UNIT-III

Classification: Classification and Prediction – Basic concepts–Decision tree induction–Bayesian classification, Rule–based classification, Lazy learner.

I. Classification and Prediction:

- Classification and prediction are two forms of data analysis that can be used to extractmodels describing important data classes or to predict future data trends.
- Classification predicts categorical (discrete, unordered) labels, *prediction* models continuous valued functions.
- For example, we can build a classification model to categorize bank loan applications as either safe or risky, or a prediction model to predict the expenditures of potential customers on computer equipment given their income and occupation.
- A predictor is constructed that predicts a continuous-valued function, or ordered value, as opposed to a categorical label.
- Regression analysis is a statistical methodology that is most often used for numeric prediction.
- Many classification and prediction methods have been proposed by researchers in machine learning, pattern recognition, and statistics.
- Most algorithms are memory resident, typically assuming a small data size. Recent data mining research has built on such work, developing scalable classification and prediction techniques capable of handling large disk-resident data.

Preparing the Data for Classification and Prediction:

The following preprocessing steps may be applied to the data to help improve the accuracy, efficiency, and scalability of the classification or prediction process.

(i) Data cleaning:

- This refers to the preprocessing of data in order to remove or reduce *noise* (by applying smoothing techniques) 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.

(ii) 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 ofthe two could be removed from further analysis.
- A database may also contain *irrelevant* attributes. Attribute subset selection can be usedin 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. Such analysis can help improve classification efficiency and scalability.

(iii) 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 to +1 or 0 to 1.
- The data can also be transformed by *generalizing* it to higher-level concepts. Concept hierarchies may be used for this purpose. This is particularly useful for continuous valued attributes.
- For example, numeric values for the attribute *income* can be generalized to discrete ranges, such as *low, medium*, and *high*. Similarly, categorical attributes, like *street*, can be generalized to higher-level concepts, like *city*.
- Data can also be reduced by applying many other methods, ranging from wavelet transformation and principle components analysis to discretization techniques, suchas binning, histogram analysis, and clustering.

Comparing Classification and Prediction Methods:

- **Accuracy:**
  - The accuracy of a classifier refers to the ability of a given classifier to correctly predict the class label of new or previously unseen data (i.e., tuples without class label information).
  - The accuracy of a predictor refers to how well a given predictor can guess the value of the predicted attribute for new or previously unseen data.
- **Speed:**

  This refers to the computational costs involved in generating and using thegiven classifier or predictor.
- **Robustness:**

  This is the ability of the classifier or predictor to make correct predictionsgiven noisy data or data with missing values.
- **Scalability:**

