Classification

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

What Is Classification?

A bank loans officer needs analysis of her data to learn which loan applicants are "safe" and which are "risky" for the bank. A marketing manager at *AllElectronics* needs data analysis to help guess whether a customer with a given profile will buy a new computer. A medical researcher wants to analyze breast cancer data to predict which one of three specific treatments a patient should receive.

In each of these examples, the data analysis task is classification, where a model or classifier is constructed to predict *class (categorical) labels*, such as "safe" or "risky" for the loan application data; "yes" or "no" for the marketing data; or "treatment A," "treatment B," or "treatment C" for the medical data.

In each of these examples, the data analysis task is classification, where a model or classifier is constructed to predict *class (categorical) labels*, such as "safe" or "risky" for the loan application data; "yes" or "no" for the marketing data; or "treatment A," "treatment B," or "treatment C" for the medical data.

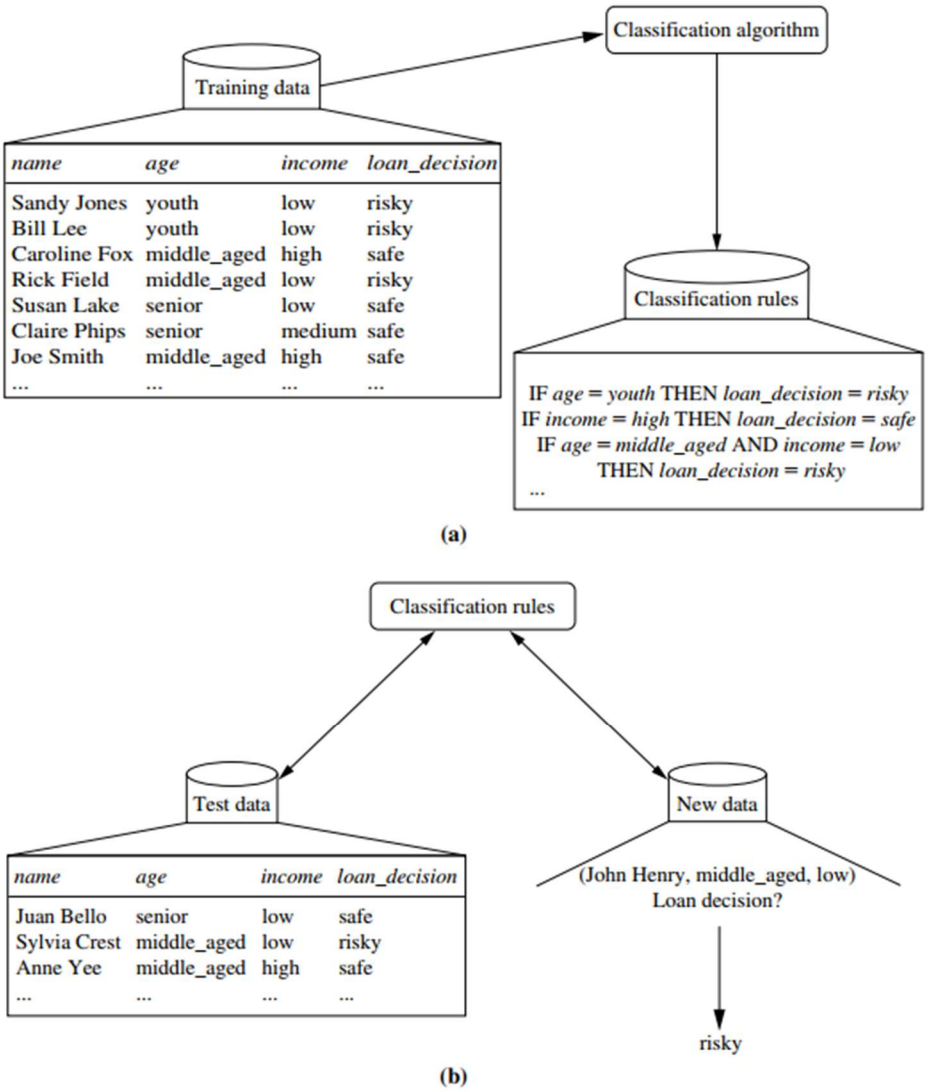
This data analysis task is an example of numeric prediction, where the model constructed predicts a *continuous-valued function*, or *ordered value*, as opposed to a class label. This model is a predictor. Regression analysis is a statistical methodology that is most often used for numeric prediction; Classification and numeric prediction are the two major types of prediction problems.

Data classification is a two-step process, consisting of a *learning step* and a *classification step.*

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, $A1, A2, \ldots\ldots, An$.

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* and are randomly sampled from the database under analysis. In the context of classification, data tuples can be referred to as *samples, examples, instances, data points*, or *objects*.



the class label of each training tuple is provided, this step is also known as *supervised learning* (i.e., the learning of the classifier is "supervised" in that it is told to which class each training tuple belongs). It contrasts with *unsupervised learning* (or *clustering*), in which the class label of each training tuple is not known, and the number or set of classes to be learned may not be known in advance.

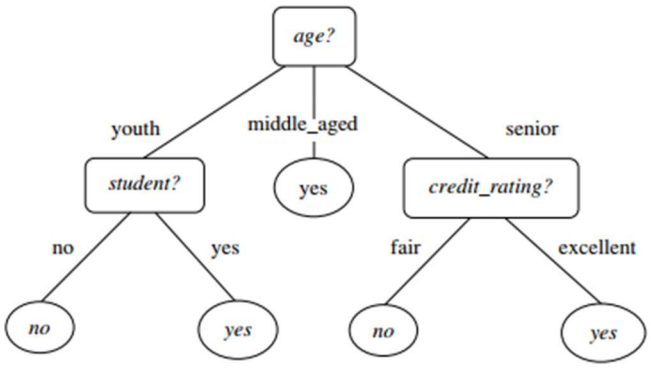
This first step of the classification process can also be viewed as the learning of a mapping or function, y = f *(*X*)*, that can predict the associated class label y of a given tuple X. the mapping is represented as classification rules that identify loan applications as being either safe or risky.

In the second step, the model is used for classification. First, the predictive accuracy of the classifier is estimated. If we were to use the training set to measure the classifier's accuracy, this estimate would likely be optimistic, because the classifier tends to overfit the data. 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. The associated class label of each test tuple is compared with the learned classifier's class prediction for that tuple.

Decision Tree Induction

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



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

Given a tuple, X, for which the associated class label is unknown, the attribute values of the tuple are tested against the decision tree. A path is traced from the root to a leaf node, which holds the class prediction for that tuple. Decision trees can easily be converted to classification rules.

The construction of decision tree classifiers does not require any domain knowledge or parameter setting, and therefore is appropriate for exploratory knowledge discovery. Decision trees can handle multidimensional data. Their representation of acquired knowledge in tree form is intuitive and generally easy to assimilate by humans. The learning and classification steps of decision tree induction are simple and fast. In general, decision tree classifiers have good accuracy. However, successful use may depend on the data at hand. Decision tree induction algorithms have been used for classification in many application areas such as medicine, manufacturing and production, financial analysis, astronomy, and molecular biology.
Decision trees are the basis of several commercial rule induction systems.

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

Classification and Regression Trees (*CART*), which described the generation of binary decision trees. ID3 and CART were invented independently of one another at around the same time, yet follow a similar approach for learning decision trees from training tuples. These two cornerstone algorithms spawned a flurry of work on decision tree induction.

