(X|$
abilities ma
tuples of cl

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, $A_1, A_2, \ldots, A_n$.

4. Given data
sive to con
naive assu
the values
class label
attributes).

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 naïve Bayesian classifier predicts that tuple X belongs to the class C_i if and only if

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

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 (Equation (6.10)),

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

$P(X|C_i)$

3. As $P(X)$ is constant for all classes, only $P(X|C_i)P(C_i)$ need be maximized. If the class prior probabilities are not known, then it is commonly assumed that the classes are

5. In order 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

$$P(X|C_i)P(C_i) > P(X|C_j)P(C_j) \quad \text{for } 1 \le j \le m, j \ne i. \tag{6.15}$$

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

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

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

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 $PX|C_i)$, 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 the above probabilities, we obtain

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

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

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

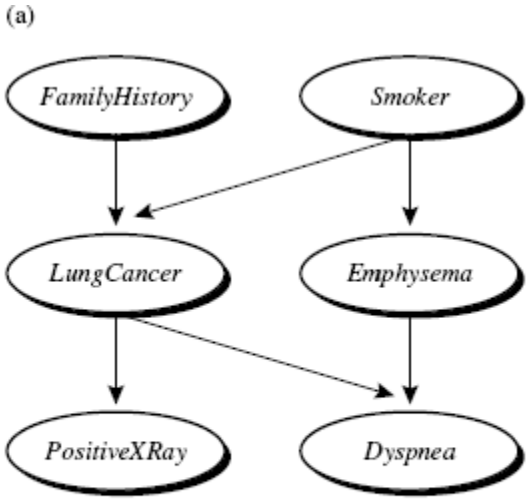
Therefore, the naïve Bayesian classifier predicts *buys_computer = yes* for tuple X. ■

3) Bayesian Belief Networks

The naïve Bayesian classifier makes the assumption of class conditional indepen dence, that is, given the class label of a tuple, the values of the attributes are assumed to be conditionally independent of one another. This simplifies computation . When the assumption holds true, then the naïve Bayesian classifier is the most accurate in comparison with all other classifiers. In practice, however, dependencies can exist between variables. Bayesian belief networks specify joint conditional probabilit y distributions. They allow class conditional independencies to be defined between subsets of variables. They provide a graphic al model of causal relationships, on which learning can be performed. Trained Bayesian belief networks can be used for classification. Bayesian belief networks are also known as belief networks, Bayesian networks, and probabilistic networks. Fo r brevity, we will

refer to them as belief networks.

A belief network is defined by two components —a *directed acyclic graph* and a set of *conditional probability tables* (Figure 6.11). Each node in the directed acyclic graph represents a random variable. The variables may be discrete or continuous-valued. They may correspond to actual attributes given in the data or to ―hidden variables‖ believed to form a relationship (e.g., in the case of medical data, a hidden variable may indicate a syndrome, representing a number of symptoms that, together, characterize a specific disease). Each arc represents a probabilistic dependence. If an arc is drawn from a node *Y* to a node *Z*, then *Y* is a parent or immediate predecessor of *Z*, and *Z* is a descendant of *Y*. *Each variable is conditionally independent of its non descendants in the graph, given its parents.*



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

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

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

$$P(LungCancer = yes \mid FamilyHistory = yes, Smoker = yes) = 0.8$$
$$P(LungCancer = no \mid FamilyHistory = no, Smoker = no) = 0.9$$

that

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

Let $\boldsymbol{X}$ = (*x*1, : : : , *xn*) be a data tuple described by the variables or attributes *Y*1, : : : , *Yn*, respectively. Recall that each variable is conditionally independent of its non descendants in the network graph, given its parents. This allows the network to provide a complete representation of the existing joint probability distribution with the following equation:

RULE-BASED CLASSIFICATION

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. We then study ways in which they can be generated, either from a decision tree or directly from the training data using a *sequential covering algorithm.*

1) 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 expression of the form

IF *condition* THEN *conclusion.*

An example is rule *R*1,

R1: IF *age = youth* AND *student = yes* THEN *buys computer = yes.*

The ―IF‖-part (or left-hand side)of a rule is known as the rule antecedent or precondition. The ―THEN‖-part (or right- hand side) is the rule consequent. In the rule antecedent, the condition consists of one or more *attribute tests* (such as *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

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

If the condition (that is, all of 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 *ncovers* be the number of tuples covered by *R*; *ncorrect* 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

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

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

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

2) Rule Extraction from a Decision Tree

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 understand how decision trees work and they are known for their accuracy. Decision trees can become large and difficult to interpret. In this subsection, we look at how to build a 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).