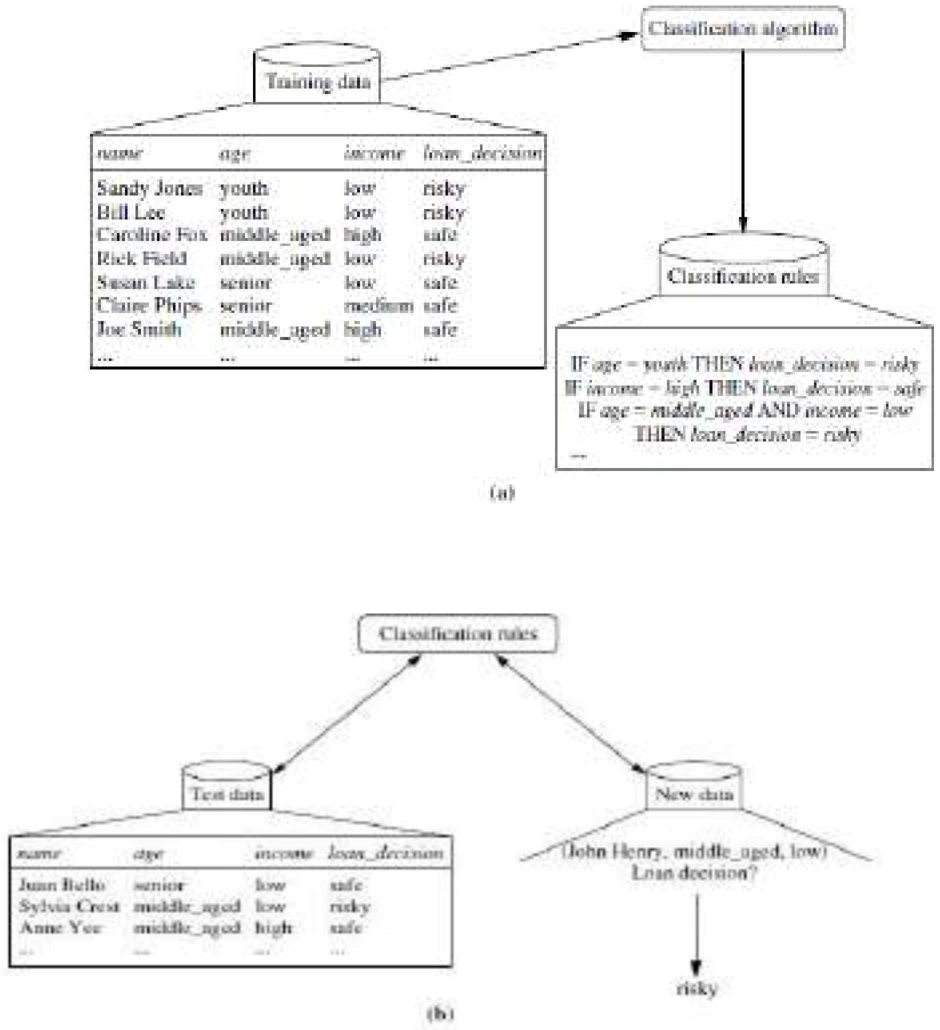
  This refers to the ability to construct the classifier or predictor efficiently given large amounts of data.
- **Interpretability:**
  - This refers to the level of understanding and insight that is providedby the classifier or predictor.
  - Interpretability is subjective and therefore more difficultto assess.

II. Basic Concepts

General Approach to Classification

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

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



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

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

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

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

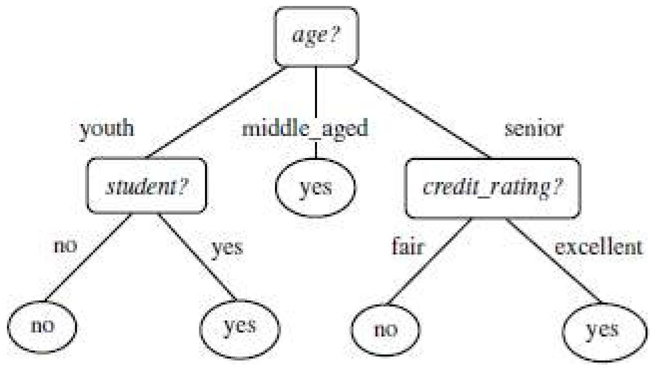
Because the class label of each training tuple is provided, this step is also known as supervised learning (i.e., the learning of the classifier is "supervised" in that it is told to which class each training tuple belongs). It contrasts with unsupervised learning (or clustering), in which the class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

