UNIT-V-I

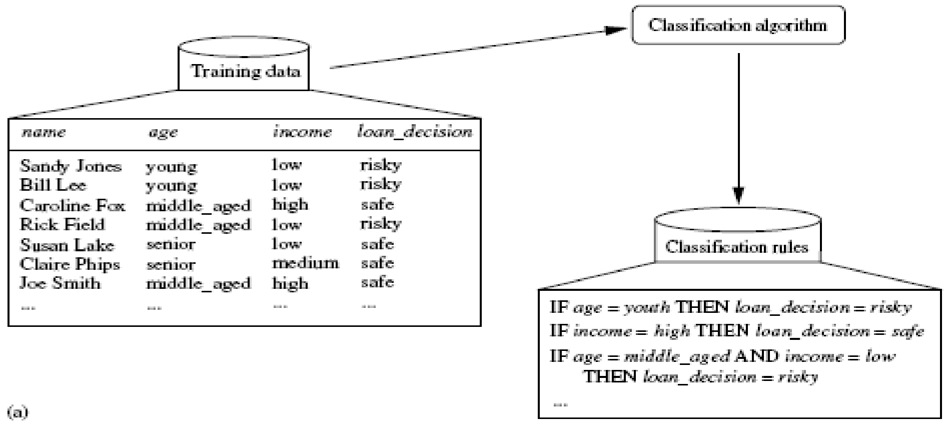
Classification and Prediction

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 *All Electronics* 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

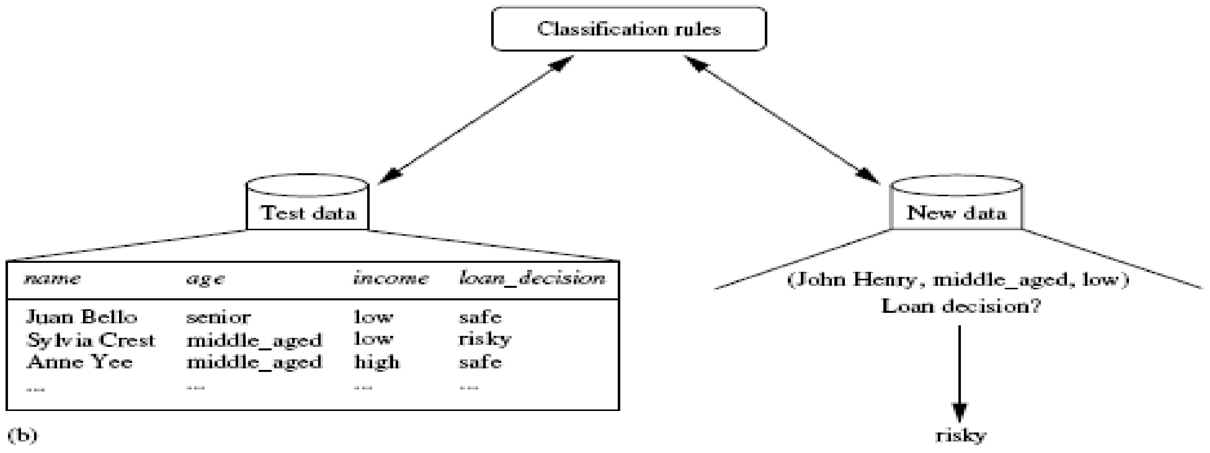


name	*age*	*income*	*loan_decision*
Sandy Jones	young	low	risky
Bill Lee	young	low	risky
Caroline Fox	middle_aged	high	safe
Rick Field	middle_aged	low	risky
Susan Lake	senior	low	safe
Claire Phips	senior	medium	safe
Joe Smith	middle_aged	high	safe
...	...	...	...