Extracting classification rules from a decision tree. The decision tree of Figure 6.2 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 6.2 are

*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*

■

Classification by Backpropagation

"What is backpropagation?" Backpropagation is a neural network learning algorithm. The field of neural networks was originally kindled by psychologists and neurobiologists who sought to develop and test computational analogues 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 co rrect 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 applications where this is feasible. They require a number of parameters that are typically best determined empirically, s uch as the network topology or
—structure.‖ Neural networks 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.

1) 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 shown in Figure 6.15.

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 —neuron like‖ 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 lay er, 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 6.15 has two layers

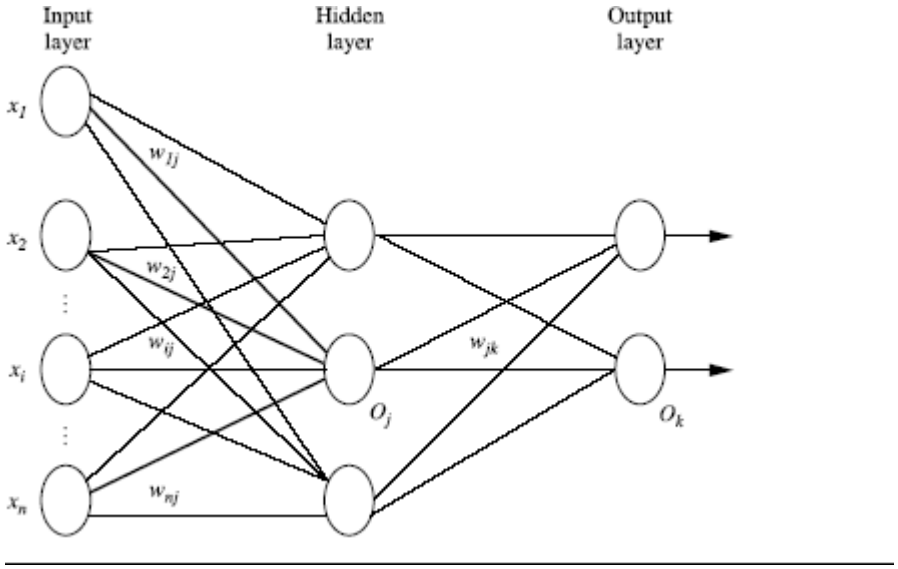


Figure 6.15 A multilayer feed-forward neuralnetwork.

2) Defining a Network Topology

"How can I design the topology of the neural network?" 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 f*a*0, *a*1, *a*2g, 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*=*a0*, then I0 is set to 1. If *A* = *a*1, *I*1 is set to 1, and so on. Neural networks can be used for both classification (to predict the class label of a given tuple) or prediction (to predict a continuous-valued output). For classification, 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.

3) 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 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, that is, 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 6.16. 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 below.

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

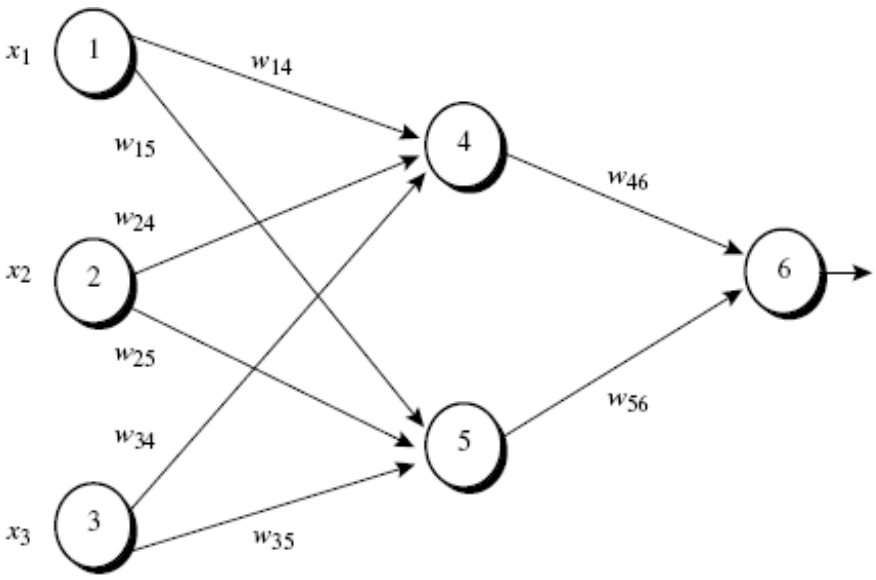
Input:

- D, a data set consisting of the training tuples and their associated target values;
- l, the learning rate;
- *network*, a multilayer feed-forward network.

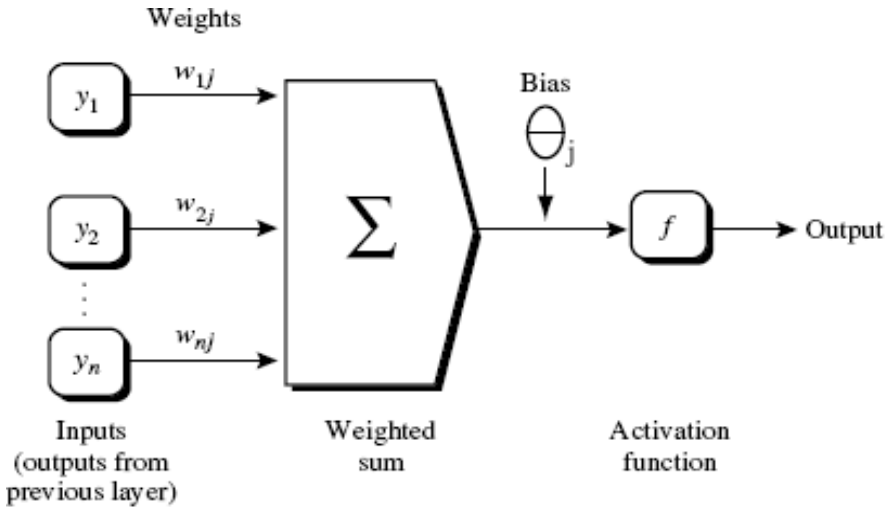
Output: A trained neural network.

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 to the 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 θ_j in *network* {
(19) $\Delta \theta_j = (l) Err_j$; // bias increment
(20) $\theta_j = \theta_j + \Delta \theta_j$; } // bias update
(21) } }



An example of a multilayer feed-forward neural network.



A hidden or output layer unit j: The inputs to unit j are outputs from the previous layer. These are multiplied by their corresponding weights in order 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. (For ease 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 $(x_1, x_2, \ldots, x_n)$.)