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

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

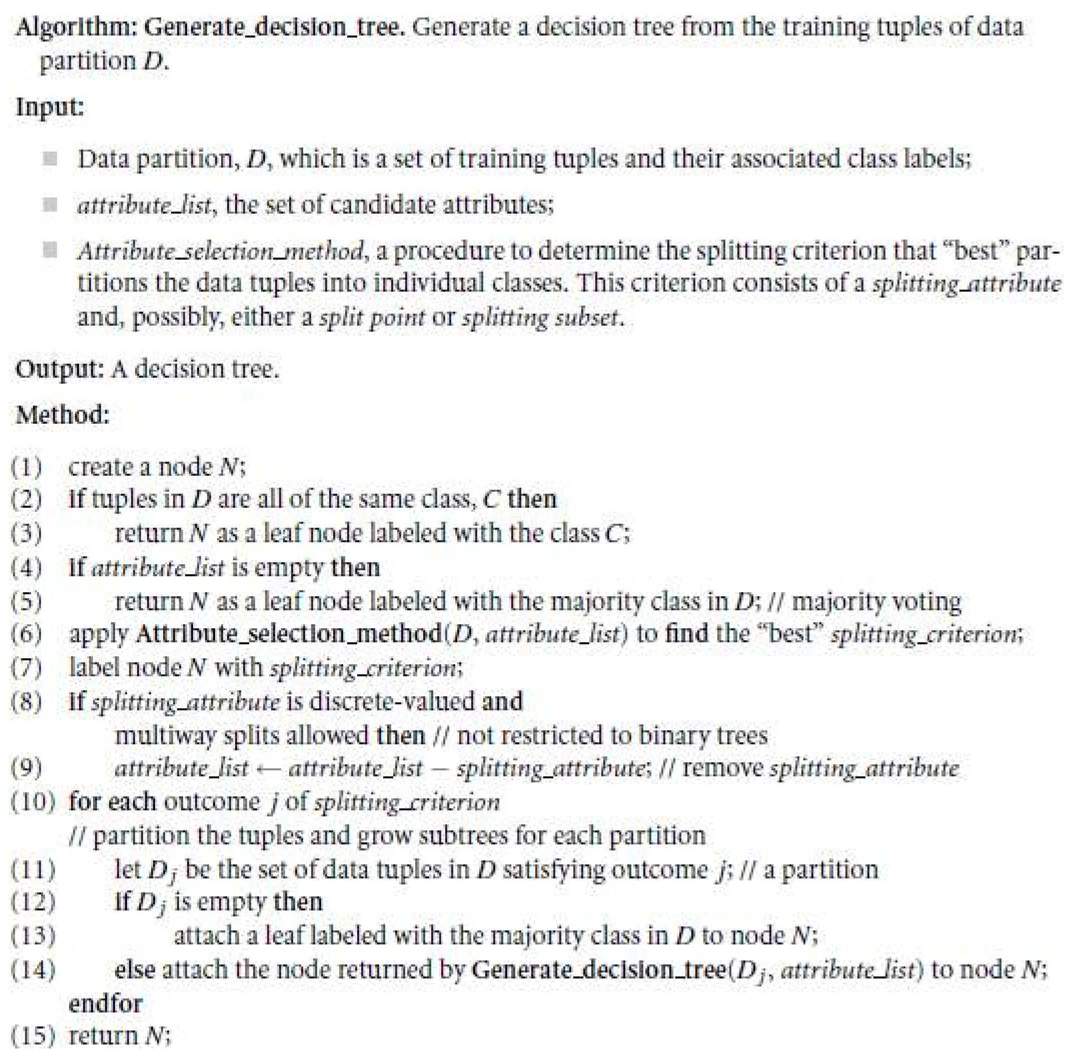
III Classification by Decision Tree Induction:

- Decision tree induction is the learning of decision trees from class-labeled training tuples.
- A decision tree is a flowchart-like tree structure,where
    - Each internal nodedenotes a test on an attribute.
    - Each branch represents an outcome of the test.
    - Each leaf node holds a class label.
    - The topmost node in a tree is the root node.



- The construction of decision tree classifiers does not require any domain knowledge or parameter setting, and therefore I appropriate for exploratory knowledge discovery.
- Decision trees can handle high dimensional data.
- Their representation of acquired knowledge in tree form is intuitive and generally easy to assimilate by humans.
- The learning and classification steps of decision tree induction are simple and fast. • In general, decision tree classifiers have good accuracy.
- 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.