ID3, C4.5, and CART adopt a greedy (i.e., nonbacktracking) approach in which decision trees are constructed in a top-down recursive divide-and-conquer manner. Most algorithms for decision tree induction also follow a top-down approach, which starts with a training set of tuples and their associated class labels. The training set is recursively partitioned into smaller subsets as the tree is being built.

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 multiway splits.

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;
```

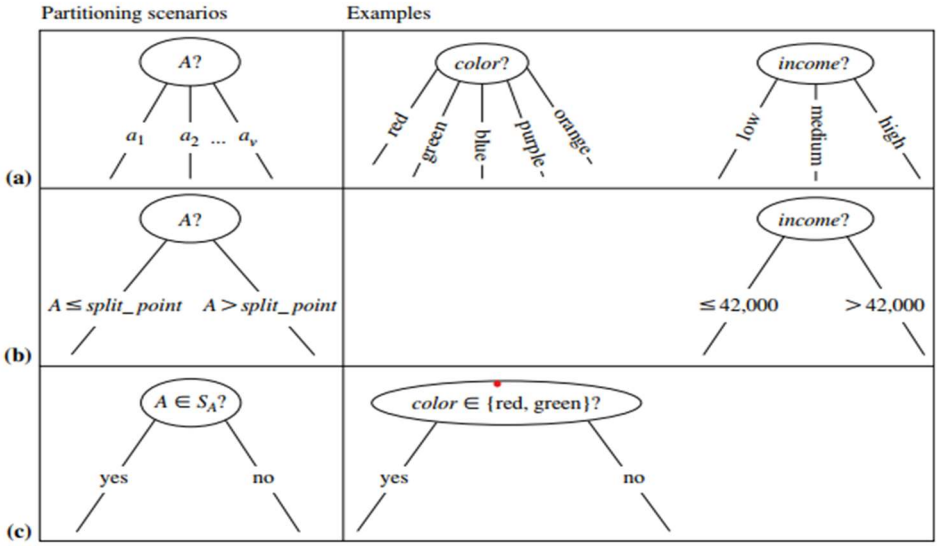
If the tuples in D are all of the same class, then node N becomes a leaf and is labeled with that class (steps 2 and 3). Note that steps 4 and 5 are terminating conditions. All terminating conditions are explained at the end of the algorithm.

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 the tuples in it belong to the same class. In other words, if we 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. 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 known values of *A*. A branch is created for each known value, *aj*, of *A* and labeled with that value. Partition *Dj* is the subset of class-labeled tuples in *D* having value *aj* of *A*.



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, where split

point is the split-point returned by Attribute selection method as part of the splitting criterion.

Two branches are grown from N and labeled according to the previous outcomes. The tuples are partitioned such that D1 holds the subset of class-labeled tuples in D for which A ≤ split point, while D2 holds the rest.

3. A is discrete-valued and a binary tree must be produced (as dictated by the attribute selection measure or algorithm being used): The test at node N is of the form "A *2* SA?," where SA is the splitting subset for A, returned by Attribute selection method as part of the splitting criterion. It is a subset of the known values of A.

The algorithm uses the same process recursively to form a decision tree for the tuples at each resulting partition, Dj, of D (step 14).
The recursive partitioning stops only when any one of the following terminating conditions is true:

1. All the tuples in partition D (represented at node N) belong to the same class (steps 2 and 3).

2. There are no remaining attributes on which the tuples may be further partitioned (step 4). In this case, *majority voting* is employed (step 5). This involves converting node N into a leaf and labeling it with the most common class in D. Alternatively, the class distribution of the node tuples may be stored.

3. There are no tuples for a given branch, that is, a partition Dj is empty (step 12). In this case, a leaf is created with the majority class in D (step 13). The resulting decision tree is returned (step 15).

Attribute Selection Measures

An *attribute selection measure* is a heuristic for selecting the splitting criterion that "best" separates a given data partition, *D*, of class-labeled training tuples into individual classes. If we were to split *D* into smaller partitions according to the outcomes of the splitting criterion, ideally each partition would be pure (i.e., all the tuples that fall into a given partition would belong to the same class). Conceptually, the "best" splitting criterion is the one that most closely results in such a scenario. Attribute selection measures are also known as *splitting rules* because they determine how the tuples at a given node are to be split. The attribute selection measure provides a ranking for each attribute describing the given training tuples. The attribute having the best score for the measure 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 describes three popular attribute selection measures—*information gain, gain ratio*, and *Gini index*.
The notation used herein is as follows. Let *D*, the data partition, be a training set of class-

labeled tuples. Suppose the class label attribute has *m* distinct values defining *m* distinct classes, *Ci* (for *i*=1,…., *m*). Let $C_{i,D}$ be the set of tuples of class *Ci* in *D*. Let (*D*) and ($C_{i,D}$) denote the number of tuples in *D* and $C_{i,D}$, respectively.

Information Gain

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

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

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

where pi is the nonzero probability that an arbitrary tuple in D belongs to class Ci and is estimated by *(*Ci,D*)/(*D*)*. A log function to the base 2 is used, because the information is encoded in bits. Info(D) is just the average amount of information needed to identify the class label of a tuple in D. Note that, at this point, the information we have is based solely on the proportions of tuples of each class. Info(D) is also known as the *entropy* of D.

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

suppose we were to partition the tuples in D on some attribute A having v distinct values, *{*a1, a2,……, av*}*, as observed from the training data. If A is discrete-valued, these values correspond directly to the v outcomes of a test on A. Attribute A can be used to split D into v partitions or subsets, *{*D1, D2,……, Dv*}*.

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

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).

Example:

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

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

Gain Ratio

The information gain measure is biased toward tests with many outcomes. That is, it prefers to select attributes having a large number of values. For example, consider an attribute that acts as a unique identifier such as *product ID*. A split on *product ID* would result in a large number of partitions (as many as there are values), each one containing just one tuple. Because each partition is pure, the information required to classify data set *D* based on this partitioning would be $Info_{product\ ID}(D) = 0$. Therefore, the information gained by partitioning on this attribute is maximal.

C4.5, a successor of ID3, uses an extension to information gain known as gain ratio, which attempts to overcome this bias. It applies a kind of normalization to information gain using a "split information" value defined analogously with Info(D) as

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

The gain ratio is defined as

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

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

$$= 1.557.$$

$$GainRatio(income) = 0.029/1.557 = 0.019.$$

Gini Index

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

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

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

for a possible split-point of A, D1 is the set of tuples in D satisfying A ≤ split point, and D2 is the set of tuples in D satisfying A > split point

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

$$Gini(D) = 1 - \left(\frac{9}{14}\right)^2 - \left(\frac{5}{14}\right)^2 = 0.459.$$

$$Gini_{income \in \{low,medium\}}(D)$$

$$= \frac{10}{14}Gini(D_1) + \frac{4}{14}Gini(D_2)$$

$$= \frac{10}{14}\left(1 - \left(\frac{7}{10}\right)^2 - \left(\frac{3}{10}\right)^2\right) + \frac{4}{14}\left(1 - \left(\frac{2}{4}\right)^2 - \left(\frac{2}{4}\right)^2\right)$$

$$= 0.443$$