Support Vector Machines

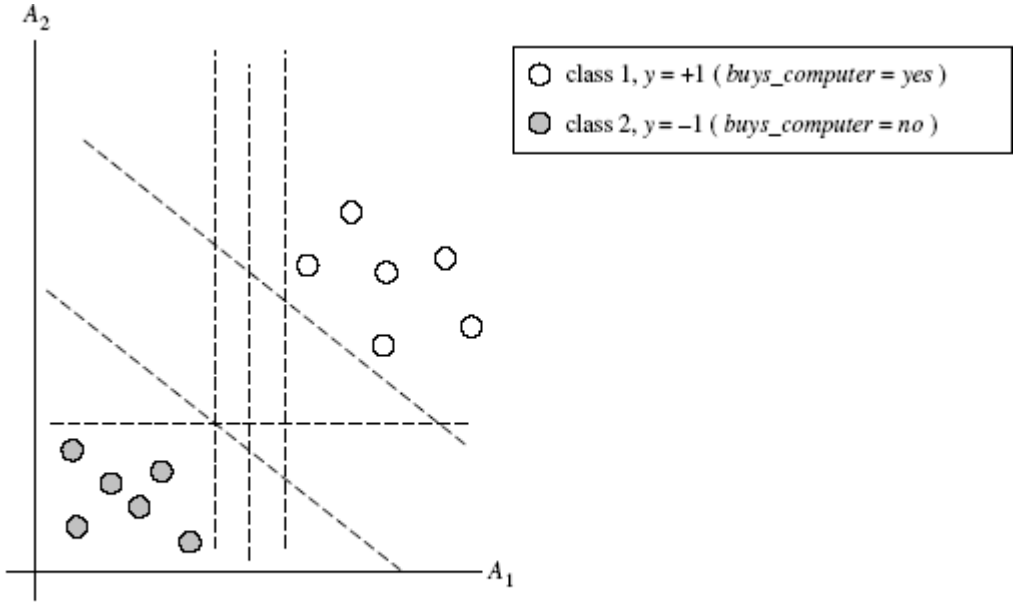
We study Support Vector Machines, a promising new method for the classification of both linear and nonlinear data. In a nutshell, a support vector machine (or SVM) is an algorithm that works as follows. It uses a nonlinear mapping to transform the original training data into a higher dimension. Within this new dimension, it searches for the linear optimal separating hyperplane (that is, a ―decision boundary‖ separating the tuples of one class from another). With an appropriate nonlinear mapping to a sufficiently high dimension, 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 further below.

"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 colleagues Bernhard Boser and Isabelle Guyon, although the groundwork for SVMs has been around since the 1960s (including early work by Va pnik 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 complex nonlinear decision boundaries. They aremuch less prone to overfitting than othermethods. The support vectors found also provide a compact description of the learned model. SVMs can be used for prediction as well as classification. They have been applied to a number of areas, including handwritten digit rec ognition, object recognition, and speaker identification, as well as benchmark time-series prediction tests.

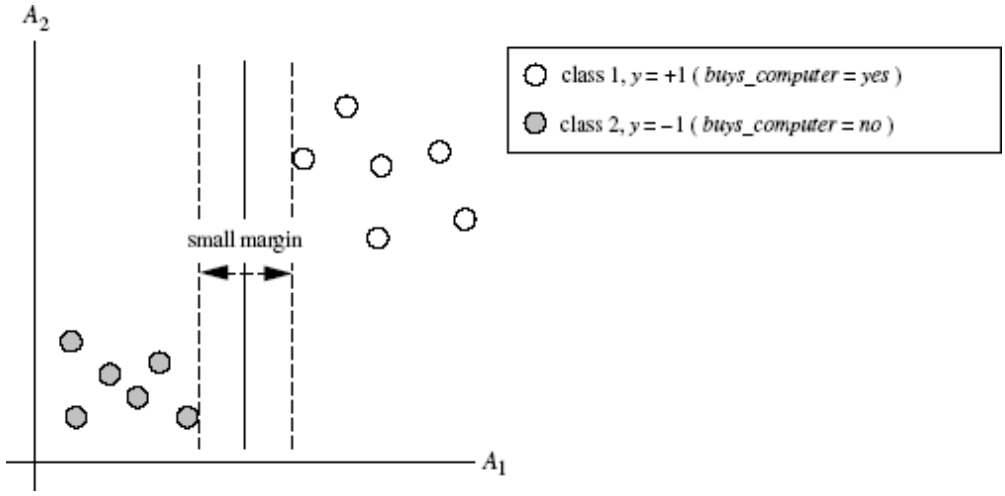
1) The Case When the Data Are Linearly Separable