Algorithm For Decision Tree Induction:



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;
```

The algorithm is called with three parameters:

- Data partition
- Attribute list
- Attribute selection method

- 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.

- The tree starts as a single node, *N*, representing the training tuples in *D*.
- If the tuples in *D* are all of the same class, then node *N* becomes a leaf and is labeled with that class .
- All of the terminating conditions are explained at the end of the algorithm.
- Otherwise, the algorithm calls Attribute selection method to determine the splittingcriterion.
- 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.

There are three possible scenarios.Let *A* be the splitting attribute. *A* has *v* distinct values,

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

1 A is discrete-valued:

- In this case, the outcomes of the test at node N correspond directly to the knownvalues of A. A branch is created for each known value, aj, of A and labeled with that value. A need not be
- considered in any future partitioning of the tuples.

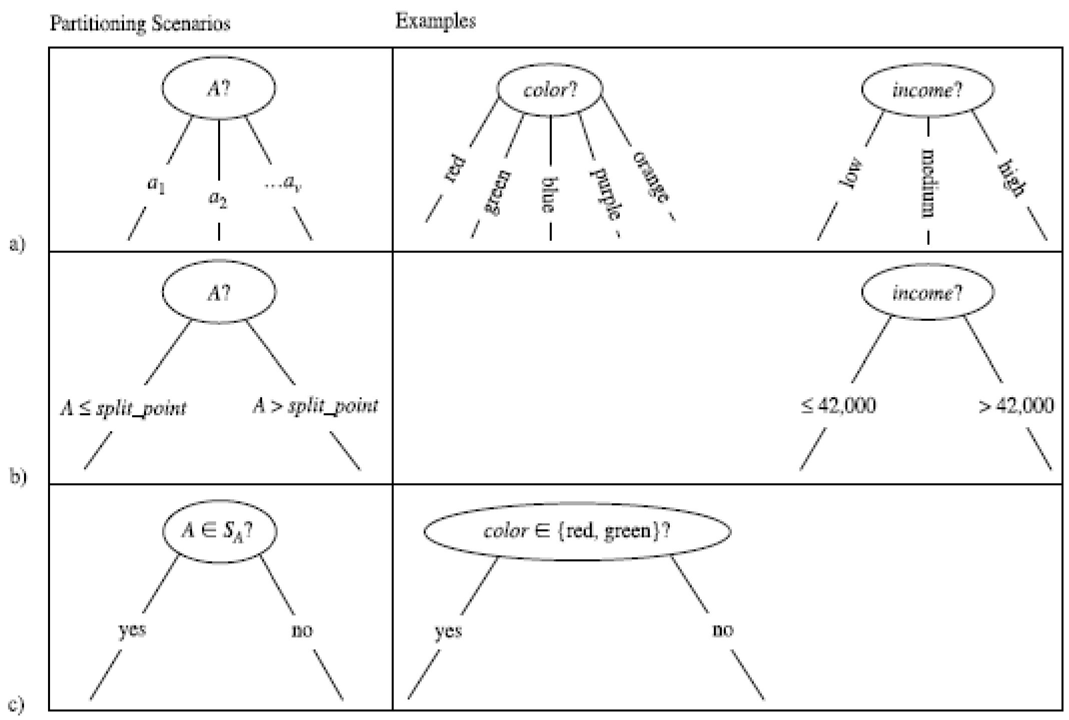
2 A is continuous-valued:

In this case, the test at node N has two possible outcomes, corresponding to the conditions A <=split point and A >split point, respectively
wheresplit point is the split-point returned by Attribute selection method as part of the splitting criterion.

3 A is discrete-valued and a binary tree must be produced:

The test at node N is of the form–A€SA?‖.

SA is the splitting subset for A, returned by Attribute selection methodas part of the splitting criterion. It is a subset of the known values of A.



(a) If A is Discrete valued (b)If A is continuous valued (c) IfA is discrete-valued and a binary tree must be produced:

IV Bayesian Classification:

- Bayesian classifiers are statistical classifiers.
- They can predictclass membership probabilities, such as the probability that a given tuple belongs toa particular class.
- Bayesian classification is based on Bayes' theorem.

Bayes' Theorem:

- Let X be a data tuple. In Bayesian terms, X is considered –evidence.‖and 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 hypothesisH holds given the ―evidence‖ or observed data tuple X.
- P(H|X) is the posterior probability, or a posteriori probability, of H conditioned on X.
- Bayes' theorem is useful in that it providesa way of calculating the posterior probability,

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

P(*H*|***X***), from *P*(*H*), *P*(***X***|*H*), and *P*(***X***).

Naïve Bayesian Classification:

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

1.Let D be a training set of tuples and their associated class labels. As usual, each tuple is represented by an n-dimensional attribute vector, X = (x1, x2, …,xn), depicting n measurements made on the tuple from n attributes, respectively, A1, A2, …, An.

2. Suppose that there are m classes, C1, C2, …, Cm. Given a tuple, X, the classifier will predict that X belongs to the class having the highest posterior probability, conditioned on X.

That is, the naïve Bayesian classifier predicts that tuple X belongs to the class Ci if and onlyif

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

Thus we maximize P(Cij**X**). The classCifor which P(Cij**X**) is maximized is called themaximum posteriori hypothesis. By Bayes' theorem

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

3.As P(X) is constant for all classes, only P(X|Ci)P(Ci) need be maximized. If the class