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

(a)

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



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

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 selection4 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)

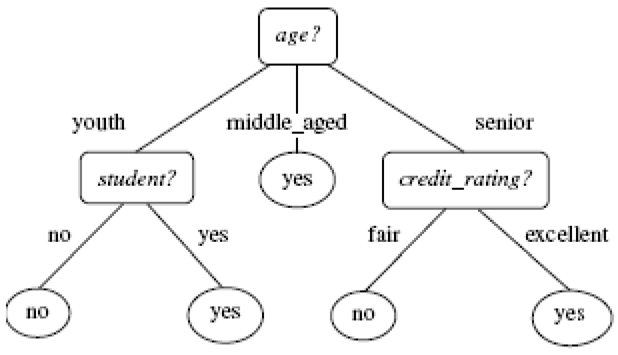
Comparing Classification and Prediction Methods

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.



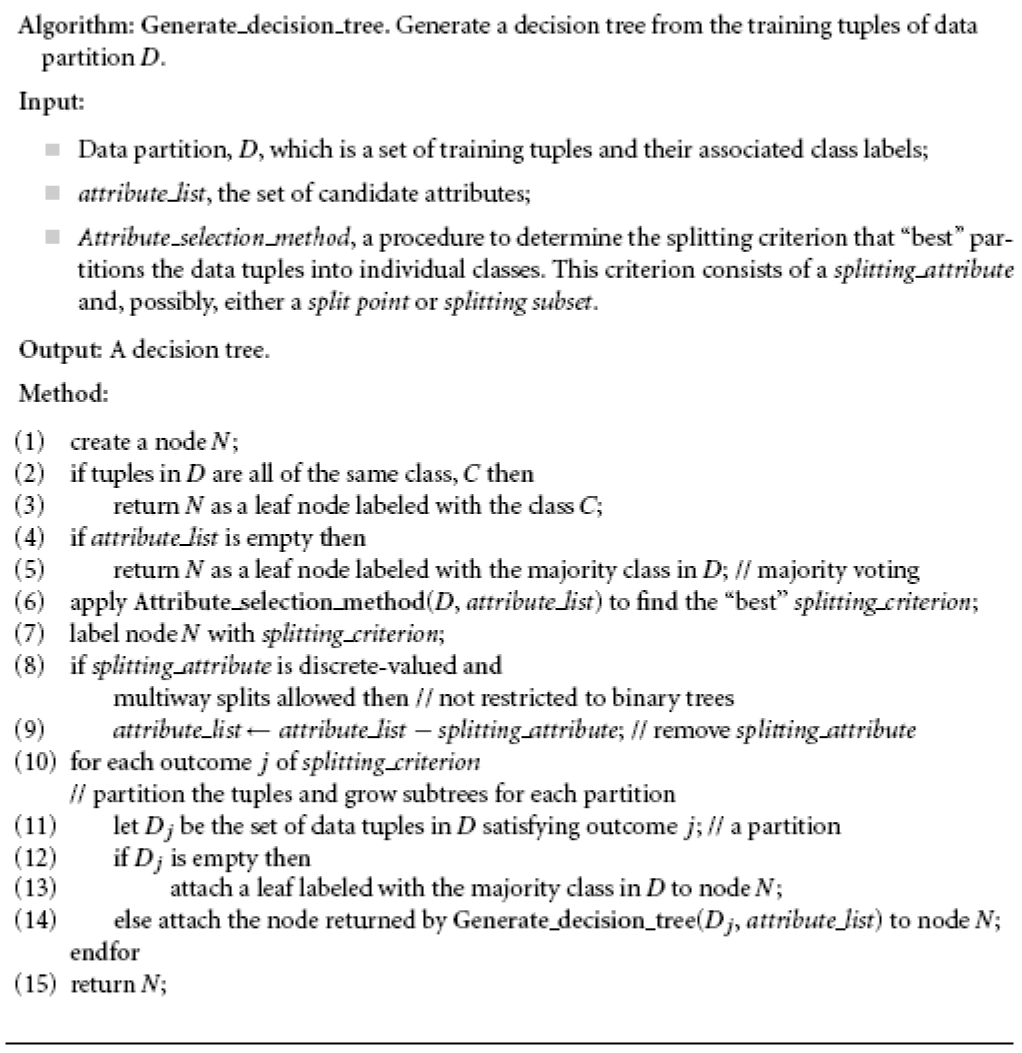
A decision tree for the concept *buys_computer*, indicating whether a customer at *AllElectronics* is likely to purchase a computer. Each internal (nonleaf) node represents a test on an attribute. Each leaf node represents a class (either *buys_computer* = *yes* or *buys_computer* = *no*).