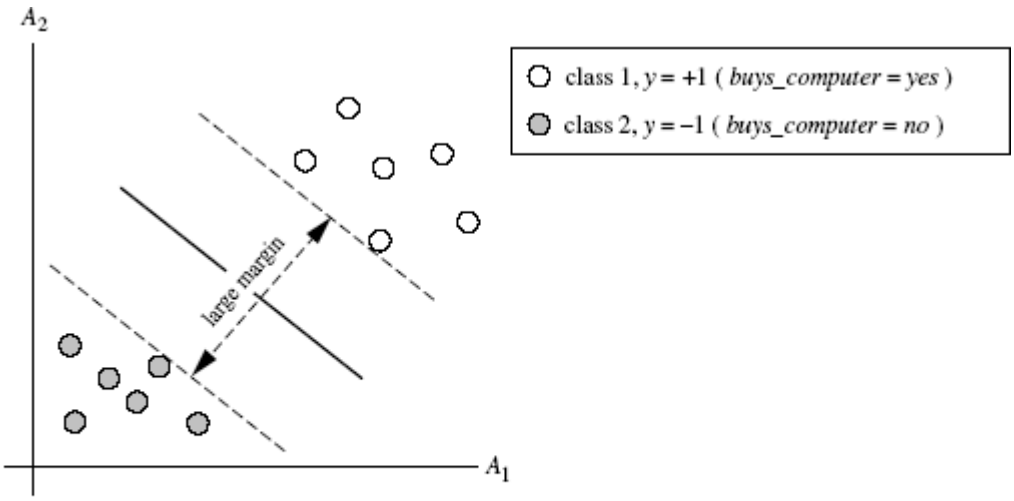
An SVM approaches this problem by searching for the maximum marginal hyperplane. Consider the below Figure , which shows two possible separating hyperplanes and their associated margins. Before we get into the definition of margins, let's take an intuitive look at this figure. Both

hyperplanes can correctly classify all of 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. 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 parallelto 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.



The 2-D training data are linearly separable. There are an infinite number of (possible) separating hyperplanes or "decision boundaries." Which one is best?





Here we see just two possible separating hyperplanes and their associated margins. Which one is better? The one with the larger margin should have greater generalization accuracy.

2) The Case When the Data Are Linearly Inseparable

We learned about linear SVMs for classifying linearly separable data, but what if the data are not linearly separable no straight line cajn 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 *nonlinearly 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 trans form 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 describe further below. 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.

Associative Classification: Classification by Association Rule Analysis

Frequent patterns and their corresponding association or correlation rules characterize interesting relationships between attribute conditions and class labels, and thus have been recently used for effective classification. Association rules show strong associations between attribute-value pairs (or *items*) that occur frequently in a given data set. Association rules are commonly used to analyze the purchasing patterns of customers in a store. Such analysis is useful in many decision-making processes, such as product placement, catalog design, and cross-marketing.

The discovery of association rules is based on *frequent itemset mining*. Many methods for frequent itemset mining and the generation of association rules were described in Chapter 5. In this

sectio n, we look at associative classification, where association rules are generated and analyzed for use in classification. The general idea is that we can search for strong associations between frequent patterns (conjunctions of attribute-value pairs) and class labels. Because association rules explore highly confident associations among multiple attributes, this approach may overcome some constraints introduced by decision - tree induction, which considers only one attribute at a time. In many studies, associative classification has been found to be more accurate than some traditional classification methods, such as C4.5. In particular, CMAR, and CPAR.

Lazy Learners (or Learning from Your Neighbors)

We study three main methods: CBA,

The classification methods discussed so far in this chapter—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.

1) *k*-Nearest-Neighbor Classifiers

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

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

―Closeness‖ is defined in terms of a distance metric, such as Euclidean distance. The Euclidean distance between two points or tuples, say, $\boldsymbol{X}1 = (x11, x12, : : : , x1n)$ and $\boldsymbol{X}2 = (x21, x22, : : : , x2n)$, is

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

2) 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 subgraphs within the new case. The case-based reasoner tries to combine the solutions of the neighboring training cases in order to propose a solution for the new case. If incompatibilities arise with the individual solutions, then backtra cking to search for other solutions may be necessary. The case-based reasoner may employ background knowledge and problem - solving strategies in order to propose a feasible combined solution.

Important 16 mark Questions in Unit-

- **Explain Associations Rule Mining**

  Or

  Explain how to generate strong association rule mining

  Or

  Explain the mining methods for association rule generation

  Or

  Explain Apriori and FP-Growth Method with Example.

- **Explain Mining Various Kinds of Association Rules**
- **Explain decision tree induction (Attribute Oriented Induction)**
- **Explain Bayesian Classification.**
- **Explain Rule Based Classification**
- **Explain Classification by Backpropagation**
- **Write short notes on Support Vector Machines and Lazy Learners**
- **Write short notes on Prediction**

.