

# MODULE 2



**Table 3.1.** A loan application data set

ID	Age	Has_job	Own_house	Credit_rating	Class
1	young	false	false	fair	No
2	young	false	false	good	No
3	young	true	false	good	Yes
4	young	true	true	fair	Yes
5	young	false	false	fair	No
6	middle	false	false	fair	No
7	middle	false	false	good	No
8	middle	true	true	good	Yes
9	middle	false	true	excellent	Yes
10	middle	false	true	excellent	Yes
11	old	false	true	excellent	Yes
12	old	false	true	good	Yes
13	old	true	false	good	Yes
14	old	true	false	excellent	Yes
15	old	false	false	fair	No



- Table 3.1 shows a small loan application data set. It has four attributes.
- The **first attribute** is **Age**, which has three possible values, young, middle and old.
- The **second attribute** is **Has\_Job**, which indicates whether an applicant has a job. Its possible values are true (has a job) and false (does not have a job).
- The **third attribute** is **Own\_house**, which shows whether an applicant owns a house.



- The **fourth attribute** is **Credit\_rating**, which has three possible values, fair, good and excellent.
- The last column is the **Class attribute**, which shows whether each loan application was approved (denoted by Yes) or not (denoted by No) in the past.



- We want to learn a classification model from this data set that can be used to classify **future loan applications**.
- That is, when a new customer comes into the bank to apply for a loan, after inputting his/her age, whether he/she has a job, whether he/she owns a house, and his/her credit rating, the classification model should predict whether his/her loan application should be approved.



- Our learning task is called **supervised learning** because the class labels (e.g., Yes and No values of the class attribute in Table 3.1) are provided in the data

# Basic Concepts



- A data set used in the learning task consists of a set of data records, which are described by a set of attributes  $A = \{A_1, A_2, \dots, A_{|A|}\}$ , where  $|A|$  denotes the number of attributes or the size of the set  $A$ .

# Basic Concepts



- The data set also has a special **target attribute C**, which is called the **class attribute**.
- In our subsequent discussions, we consider C separately from attributes in A due to its special status, i.e., we assume that C is not in A.
- The class attribute C has a set of discrete values, i.e.,  $C = \{c_1, c_2, \dots, c_{|C|}\}$ , where  $|C|$  is the number of classes and  $|C| \geq 2$ . A class value is also **called a class label**.



# Basic Concepts



- A class value is also called a **class label**.
- A data set for learning is simply a relational table. Each data record describes a piece of “**past experience**”.
- In the machine learning and data mining literature, **a data record is also called an example, an instance, a case or a vector**.
- A data set basically consists of a set of examples or instances

# Basic Concepts



- The data set used for learning is called the **training data (or the training set)**.
- After a model is learned or built from the training data by a learning algorithm, it is evaluated using a set of **test data** (or unseen data) to assess the **model accuracy**.
- The accuracy of a classification model on a test set is defined as:

# Basic Concepts



The accuracy of a classification model on a test set is defined as:

$$Accuracy = \frac{\text{Number of correct classifications}}{\text{Total number of test cases}},$$



- A correct classification means that the learned model predicts the same class as the original class of the test case.

# Steps of learning



In step 1, a learning algorithm uses the training data to generate a classification model. This step is also called the **training step or training phase**.

In step 2, the learned model is tested using the test set to obtain the classification accuracy. This step is called the **testing step or testing phase**.

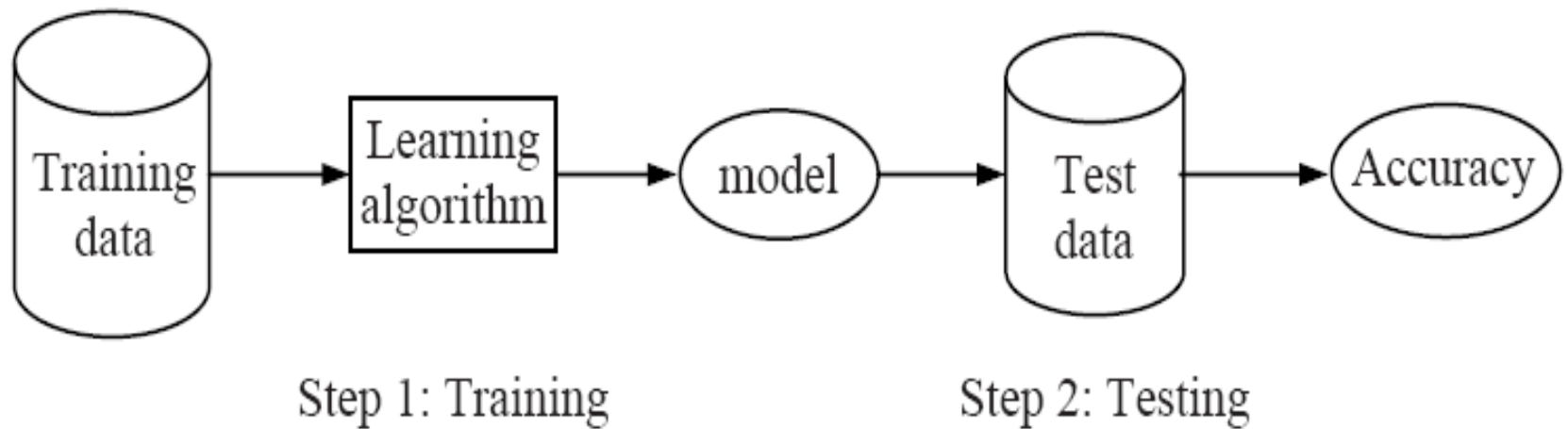


- If **the accuracy** of the learned model on the test data is satisfactory, the model can be used in real-world tasks to predict classes of new cases (which do not have classes).
- If the accuracy is not satisfactory, we need to go back and choose a different learning algorithm and/or do some further processing of the data (this step is called **data pre-processing**)



- A practical learning task typically involves many iterations of these steps before a satisfactory model is built.