A typical decision tree is shown in Figure. It represents the concept *buys computer*, that is, it predicts whether a customer at *All Electronics* 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 according to class. This procedure employs an attribute selection measure, such as information gain or the gini index. Whether the tree is strictly binary is generally driven by the attribute selection measure. Some attribute selection measures, such as the gini index, enforce the resulting tree to be binary. Others, like information gain, do not, therein allowing multi way 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 $D_j$ be the set of data tuples in $D$ satisfying outcome $j$; // a partition
(12)     if $D_j$ is empty then
(13)         attach a leaf labeled with the majority class in $D$ to node $N$;
(14)     else attach the node returned by Generate_decision_tree($D_j$, attribute_list) to node $N$;
     endfor
(15) return $N$;
```

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, {*a*1, *a*2, : : : , *av*}, based on the training data.

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

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

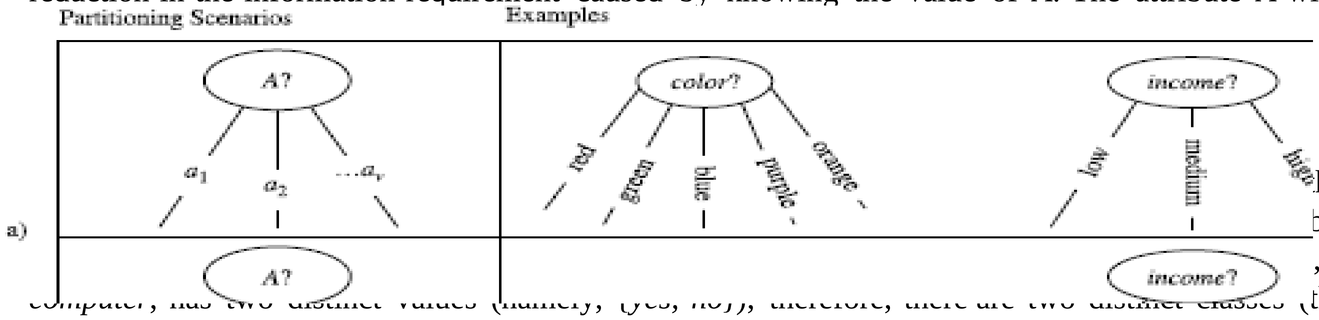
Information gain

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



he *All*

bute is

, *buys*

computer, has two distinct values (namely, {*yes*, *no*}); therefore, there are two distinct classes (that is, *m* = 2). Let class *C*1 correspond to *yes* and class *C*2 correspond to *no*. There are nine tuples of class *yes* and five tuples of class *no*. A (root) node *N* is created for the tuples in *D*. To find the splitting criterion for these tuples, we must compute the information gain of each attribute. We first use Equation (6.1) to compute the expected information needed to classify a tuple in *D*:

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

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

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

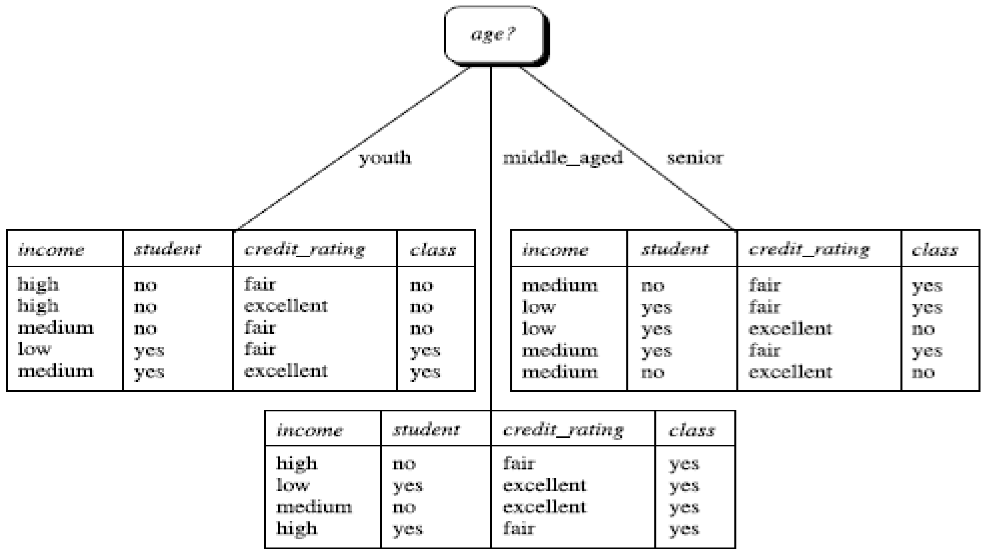
The expected information needed to classify a tuple in *D* if the tuples are partitioned according to *age* is

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

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

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

Similarly, we can compute *Gain*(*income*) = 0.029 bits, *Gain*(*student*) = 0.151 bits, and *Gain*(*credit rating*) = 0.048 bits. Because *age* has the highest information gain among the attributes, it is selected as the splitting attribute. Node *N* is labeled with *age*, and branches are grown for each of the attribute's values. The tuples are then partitioned accordingly, as shown in Figure 6.5. Notice that the tuples falling into the partition for *age* = *middle aged* all belong to the same class. Because they all belong to class *"yes,"* a leaf should therefore be created at the end of this branch and labeled with *"yes."* The final decision tree returned by the algorithm is shown in Figure 6.5.



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.

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