$$= Gini_{income \in \{high\}}(D).$$

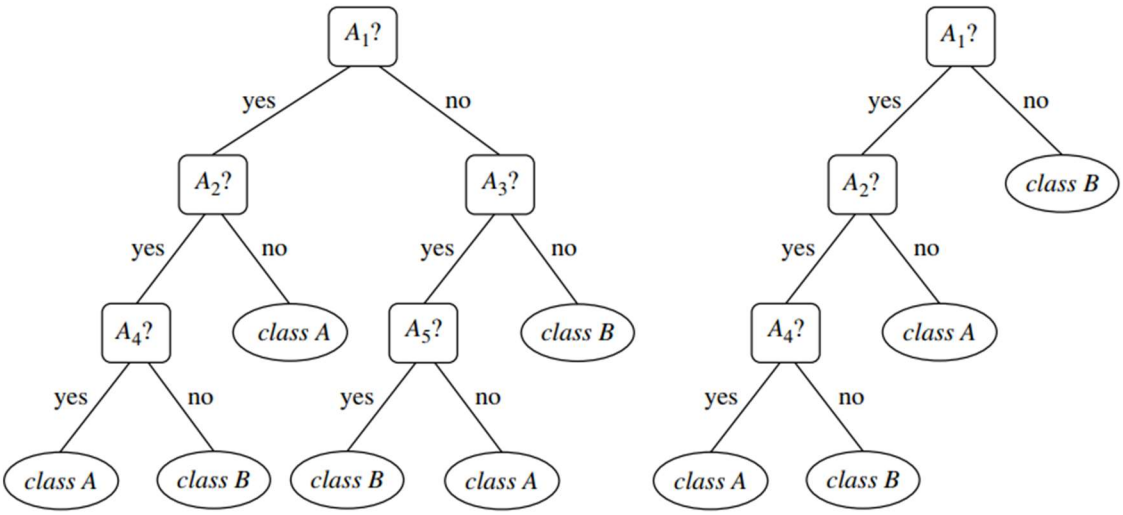
Tree Pruning

When a decision tree is built, many of the branches will reflect anomalies in the training data due to noise or outliers. Tree pruning methods address this problem of overfitting the data. Such methods typically use statistical measures to remove the least-reliable branches Pruned trees tend to be smaller and less complex and, thus, easier to comprehend. They are usually faster and better at correctly classifying independent test data (i.e., of previously unseen tuples) than unpruned trees.

In the prepruning approach, a tree is "pruned" by halting its construction early. Upon halting, the node becomes a leaf. The leaf may hold the most frequent class among the subset tuples or the probability distribution of those tuples.

When constructing a tree, measures such as statistical significance, information gain, Gini index, and so on, can be used to assess the goodness of a split. If partitioning the tuples at a node would result in a split that falls below a prespecified threshold, then further partitioning of the given subset is halted. There are difficulties, however, in choosing an appropriate threshold. High thresholds could result in oversimplified trees, whereas low thresholds could result in very little simplification.

The second and more common approach is postpruning, which removes subtrees from a "fully grown" tree. A subtree at a given node is pruned by removing its branches and replacing it with a leaf. The leaf is labeled with the most frequent class among the subtree being replaced



Bayes Classification Methods

Bayesian classifiers are statistical classifiers. They can predict class membership probabilities such as the probability that a given tuple belongs to a particular class. Naive Bayesian classifiers assume that the effect of an attribute value on a given class is independent of the values of the other attributes. This assumption is called class conditional independence. It is made to simplify the computations involved and, in this sense, is considered "naïve".

Bayes' Theorem

Bayes' theorem is named after Thomas Bayes. Let X be a data tuple. In Bayesian terms, X is considered "evidence." As usual, it is described by measurements made on a set of *n* attributes. Let *H* be some hypothesis such as that the data tuple X belongs to a specified class *C*. For classification problems, we want to determine *P(H*/X*)*, the probability that the hypothesis *H* holds given the "evidence" or observed data tuple X.

P*(*H/*X)* is the *posterior probability*, or a posteriori probability, of H conditioned on *X*. For example, suppose our world of data tuples is confined to customers described by the attributes age and income, respectively, and that *X* is a 35-year-old customer with an income of $40,000. Suppose that H is the hypothesis that our customer will buy a computer. Then P*(*H/*X)* reflects the probability that customer *X* will buy a computer given that we know the customer's age and income.

P*(*H*)* *is the prior probability, or* a priori probability, *of* H. *For our example, this is the probability that any given customer will buy a computer, regardless of age, income.*

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

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

Bayes' theorem is

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

Naive Bayesian Classification:

The naive Bayesian classifier, or simple Bayesian classifier, works as follows: *1*. Let *D* be a training set of tuples and their associated class labels. As usual, each tuple is represented by an *n*-dimensional attribute vector, X= *(x*1, *x*2,……, *xn)*, depicting *n* measurements made on the tuple from *n* attributes, respectively, *A*1, *A*2,……, *An*. *2*. Suppose that there are *m* classes, *C*1, *C*2,……, *Cm*. Given a tuple, X, the classifier will predict that X belongs to the class having the highest posterior probability, conditioned on X. That is, the naive Bayesian classifier predicts that tuple X belongs to the class *Ci* if and only if

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

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

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

As P*(X)* is constant for all classes, only P*(X*|Ci*)*P*(*Ci*)* needs to be maximized. If the class prior probabilities are not known, then it is commonly assumed that the classes are equally likely, that is, P*(*C1*)*= P*(*C2*)* = ⋯ =P*(*Cm*)*, and we would therefore maximize P*(X*/Ci*)*. Otherwise, we maximize P*(X*/Ci*)*P*(*Ci*)*. Note that the class prior

probabilities may be estimated by PCi*)*= (Ci,D)/(D), where (Ci,D) is the number of training tuples of class Ci in D.

4. Given data sets with many attributes, it would be extremely computationally expensive to compute P*(X*/Ci*)*. To reduce computation in evaluating P*(X*/Ci*)*, the naive assumption of *class-conditional independence* is made. This presumes that the attributes' values are conditionally independent of one another, given the class label of the tuple. Thus,

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

We can easily estimate the probabilities P*(*x1/Ci*)*, P*(*x2/Ci*)*,……, P*(*xn/Ci*)* from the training tuples. Recall that here xk refers to the value of attribute Ak for tuple X. For each attribute, we look at whether the attribute is categorical or continuous-valued. For instance, to compute P*(*X/Ci*)*, we consider the following:
(a) If Ak is categorical, then P*(*xk/Ci*)* is the number of tuples of class Ci in D having the value xk for Ak, divided by *(*Ci,D*)*, the number of tuples of class Ci in D.
(b) If Ak is continuous-valued, then we need to do a bit more work, but the calculation is pretty straightforward. A continuous-valued attribute is typically assumed to have a Gaussian distribution with a mean μ and standard deviation σ, defined by

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

For example, let X = *(*35,$40,000*)*, where *A*1 and *A*2 are the attributes *age* and *income*, respectively. Let the class label attribute be *buys_computer*. The associated class label for X is *yes* (i.e., *buys_computer* = *yes*). Let's suppose that *age* has not been discretized and therefore exists as a continuous-valued attribute. Suppose that from the training set, we find that customers in *D* who buy a computer are 38 ± 12 years of age. In other words, for attribute *age* and this class, we have μ = 38 years and σ = 12.

To predict the class label of X, *P(*X*/Ci)P(Ci)* is evaluated for each class *Ci*. The classifier predicts that the class label of tuple X is the class *Ci* if and only if

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

Predicting a class label using naive Bayesian classification. We wish to predict the class label of a tuple using naive Bayesian classification, given the same training data for decision tree induction. The training data were shown earlier in Table above. 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 $C1$ correspond to the class *buys_computer = yes* and $C2$ correspond to *buys_computer = no.* The tuple we wish to classify is X = (*age = youth, income = medium, student = yes, credit rating = fair*)

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