prior probabilities are not known, then it is commonly assumed that the classes are equally likely, that is, P(C1) = P(C2) = …= P(Cm), and we would therefore maximize P(X|Ci). Otherwise, we maximize P(X|Ci)P(Ci).Given data sets with many attributes, it would be extremely computationally expensiveto compute P(X|Ci). In order to reduce computation in evaluating P(X|Ci), the naive assumption of class conditional independence is made. This presumes that the values of the attributes areconditionally independent of one another, given the class label of the tuple. Thus,

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

We can easily estimate the probabilities *P*(*x*1|*Ci*), *P*(*x*2|*Ci*), : : : , *P*(*xn*|*Ci*) fromthe trainingtuples. For eachattribute, we look at whether the attribute is categorical or continuous-valued. Forinstance, to compute *P*(***X***|*Ci*), we consider the following:

- If $A_k$is categorical, then $P(x_k|Ci)$ is the number of tuples of class *Ci*in *D* havingthe value

$x_k$for A_k, divided by $|C_{i,D}|$ the number of tuples of class $C_i$in *D*.

- If $A_k$is continuous-valued, then we need to do a bit more work, but the calculationis pretty straightforward.

A continuous-valued attribute is typically assumed tohave a Gaussian distribution with a mean μ and standard deviation , defined by

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

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

4.In order to predict the class label of *X*, *P*(*X*j*Ci*)*P*(*Ci*) is evaluated for each class *Ci*.The classifier predicts that the class label of tuple *X* is the class *Ci*if and only if

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

V .Rule-Based Classification

Using IF-THEN Rules for Classification

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

IF *condition* THEN *conclusion*.An example is rule *R*1,

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

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

*R*1: (*age* = *youth*) ^ (*student* = *yes*) ⇒(*buys_computer* = *yes*).

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

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

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

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

of*R* as

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

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

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

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

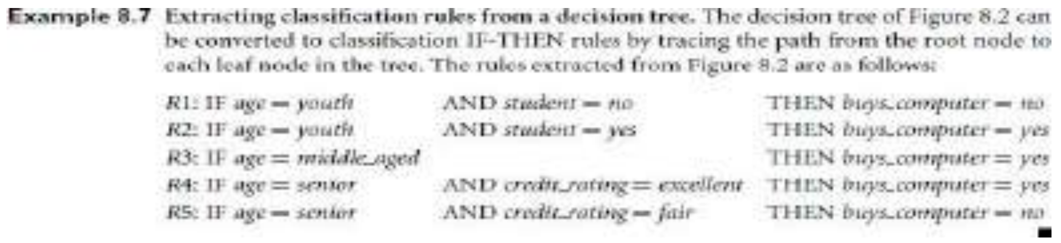
We would like to classify ***X*** according to *buys computer*. ***X*** satisfies *R*1, which triggers the rule.

If *R*1 is the only rule satisfied, then the rule **fires** by returning the class prediction for ***X***. Note that triggering does not always mean firing because there may be more than one rule that is satisfied! If more than one rule is triggered, we have a potential problem. What if they each specify a different class? Or what if no rule is satisfied by ***X***?We tackle the first question. If more than one rule is triggered, we need a **conflict resolution strategy** to figure out which rule gets to fire and assign its class prediction to ***X***. There are many possible strategies. We look at two, namely *size ordering* and *rule ordering*. The **size ordering** scheme assigns the highest priority to the triggering rule that has the "toughest" requirements, where toughness is measured by the rule antecedent *size*. That is, the triggering rule with the most attribute tests is fired.The **rule ordering** scheme prioritizes the rules beforehand. The ordering may be *class-based* or *rule- based*. With **class-based ordering**, the classes are sorted in order of decreasing "importance" such as by decreasing *order of prevalence*. That is, all the rules for the most prevalent (or most frequent) class come first, the rules for the next prevalent class come next, and so on.With **rule-based ordering**, the rules are organized into one long priority list, according to some measure of rule quality, such as accuracy, coverage, or size (number of attribute tests in the rule antecedent), or based on advice from domain experts. When rule ordering is used, the rule set is known as a **decision list**.Note that in the first strategy, overall the rules are *unordered*. They can be applied in any order when classifying a tuple. That is, a disjunction (logical OR) is implied between each of the rules. Each rule represents a standalone nugget or piece of knowledge. This is in contrast to the rule ordering (decision list) scheme for which rules must be applied in the prescribed order so as to avoid conflicts. Each rule in a decision list implies the negation of the rules that come before it in the list. Hence, rules in a decision list are more difficult to interpret. Now that we have seen how we can handle conflicts, let's go back to the scenario where there is no rule satisfied by ***X***. How, then, can we determine the class label of ***X***? In this case, a fallback or **default rule** can be set up to specify a default class, based on a training set. This may be the class in majority or the majority class of the tuples that were not covered by any rule. The default rule is evaluated at the end, if and only if no other rule covers ***X***.

Rule Extraction from a Decision Tree

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

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



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:

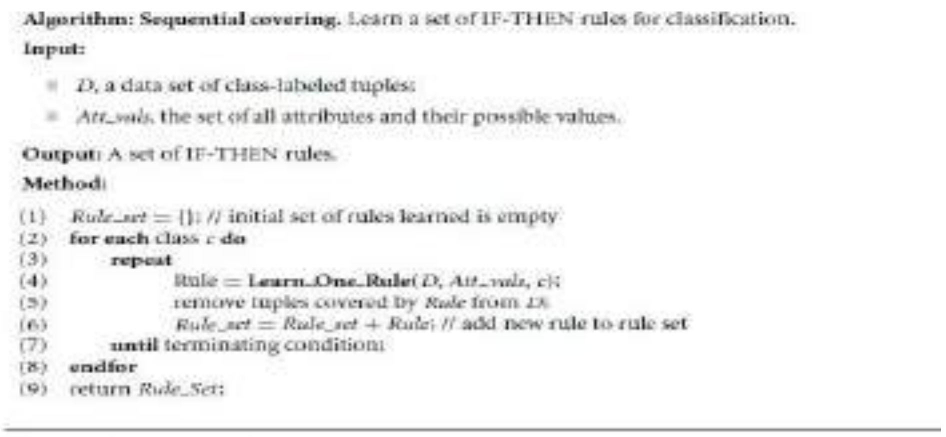
R1: IF *age* = *youth*	AND *student* = *no*	THEN *buys_computer* = *no*
R2: IF *age* = *youth*	AND *student* = *yes*	THEN *buys_computer* = *yes*
R3: IF *age* = *middle_aged*		THEN *buys_computer* = *yes*
R4: IF *age* = *senior*	AND *credit_rating* = *excellent*	THEN *buys_computer* = *yes*
R5: IF *age* = *senior*	AND *credit_rating* = *fair*	THEN *buys_computer* = *no*

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

"How can we prune the rule set?" For a given rule antecedent, any condition that does not improve the estimated accuracy of the rule can be pruned (i.e., removed), thereby generalizing the rule. C4.5 extracts rules from an unpruned tree, and then prunes the rules using a pessimistic approach similar to its tree pruning method. The training tuples and their associated class labels are used to estimate rule accuracy. However, because this would result in an optimistic estimate, alternatively, the estimate is adjusted to compensate for the bias, resulting in a pessimistic estimate. Other problems arise during rule pruning, however, as the rules *will no longer be* mutually exclusive and exhaustive. For conflict resolution, C4.5 adopts a **class-based ordering scheme**. It groups together all rules for a single class, and then determines a ranking of these class rule sets. Within a rule set, the rules are not ordered.

Rule Induction Using a Sequential Covering Algorithm

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



Algorithm: Sequential covering. Learn a set of IF-THEN rules for classification.

Input:

- *D*, a data set of class-labeled tuples;
- *Att_vals*, the set of all attributes and their possible values.

Output: A set of IF-THEN rules.

Method:

```
(1)  Rule_set = {}; // initial set of rules learned is empty
(2)  for each class c do
(3)      repeat
(4)          Rule = Learn_One_Rule(D, Att_vals, c);
(5)          remove tuples covered by Rule from D;
(6)          Rule_set = Rule_set + Rule; // add new rule to rule set
(7)      until terminating condition;
(8)  endfor
(9)  return Rule_Set;
```

Figure 8.10 Basic sequential covering algorithm.

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

"How are rules learned?" Typically, rules are grown in a *general-to-specific* manner (Figure 8.11). We can think of this as a beam search, where we start off with an empty rule and then gradually keep appending attribute tests to it. We append by adding the attribute test as a logical conjunct to the existing condition of the rule antecedent. Suppose our training set, *D*, consists of loan application data. Attributes regarding each applicant include their age, income, education level, residence, credit rating, and the term of the loan. The classifying attribute is *loan decision*, which indicates whether a loan is accepted (considered safe) or rejected (considered risky). To learn a rule for the class "accept," we start off with the most general rule possible, that is, the condition of the rule antecedent is empty. The rule is IF THEN *loan_decision* = *accept*.We then consider each possible attribute test that may be added to the rule. These can be derived from the parameter *Att_vals*, which contains a list of attributes with their associated values. For example, for an

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

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

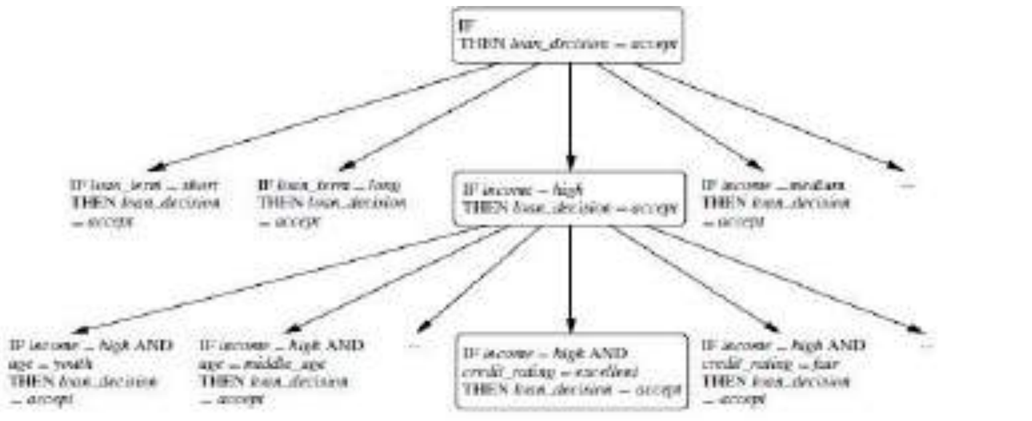

IF
THEN loan_decision = accept
IF loan_term = short THEN loan_decision = accept
IF loan_term = long THEN loan_decision = accept
IF income = high THEN loan_decision = accept
IF income = medium THEN loan_decision = accept
...
IF income = high AND age = youth THEN loan_decision = accept
IF income = high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

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

Each time we add an attribute test to a rule, the resulting rule should cover relatively more of the "accept" tuples. During the next iteration, we again consider the possible attribute tests and end up selecting *credit_rating* = *excellent*. Our current rule grows to become IF *income* = *high* AND *credit rating* = *excellent* THEN *loan decision* = *accept*. The process repeats, where at each step we continue to greedily grow rules until the resulting rule meets an acceptable quality level.

Rule Quality Measures

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

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

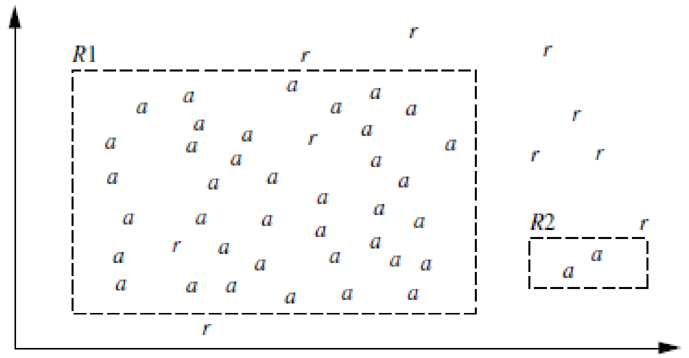


Figure 8.12 Rules for the class *loan_decision* = *accept*, showing *accept* (*a*) and *reject* (*r*) tuples.

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

Our current rule is *R*: IF *condition* THEN *class* = *c*. We want to see if logically ANDing a given attribute test to *condition* would result in a better rule.We call the new condition, *condition*0, where *R*0: IF *condition*0 THEN *class* = *c* is our potential new rule. In other words, we want to see if *R*0 is any better than *R*.Another measure is based on information gain and was proposed in **FOIL** (First Order Inductive Learner), a sequential covering algorithm that learns first-order logic rules. Learning first-order rules is more complex because such rules contain variables, whereas the rules we are concerned with in this section are propositional (i.e., variablefree). FOIL assesses the information gained by extending *condition*0 as It favors rules that have high accuracy and cover many positive tuples.