selected neural network classifiers. Bayesian classifiers have also exhibited high accuracy 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*(***X***j*H*), and *P*(***X***) may be estimated from the given data, as we shall see below. Bayes' theorem is useful in that it provides a way of calculating the posterior probability, *P*(*H*j***X***), from *P*(*H*), *P*(***X***j*H*), and *P*(***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 m

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 com

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

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

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 (Equation (6.10)),

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

$P(X|C_i)$

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

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

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

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

Using the above probabilities, we obtain

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

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. ■

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).

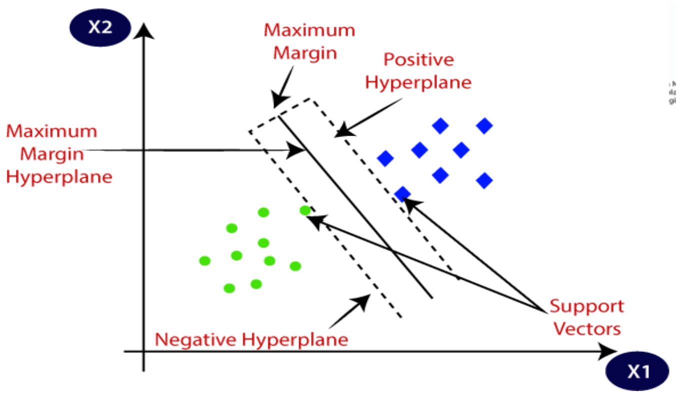
Extracting classification rules from a decision tree. The decision tree of Figure 6.2 can be converted to classification IF-THEN rules by tracing the path from the root node to each leaf node in the tree. The rules extracted from Figure 6.2 are

*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*

■

Support Vector Machines

- Support Vector Machine or SVM is one of the most popular Supervised Learning algorithms, which is used for Classification as well as Regression problems. However, primarily, it is used for Classification problems in Machine Learning.
- The goal of the SVM algorithm is to create the best line or decision boundary that can segregate n-dimensional space into classes so that we can easily put the new data point in the correct category in the future. This best decision boundary is called a hyperplane.
- SVM chooses the extreme points/vectors that help in creating the hyperplane. These extreme cases are called as support vectors, and hence algorithm is termed as Support Vector Machine.
- Consider the below diagram in which there are two different categories that are classified using a decision boundary or hyperplane:



➢ Here, the Solid line is called hyperplane, and the other two lines are known as boundary lines.

Example:

- SVM can be understood with the example that we have used in the KNN classifier.
- Suppose we see a strange cat that also has some features of dogs, so if we want a model that can accurately identify whether it is a cat or dog, so such a model can be created by using the SVM algorithm.

- We will first train our model with lots of images of cats and dogs so that it can learn about different features of cats and dogs, and then we test it with this strange creature .
- So as support vector creates a decision boundary between these two data (cat and dog) and choose extreme cases (support vectors), it will see the extreme case of cat and dog.
- SVM algorithm can be used for Face detection, image classification, text categorization, etc.
- On the basis of the support vectors, it will classify it as a cat. Consider the below diagram:

Types of SVM

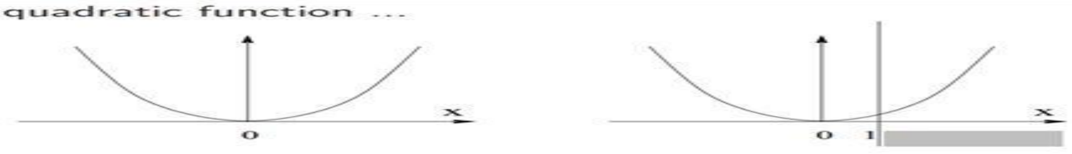