We need to maximize $P(\boldsymbol{X}|C_i)P(C_i)$, for $i = 1, 2$. $P(C_i)$, the prior probability of each class, can be computed based on the training tuples:

$$P(buys\_computer = yes) = 9/14 = 0.643$$
$$P(buys\_computer = no) = 5/14 = 0.357$$

To compute $P(\boldsymbol{X}|C_i)$, for $i = 1, 2$, we compute the following conditional probabilities:

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

Using these probabilities, we obtain

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

Similarly,

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

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

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

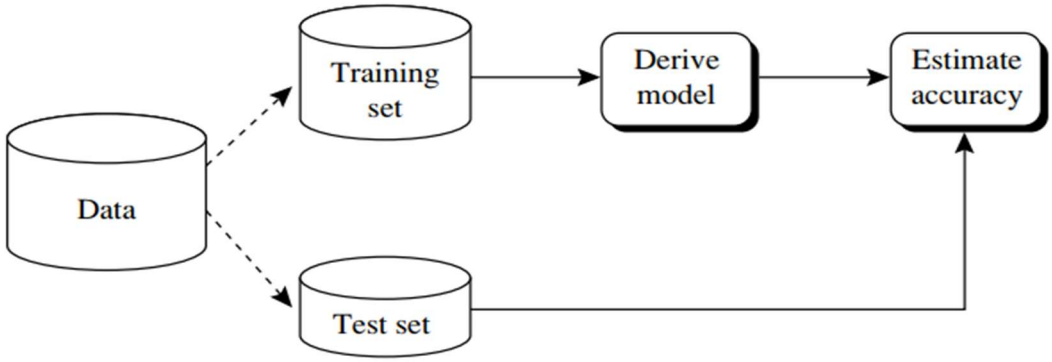
Therefore, the naïve Bayesian classifier predicts *buys_computer = yes* for tuple $\boldsymbol{X}$.

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

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

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

False positives (*FP*)*: These are the negative tuples that were incorrectly labeled as positive (e.g., tuples of class buys_computer = no for which the classifier predicted buys_computer =*



Cross-Validation

In k-fold cross-validation, the initial data are randomly partitioned into k *mutually exclusive subsets or "folds,"* D*1,* D*2,……,* Dk*, each of approximately equal size. Training and testing is performed* k *times. In iteration* i*, partition* Di *is reserved as the test set, and the remaining partitions are collectively used to train the model. That is, in the first iteration, subsets* D*2,……,* Dk *collectively serve as the training set to obtain a first model, which is tested on* D*1; the second iteration is trained on subsets* D*1,* D*3,……,* Dk *and tested on* D*2; and so on. Unlike*

the holdout and random subsampling methods, here each sample is used the same number of times for training and once for testing. For classification, the accuracy estimate is the overall number of correct classifications from the k *iterations, divided by the total number of tuples in the initial data.*

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

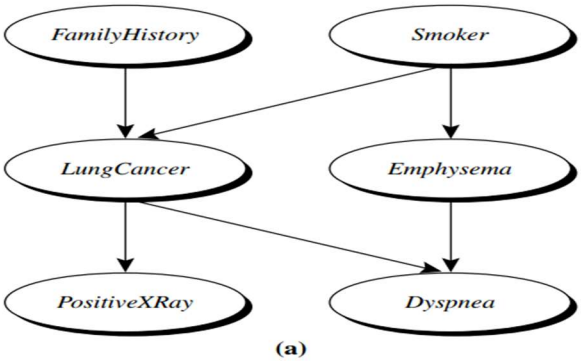
In general, stratified 10-fold cross-validation is recommended for estimating accuracy (even if computation power allows using more folds) due to its relatively low bias and variance

Bayesian Belief Networks

Bayesian belief networks—probabilistic graphical models, which unlike naïve Bayesian classifiers allow the representation of dependencies among subsets of attributes. Bayesian belief networks can be used for classification.

The naıve Bayesian classifier makes the assumption of class conditional independence, 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 probability distributions. They allow class conditional independencies to be defined between subsets of variables. They provide a graphical 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.

A belief network is defined by two components—a directed acyclic graph and a set of conditional probability tables. 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*



(a)

	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

(b)

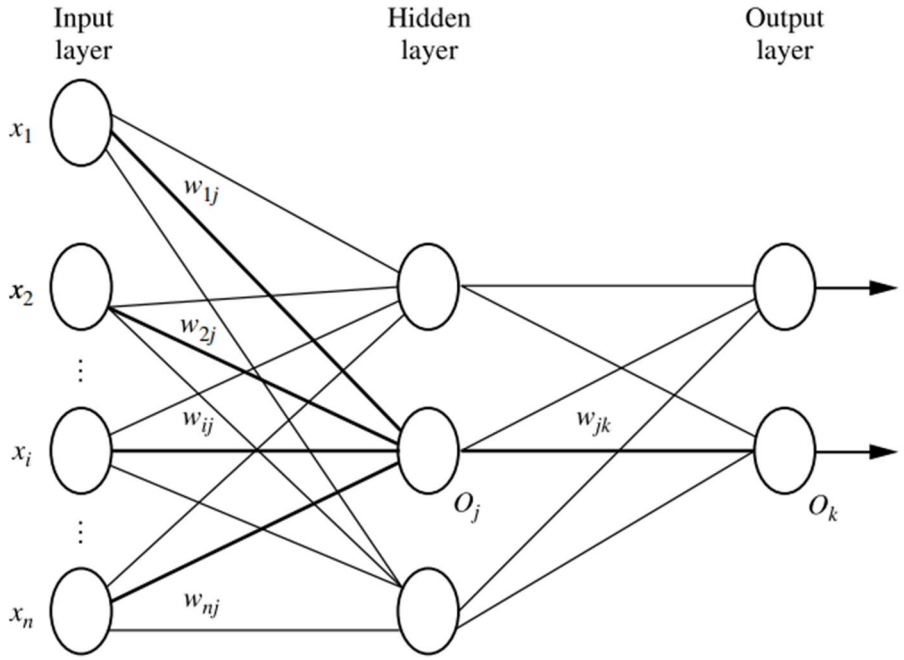
Simple Bayesian belief network. (a) A proposed causal model, represented by a directed acyclic graph. (b) The conditional probability table for the values of the variable LungCancer (LC) showing each possible combination of the values of its parent nodes, FamilyHistory (FH) and Smoker (S).

A belief network has one conditional probability table (CPT) for each variable. The CPT for a variable *Y* specifies the conditional distribution *P(Y/Parents(Y))*, where *Parents(Y)* are the parents of *Y*. The conditional probability for each known value of *LungCancer* is given for each possible combination of the values of its parents. For instance, from the upper leftmost and bottom rightmost entries, respectively, we see that *P(LungCancer = yes/FamilyHistory = yes, Smoker = yes)* = 0.8 *P(LungCancer = no/FamilyHistory = no, Smoker = no)* = 0.9. Let *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 nondescendants 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:

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

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.



Each layer is made up of units. The inputs to the network correspond to the attributes measured for each training tuple. The inputs are fed simultaneously into the units making up the input layer.

These inputs pass through the input layer and are then weighted and fed simultaneously to a second layer of "neuronlike" units, known as a hidden layer. The outputs of the hidden layer units can be input to another hidden layer, and so on. The number of hidden layers is arbitrary, although in practice, usually only one is used. The weighted outputs of the last hidden layer are input to units making up the output layer, which emits the network's prediction for given tuples. The units in the input layer are called input units.

The units in the hidden layers and output layer are sometimes referred to as neurodes, due to their symbolic biological basis, or as output units. Therefore, we say that it is a two-layer neural network. (The input layer is not counted because it serves only to pass the input values to the next layer.) Similarly, a network containing two hidden layers is called a *three-layer* neural network, and so on. It is a feed-forward network since none of the weights cycles back to an input unit or to a previous layer's output unit. It is fully connected in that each unit provides input to each unit in the next forward layer

Defining a Network Topology

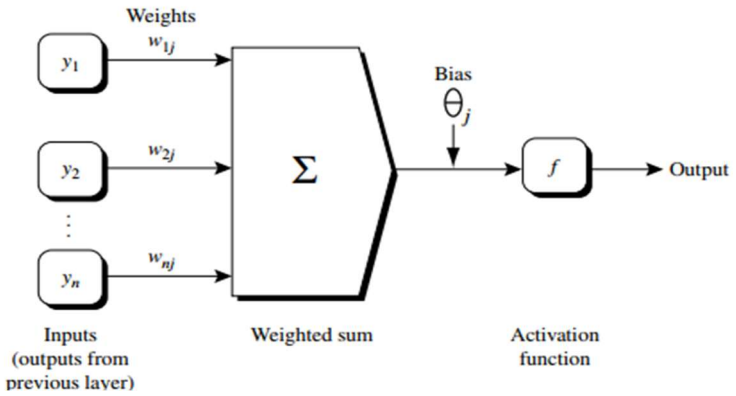
"*How can I design the neural network's topology?*" Before training can begin, the user must decide on the network topology by specifying the number of units in the input layer, the number of hidden layers (if more than one), the number of units in each hidden layer, and the number of units in the output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

Normalizing the input values for each attribute measured in the training tuples will help speed up the learning phase. Typically, input values are normalized so as to fall between 0.0 and 1.0. Discrete-valued attributes may be encoded such that there is one input unit per domain value. For example, if an attribute A has three possible or known values, namely $\{a0, a1, a2\}$, then we may assign three input units to represent A. That is, we may have, say, $I0$, $I1$, $I2$ as input units. Each unit is initialized to 0. If $A = a0$, then $I0$ is set to 1 and the rest are 0. If $A = a1$, then $I1$ is set to 1 and the rest are 0, and so on.

Neural networks can be used for both classification (to predict the class label of a given tuple) and numeric prediction (to predict a continuous-valued output). For 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. There are no clear rules as to the "best" number of hidden layer units. Network design is a trial-and-error process and may affect the accuracy of the resulting trained network. The initial values of the weights may also affect the resulting accuracy.

Once a network has been trained and its accuracy is not considered acceptable, it is common to repeat the training process with a different network topology or a different set of initial weights.