Rule Pruning

Learn One Rule does not employ a test set when evaluating rules. Assessments of rule quality as described previously are made with tuples from the original training data. These assessments are optimistic because the rules will likely overfit the data. That is, the rules may perform well on the training data, but less well on subsequent data. To compensate for this, we can prune the rules. rule is pruned by removing a conjunct (attribute test). We choose to prune a rule, *R*, if the pruned version of *R* has greater quality, as assessed on an independent set of tuples.FOIL uses a simple yet effective method. Given a rule, *R*,

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

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

VI .Lazy Learners (or Learning from Your Neighbors)

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 lastminute 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 training tuples. Unlike eager learning methods, lazy learners do less work when a training tuple is presented and more work when making a classification or 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 hyper polygonal shapes that may not be as easily describable by other learning algorithms (such as hyperrectangular shapes modeled by decision trees).

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, $X_1 = (x_{11}, x_{12}, \ldots, x_{1n})$ and $X_2 = (x_{21}, x_{22}, \ldots, x_{2n})$, is

$$dist(X_1, X_2) = \sqrt{\sum_{i=1}^{n} (x_{1i} - x_{2i})^2}. \qquad (9.22)$$

In other words, for each numeric attribute, we take the difference between the corresponding values of that attribute in tuple ***X***1 and in tuple ***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 example, can be used to transform a value *v* of a numeric attribute *A* to *v* 0 in the range [0, 1]
by computing

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

where *minA* and *maxA* are the minimum and maximum values of attribute *A*. For *k*-nearest-neighbor classification, the unknown tuple is assigned the most common class among its *k*-nearest neighbors. When *k* D 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 ***X***1 with that

in tuple ***X***2. If the two are identical (e.g., tuples ***X***1 and ***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 ***X***1 is blue but tuple ***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 ***X***1 and/or in tuple ***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 fromboth tuples ***X***1 and ***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* 0) 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* D 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* D 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 frompoor 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 j*D*j tuples and *k* D 1, then *O*.j*D*j) 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.*j*D*j/. Parallel implementation can reduce the running time to a constant, that is, *O*.1/, which is independent of j*D*j. 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 subgraphs 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.