SVM can be of two types:

- Linear SVM: Linear SVM is used for linearly separable data, which means if a dataset can be classified into two classes by using a single straight line, then such data is termed as linearly separable data, and classifier is used called as Linear SVM classifier.
- Non-linear SVM: Non-Linear SVM is used for non-linearly separated data, which means if a dataset cannot be classified by using a straight line, then such data is termed as non-linear data and classifier used is called as Non-linear SVM classifier.

SVM PRIMAL AND DUAL:

- SVM is defined in two ways one is dual form and the other is the primal form. Both get the same optimization result but the way they get it is very different. Before we delve deep into mathematics let me tell you which one is used when. Primal mode is preferred when we don’t need to apply kernel trick to the data and the dataset is large but the dimension of each data point is small. Dual form is preferred when data has a huge dimension and we need to apply the kernel trick

A simple quadratic function ...



- Example of **unconstrained** optimization problem (solution: $x = 0$)

$$\min_x x^2$$

- Example of **constrained** optimization problem (active constraint)

$$\min_x x^2 \text{ such that } x \geq 1$$

- Optimization problem (primal problem):

$$\min_{w,b} \frac{1}{2} w^T w \quad \text{s.t.} \quad y_k[w^T x_k + b] \geq 1, \quad k = 1, ..., N$$

i.e. **maximize margin** and classify all training data correctly.

- Lagrangian

$$\mathcal{L}(w, b; \alpha) = \frac{1}{2} w^T w - \sum_{k=1}^{N} \alpha_k \{y_k[w^T x_k + b] - 1\}$$

with Lagrange multipliers $\alpha_k \geq 0$ for $k = 1, ..., N$.

Lazy Learners (or Learning from Your Neighbors)

- **Lazy vs. eager learning** – Eager learning e.g. decision tree induction, Bayesian classification, rule based classification Given a set of training set, constructs a classification model before receiving new (e.g., test) data to classify
- **Lazy Learners** – Lazy learning e.g., k-nearest-neighbor classifiers, case-based reasoning classifiers Simply stores training data (or only minor processing) and waits until it is given a new instance
- **Lazy**: less time in training but more time in predicting
  Lazy learners store training examples and delay the processing

(“lazy evaluation”) until a new instance must be classified

- **Accuracy** – Lazy method effectively uses a richer hypothesis space since it uses many local linear functions to form Lazy Learners its implicit global approximation to the target function – Eager: must commit to a single hypothesis that covers the entire instance space

Example Problem: Face Recognition
We have a database of (say) 1 million face images We are given a new image and want to find the most similar images in the database
Represent faces by (relatively) invariant values, Lazy Learners e.g., ratio of nose width to eye width
Each image represented by a large number of numerical features
Problem: given the features of a new face, find those in the DB that are close in at least ¾ (say) of the features Introduction
Typical approaches – k-nearest neighbor approach Instances represented as points in a Euclidean space. – Case-based reasoning Uses symbolic representations and knowledge-based inference