Cross-validation techniques for accuracy estimation can be used to help decide when an acceptable network has been found. A number of automated techniques have been proposed that search for a "good" network structure. These typically use a hill-climbing approach that starts with an initial structure that is selectively modified.



Algorithm: Backpropagation. Neural network learning for classification or numeric 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 $\boldsymbol{X}$ 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) } }

Backpropagation algorithm.

Hidden or output layer unit j: The inputs to unit j are outputs from the previous layer. These are multiplied by their corresponding weights to form a weighted sum, which is added to the bias associated with unit j. A nonlinear activation function is applied to the net input. (For ease of explanation, the inputs to unit j are labeled y1, y2,……, yn. If unit j were in the first hidden layer, then these inputs would correspond to the input tuple *(*x1, x2,……, xn*)*.)

$$I_j = \sum_i w_{ij} O_i + \theta_j,$$

where wij is the weight of the connection from unit i in the previous layer to unit j; Oi is the output of unit i from the previous layer; and θj is the *bias* of the unit. The bias acts as a threshold in that it serves to vary the activity of the unit.

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

$$O_j = \frac{1}{1 + e^{-I_j}}.$$

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

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

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

$$Err_j = O_j(1 - O_j)(T_j - O_j)$$

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

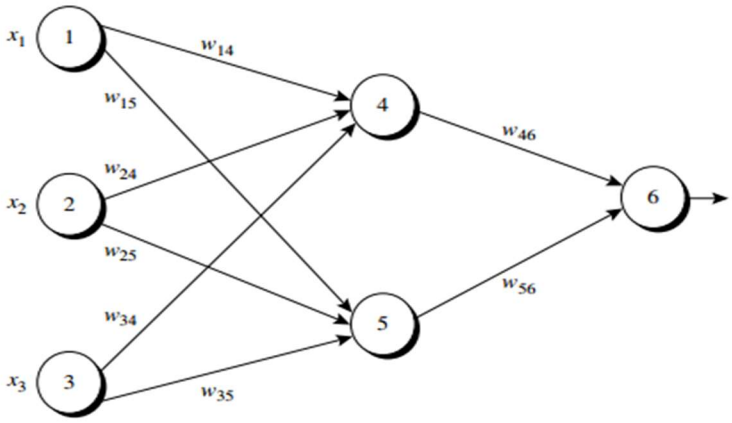
$$Err_j = O_j(1 - O_j) \sum_k Err_k w_{jk},$$

The weights and biases are updated to reflect the propagated errors. Weights are updated by the following equations, where Δw_{ij} is the change in weight *wij*

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

Biases are updated by the following equations, where $\Delta\theta_j$ is the change in bias θj:

$$\Delta\theta_j = (l) Err_j.$$
$$\theta_j = \theta_j + \Delta\theta_j.$$



5 Example of a multilayer feed-forward neural network.

1 Initial Input, Weight, and Bias Values

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

2 Net Input and Output Calculations

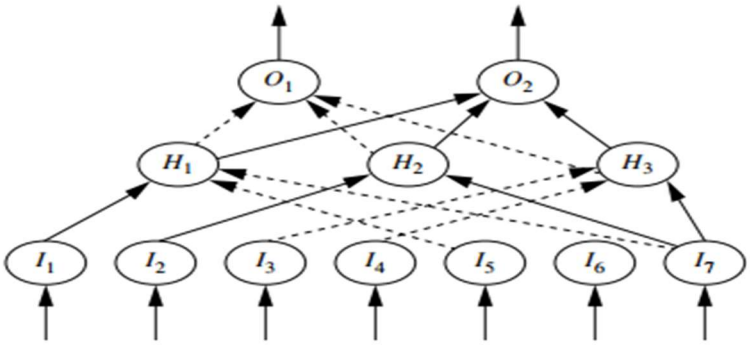
Unit, j	*Net Input, I_j*	*Output, O_j*
4	$0.2+0-0.5-0.4=-0.7$	$1/(1+e^{0.7})=0.332$
5	$-0.3+0+0.2+0.2=0.1$	$1/(1+e^{-0.1})=0.525$
6	$(-0.3)(0.332)-(0.2)(0.525)+0.1=-0.105$	$1/(1+e^{0.105})=0.474$

3 Calculation of the Error at Each Node

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

Calculations for Weight and Bias Updating

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



Identify sets of common activation values for each hidden node, H_i: for H_1: (−1,0,1) for H_2: (0,1) for H_3: (−1,0.24,1)
Derive rules relating common activation values with output nodes, O_j: IF (H_2=0 AND H_3=−1) OR (H_1=−1 AND H_2=1 AND H_3=−1) OR (H_1=−1 AND H_2=0 AND H_3=0.24) THEN O_1=1, O_2=0 ELSE O_1=0, O_2=1

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

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

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, X1 = *(*x11, x12,……, x1n*)* and X2 = *(*x21, x22,……, x2n*)*, is

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

Clustering is the process of grouping a set of data objects into multiple groups or clusters so that objects within a cluster have high similarity, but are very dissimilar to objects in other clusters. Dissimilarities and similarities are assessed based on the attribute values describing the objects and often involve distance measures.

Clustering as a data mining tool has its roots in many application areas such as biology, security, business intelligence, and Web search.

Cluster analysis or simply clustering is the process of partitioning a set of data objects (or observations) into subsets. Each subset is a cluster, such that objects in a cluster are similar to one another, yet dissimilar to objects in other clusters. The set of clusters resulting from a cluster analysis can be referred to as a clustering. In this context, different clustering methods may generate different clusterings on the same data set. The partitioning is not performed by humans, but by the clustering algorithm. Hence, clustering is useful in that it can lead to the discovery of previously unknown groups within the data.

Cluster analysis has been widely used in many applications such as business intelligence, image pattern recognition, Web search, biology, and security. In business intelligence, clustering can be used to organize a large number of customers into groups, where customers within a group share strong similar characteristics. This facilitates the development of business strategies for enhanced customer relationship management.

Moreover, consider a consultant company with a large number of projects. To improve project management, clustering can be applied to partition projects into categories based on similarity so that project auditing and diagnosis can be conducted effectively.

In image recognition, clustering can be used to discover clusters or "subclasses" in handwritten character recognition systems. Suppose we have a data set of handwritten digits, where each digit is labeled as either 1, 2, 3, and so on. Note that there can be a large variance in the way in which people write the same digit. Some people may write it with a small circle at the left bottom part, while some others may not. We can use clustering to determine subclasses for "2," each of which represents a variation on the way in which 2 can be written. Using multiple models based on the subclasses can improve overall recognition accuracy.

Clustering has also found many applications in Web search. For example, a keyword search may often return a very large number of hits due to the extremely large number of web pages. Clustering can be used to organize the search results into groups and present the results in a concise and easily accessible way. Moreover, clustering techniques have been developed to cluster documents into topics, which are commonly used in information retrieval practice.

a cluster is a collection of data objects that are similar to one another within the cluster and dissimilar to objects in other clusters, a cluster of data objects can be treated as an implicit class. In this sense, clustering is sometimes called automatic classification

classification is known as supervised learning because the class label information is given, that is, the learning algorithm is supervised in that it is told the class membership of each training tuple. Clustering is known as unsupervised learning because the class label information is not present. For this reason, clustering is a form of learning by observation, rather than *learning by examples*.

Requirements for Cluster Analysis

Clustering is a challenging research field, the requirements for clustering as a data mining tool, as well as aspects that can be used for comparing clustering methods.

The following are typical requirements of clustering in data mining.

Scalability: Many clustering algorithms work well on small data sets containing fewer than several hundred data objects; however, a large database may contain millions or even billions of objects, particularly in Web search scenarios. Clustering on only a sample of a given large data set may lead to biased results. Therefore, highly scalable clustering algorithms are needed.

Ability to deal with different types of attributes: Many algorithms are designed to cluster numeric (interval-based) data. However, applications may require clustering other data types, such as binary, nominal (categorical), and ordinal data, or mixtures of these data types. Recently, more and more applications need clustering techniques for complex data types such as graphs, sequences, images, and documents.

Discovery of clusters with arbitrary shape: Many clustering algorithms determine clusters based on Euclidean or Manhattan distance measures. Algorithms based on such distance measures tend to find spherical clusters with similar size and density. However, a cluster could be of any shape. Consider sensors, for example, which are often deployed for environment surveillance. Cluster analysis on sensor readings can detect interesting phenomena. We may want to use clustering to find the frontier of a running forest fire, which is often not spherical. It is important to develop algorithms that can detect clusters of arbitrary shape.

Requirements for domain knowledge to determine input parameters: Many clustering algorithms require users to provide domain knowledge in the form of input parameters such as the desired number of clusters. Consequently, the clustering results may be sensitive to such parameters. Parameters are often hard to determine, especially for high-dimensionality data sets and where users have yet to grasp a deep understanding of their data. Requiring the specification of domain knowledge not only burdens users, but also makes the quality of clustering difficult to control.

Ability to deal with noisy data: Most real-world data sets contain outliers and/or missing, unknown, or erroneous data. Sensor readings, for example, are often noisy—some readings may be inaccurate due to the sensing mechanisms, and some readings may be erroneous due to interferences from surrounding transient objects. Clustering algorithms can be sensitive to such noise and may produce poor-quality clusters. Therefore, we need clustering methods that are robust to noise.

Incremental clustering and insensitivity to input order: In many applications, incremental updates (representing newer data) may arrive at any time. Some clustering algorithms cannot incorporate incremental updates into existing clustering structures and, instead, have to recompute a new clustering from scratch. Clustering algorithms may also be sensitive to the input data order. That is, given a set of data objects, clustering algorithms may return dramatically different clusterings depending on the order in which the objects are presented.

Incremental clustering algorithms and algorithms that are insensitive to the input order are needed.

Capability of clustering high-dimensionality data: A data set can contain numerous dimensions or attributes. When clustering documents, for example, each keyword can be regarded as a dimension, and there are often thousands of keywords. Most clustering algorithms are good at handling low-dimensional data such as data sets involving only two or three dimensions. Finding clusters of data objects in a highdimensional space is challenging, especially considering that such data can be very sparse and highly skewed.

Constraint-based clustering: Real-world applications may need to perform clustering under various kinds of constraints. Suppose that your job is to choose the locations for a given number of new automatic teller machines (ATMs) in a city. To decide upon this, you may cluster households while considering constraints such as the city's rivers and highway networks and the types and number of customers per cluster. A challenging task is to find data groups with good clustering behavior that satisfy specified constraints.

Interpretability and usability: Users want clustering results to be interpretable, comprehensible, and usable. That is, clustering may need to be tied in with specific semantic interpretations and applications. It is important to study how an application goal may influence the selection of clustering features and clustering methods.

The following are orthogonal aspects with which clustering methods can be compared:

The partitioning criteria: In some methods, all the objects are partitioned so that no hierarchy exists among the clusters. That is, all the clusters are at the same level conceptually. Such a method is useful, for example, for partitioning customers into groups so that each group has its own manager. Alternatively, other methods partition data objects hierarchically, where clusters can be formed at different semantic levels. For example, in text mining, we may want to organize a corpus of documents into multiple general topics, such as "politics" and "sports," each of which may have subtopics, For instance, "football," "basketball," "baseball," and "hockey" can exist as subtopics of "sports." The latter four subtopics are at a lower level in the hierarchy than "sports."

Separation of clusters: Some methods partition data objects into mutually exclusive clusters. When clustering customers into groups so that each group is taken care of by one manager, each customer may belong to only one group. In some other situations, the clusters may not be exclusive, that is, a data object may belong to more than one cluster. For example, when clustering documents into topics, a document may be related to multiple topics. Thus, the topics as clusters may not be exclusive.

Similarity measure: Some methods determine the similarity between two objects by the distance between them. Such a distance can be defined on Euclidean space, a road network, a vector space, or any other space. In other methods, the similarity may be defined by connectivity based on density or contiguity, and may not rely on the absolute distance between two objects. Similarity measures play a fundamental role in the design of clustering methods. While distance-based methods can often take advantage of optimization techniques, density- and continuity-based methods can often find clusters of arbitrary shape.

Clustering space: Many clustering methods search for clusters within the entire given data space. These methods are useful for low-dimensionality data sets. With highdimensional data, however, there can be many irrelevant attributes, which can make similarity measurements unreliable. Consequently, clusters found in the full space are often meaningless. It's often better to instead search for clusters within different subspaces of the same data set. *Subspace clustering* discovers clusters and subspaces (often of low dimensionality) that manifest object similarity.

Partitioning methods: Given a set of n objects, a partitioning method constructs k partitions of the data, where each partition represents a cluster and $k \leq n$. That is, it divides the data into k groups such that each group must contain at least one object. In other words, partitioning methods conduct one-level partitioning on data sets.

The basic partitioning methods typically adopt *exclusive cluster separation*. That is, each object must belong to exactly one group. This requirement may be relaxed, for example, in fuzzy partitioning techniques.

Most partitioning methods are distance-based. Given k, the number of partitions to construct, a partitioning method creates an initial partitioning. It then uses an iterative relocation technique that attempts to improve the partitioning by moving objects from one group to another. The general criterion of a good partitioning isthat objects in the same cluster are "close" or related to each other, whereas objects in different clusters are "far apart" or very different. There are various kinds of other criteria for judging the quality of partitions. Traditional partitioning methods can be extended for subspace clustering, rather than searching the full data space. This is useful when there are many attributes and the data are sparse.

Achieving global optimality in partitioning-based clustering is often computationally prohibitive, potentially requiring an exhaustive enumeration of all the possible partitions. Instead, most applications adopt popular heuristic methods, such as greedy approaches like the *k-means* and the *k-medoids* algorithms, which progressively improve the clustering quality and approach a local optimum. These heuristic clustering methods work well for finding spherical-shaped clusters in small- to medium-size databases. To find clusters with complex shapes and for very large data sets, partitioning-based methods need to be extended.

Hierarchical methods: A hierarchical method creates a hierarchical decomposition of the given set of data objects. A hierarchical method can be classified as being either *agglomerative* or *divisive*, based on how the hierarchical decomposition is formed. The *agglomerative approach*,

also called the *bottom-up* approach, starts with each object forming a separate group. It successively merges the objects or groups close to one another, until all the groups are merged into one (the topmost level of the hierarchy), or a termination condition holds. The *divisive approach*, also called the *top-down* approach, starts with all the objects in the same cluster. In each successive iteration, a cluster is split into smaller clusters, until eventually each object is in one cluster, or a termination condition holds. Hierarchical clustering methods can be distance-based or density- and continuity based. Various extensions of hierarchical methods consider clustering in subspaces as well.

Hierarchical methods suffer from the fact that once a step (merge or split) is done, it can never be undone. This rigidity is useful in that it leads to smaller computation costs by not having to worry about a combinatorial number of different choices. Such techniques cannot correct erroneous decisions; however, methods for improving the quality of hierarchical clustering have been proposed.

Density-based methods: Most partitioning methods cluster objects based on the distance between objects. Such methods can find only spherical-shaped clusters and encounter difficulty in discovering clusters of arbitrary shapes. Other clustering methods have been developed based on the notion of *density*. Their general idea is to continue growing a given cluster as long as the density (number of objects or data points) in the "neighborhood" exceeds some threshold. For example, for each data point within a given cluster, the neighborhood of a given radius has to contain at least a minimum number of points. Such a method can be used to filter out noise or outliers and discover clusters of arbitrary shape. Density-based methods can divide a set of objects into multiple exclusive clusters, or a hierarchy of clusters. Typically, density-based methods consider exclusive clusters only, and do not consider fuzzy clusters. Moreover, density-based methods can be extended from full space to subspace clustering.

Grid-based methods: Grid-based methods quantize the object space into a finite number of cells that form a grid structure. All the clustering operations are performed on the grid structure (i.e., on the quantized space). The main advantage of this approach is its fast processing time, which is typically independent of the number of data objects and dependent only on the number of cells in each dimension in the quantized space.

Method	General Characteristics
Partitioning methods	– Find mutually exclusive clusters of spherical shape – Distance-based – May use mean or medoid (etc.) to represent cluster center – Effective for small- to medium-size data sets
Hierarchical methods	– Clustering is a hierarchical decomposition (i.e., multiple levels) – Cannot correct erroneous merges or splits – May incorporate other techniques like microclustering or consider object "linkages"
Density-based methods	– Can find arbitrarily shaped clusters – Clusters are dense regions of objects in space that are separated by low-density regions – Cluster density: Each point must have a minimum number of points within its "neighborhood" – May filter out outliers
Grid-based methods	– Use a multiresolution grid data structure – Fast processing time (typically independent of the number of data objects, yet dependent on grid size)

Partitioning Methods

The simplest and most fundamental version of cluster analysis is partitioning, which organizes the objects of a set into several exclusive groups or clusters. To keep the problem specification concise, we can assume that the number of clusters is given as background knowledge. This parameter is the starting point for partitioning methods. Formally, given a data set, *D*, of *n* objects, and *k*, the number of clusters to form, a *partitioning algorithm* organizes the objects into *k* partitions ($k \leq n$), where each partition represents a cluster. The clusters are formed to optimize an objective partitioning criterion, such as a dissimilarity function based on distance, so that the objects within a cluster are "similar" to one another and "dissimilar" to objects in other clusters in terms of the data set attributes.

k-*Means: A Centroid-Based Technique*

Suppose a data set, D, contains n objects in Euclidean space. Partitioning methods distribute the objects in D into k clusters, C1,……,Ck, that is, $Ci \subset D$ *and* $Ci \cap Cj = \emptyset$ *; for* ($1 \leq i,j \leq k$). *An objective function is used to assess the partitioning quality so that objects within a cluster are similar to one another but dissimilar to objects in other clusters. This is, the objective function aims for high intra cluster similarity and low inter cluster similarity.*

A centroid-based partitioning technique uses the centroid of a cluster, Ci, to represent that cluster. Conceptually, the centroid of a cluster is its center point. The centroid can be defined in various ways such as by the mean or medoid of the objects (or points) assigned to the cluster. The difference between an object p 2 Ci and ci, the representative of the cluster, is measured by dist(p,ci), where dist(x,y) is the Euclidean distance between two points x and y. The quality of cluster Ci can be measured by the within cluster variation*, which is the sum of squared error between all objects in Ci and the centroid ci, defined as*

$$E = \sum_{i=1}^{k} \sum_{p \in C_i} dist(\boldsymbol{p}, \boldsymbol{c_i})^2,$$

The k-means algorithm defines the centroid of a cluster as the mean value of the points within the cluster. It proceeds as follows. First, it randomly selects k of the objects in D, each of which initially represents a cluster mean or center. For each of the remaining objects, an object is assigned to the cluster to which it is the most similar, based on the Euclidean distance between the object and the cluster mean. The k-means algorithm then iteratively improves the within-cluster variation. For each cluster, it computes the new mean using the objects assigned to the cluster in the previous iteration. All the objects are then reassigned using the updated means as the new cluster centers. The iterations continue until the assignment is stable, that is, the clusters formed in the current round are the same as those formed in the previous round.

Algorithm: *k*-means. The *k*-means algorithm for partitioning, where each cluster's center is represented by the mean value of the objects in the cluster.

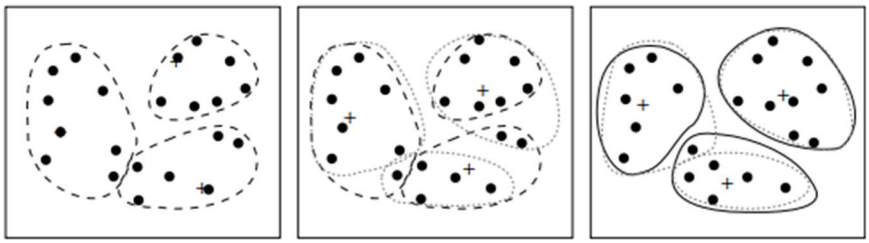
Input:

- *k*: the number of clusters,
- *D*: a data set containing *n* objects.

Output: A set of *k* clusters.

Method:

(1) arbitrarily choose *k* objects from *D* as the initial cluster centers;
(2) **repeat**
(3) (re)assign each object to the cluster to which the object is the most similar, based on the mean value of the objects in the cluster;
(4) update the cluster means, that is, calculate the mean value of the objects for each cluster;
(5) **until** no change;



(a) Initial clustering **(b)** Iterate **(c)** Final clustering

k-*Medoids: A Representative Object-Based Technique*

The k-means algorithm is sensitive to outliers because such objects are far away from the majority of the data, and thus, when assigned to a cluster, they can dramatically distort the mean value of the cluster. This inadvertently affects the assignment of other objects to clusters

The Partitioning Around Medoids (PAM) algorithm is a popular realization of *k*-medoids clustering. It tackles the problem in an iterative, greedy way. Like the *k*-means algorithm, the initial representative objects (called seeds) are chosen arbitrarily. We consider whether replacing a representative object by a nonrepresentative object would improve the clustering quality. All the possible replacements are tried out. The iterative process of replacing representative objects by other objects continues until the quality of the resulting clustering cannot be improved by any replacement. This quality is measured by a cost function of the average dissimilarity between an object and the representative object of its cluster. Specifically, let *o*1,…….,*ok* be the current set of representative objects (i.e., medoids). To determine whether a nonrepresentative object, denoted by *orandom*, is a good replacement for a current medoid *oj* ($1 \leq j \leq k$), we calculate the distance from every object *p* to the closest

object in the set *{o*1,…….,*oj*-1,*orandom*,*oj*+1,…….,*ok}*, and use the distance to update the cost function. The reassignments of objects to *{o*1,……,*oj*-1,*orandom*,*oj*+1,……,*ok}* are simple. Suppose object *p* is currently assigned to a cluster represented by medoid *oj*.

The k-medoids method is more robust than k-means in the presence of noise and outliers because a medoid is less influenced by outliers or other extreme values than a mean. However, the complexity of each iteration in the k-medoids algorithm is $O(k(n-k)^2)$. For large values of n and k, such computation becomes very costly, and much more costly than the k-means method. Both methods require the user to specify k, the number of clusters. "How can we scale up the k-medoids method?" A typical k-medoids partitioning algorithm like PAM works effectively for small data sets, but does not scale well for large data sets. To deal with larger data sets, a sampling-based method called CLARA (Clustering LARge Applications) can be used. Instead of taking the whole data set into consideration, CLARA uses a random sample of the data set. The PAM algorithm is then applied to compute the best medoids from the sample.

Hierarchical Methods

While partitioning methods meet the basic clustering requirement of organizing a set of objects into a number of exclusive groups, in some situations we may want to partition our data into groups at different levels such as in a hierarchy. A *hierarchical clustering method* works by grouping data objects into a hierarchy or "tree" of clusters.

agglomerative versus divisive hierarchical clustering, which organize objects into a hierarchy using a bottom-up or top-down strategy, respectively. Agglomerative methods start with individual objects as clusters, which are iteratively merged to form larger clusters. Conversely, divisive methods initially let all the given objects form one cluster, which they iteratively split into smaller clusters.

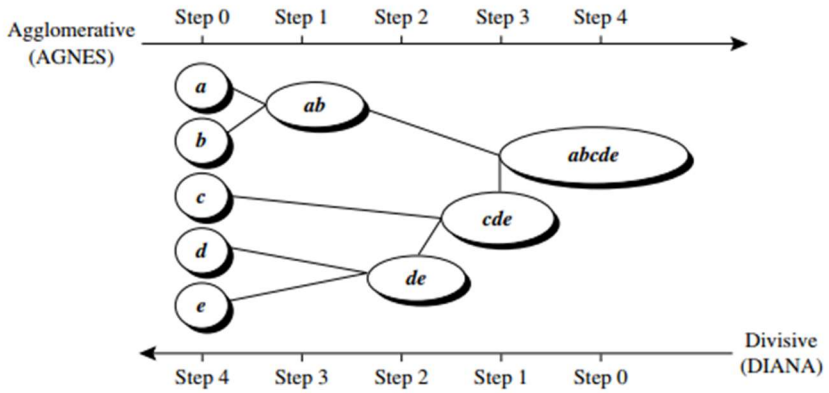
Hierarchical clustering methods can encounter difficulties regarding the selection of merge or split points. Such a decision is critical, because once a group of objects is merged or split, the process at the next step will operate on the newly generated clusters. It will neither undo what was done previously, nor perform object swapping between clusters. Thus, merge or split decisions, if not well chosen, may lead to low-quality clusters. Moreover, the methods do not scale well because each decision of merge or split needs to examine and evaluate many objects or clusters.

A hierarchical clustering method can be either agglomerative or divisive, depending on whether the hierarchical decomposition is formed in a bottom-up (merging) or topdown (splitting) fashion.

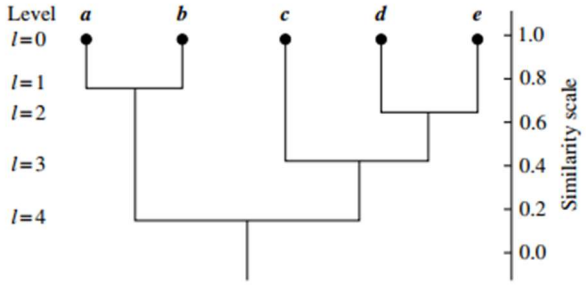
An agglomerative hierarchical clustering method uses a bottom-up strategy. It typically starts by letting each object form its own cluster and iteratively merges clusters into larger and larger clusters, until all the objects are in a single cluster or certain termination conditions are satisfied. The single cluster becomes the hierarchy's root. For the merging step, it finds the two clusters that are closest to each other (according to some similarity measure), and combines the two to form one cluster. Because two clusters are merged per iteration, where each cluster contains at least one object, an agglomerative method requires at most *n* iterations.

A divisive hierarchical clustering method employs a top-down strategy. It starts by placing all objects in one cluster, which is the hierarchy's root. It then divides the root cluster into several smaller subclusters, and recursively partitions those clusters into smaller ones. The partitioning process continues until each cluster at the lowest level is coherent enough—either containing only one object, or the objects within a cluster are sufficiently similar to each other.

A tree structure called a dendrogram is commonly used to represent the process of hierarchical clustering. It shows how objects are grouped together (in an agglomerative method) or partitioned (in a divisive method) step-by-step.



Agglomerative and divisive hierarchical clustering on data objects $\{a, b, c, d, e\}$.



Dendrogram representation for hierarchical clustering of data objects $\{a, b, c, d, e\}$.