# Basic Learning Process: Training and Testing



**Fig. 3.1.** The basic learning process: training and testing

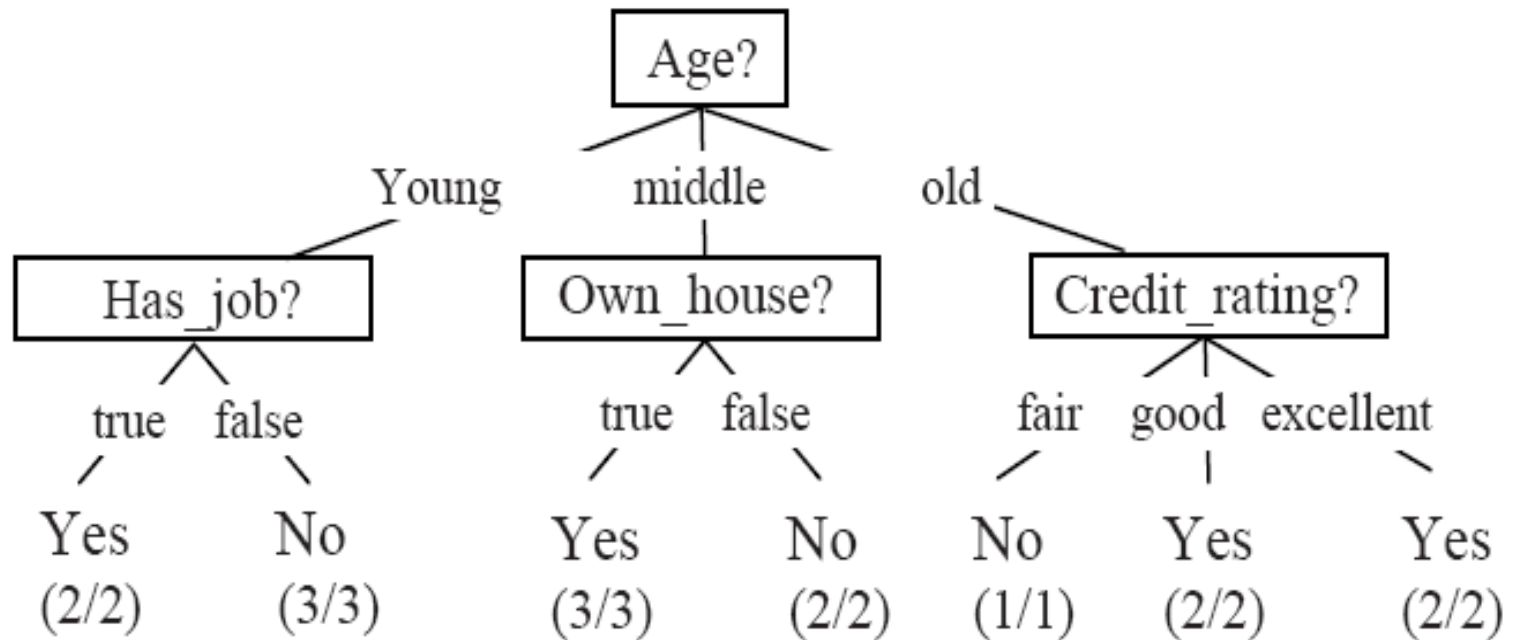


## Supervised Learning Algorithms-Decision Tree Induction



- **Decision tree learning** is one of the most widely used techniques for classification.
- Its classification accuracy is competitive with other learning methods, and it is very efficient.
- The learned classification model is represented as a tree, called a **decision tree**.

# Decision Tree Induction



**Fig. 3.2.** A decision tree for the data in [Table 3.1](#)

**Table 3.1.** A loan application data set

ID	Age	Has_job	Own_house	Credit_rating	Class
1	young	false	false	fair	No
2	young	false	false	good	No
3	young	true	false	good	Yes
4	young	true	true	fair	Yes
5	young	false	false	fair	No
6	middle	false	false	fair	No
7	middle	false	false	good	No
8	middle	true	true	good	Yes
9	middle	false	true	excellent	Yes
10	middle	false	true	excellent	Yes
11	old	false	true	excellent	Yes
12	old	false	true	good	Yes
13	old	true	false	good	Yes
14	old	true	false	excellent	Yes
15	old	false	false	fair	No

# Decision Tree Induction



**Example 2:** Fig. 3.2 shows a possible decision tree learnt from the data in Table 3.1.

- The tree has two types of nodes, **decision nodes** (which are internal nodes) and **leaf nodes**.
- A **decision node** specifies some test (i.e., asks a question) on a single attribute.
- A **leaf node** indicates a class.

# Decision Tree Induction



- The root node of the decision tree in Fig. 3.2 is Age, which basically asks the question: what is the age of the applicant? It has three possible answers or **outcomes**, which are the three possible values of Age.
- These three values form three tree branches/edges.
- The other internal nodes have the same meaning.

# Decision Tree Induction



- Each leaf node gives a class value (Yes or No).
- $(x/y)$  below each class means that  $x$  out of  $y$  training examples that reach this leaf node have the class of the leaf.
- For instance, the class of the left most leaf node is Yes.
- Two training examples (examples 3 and 4 in Table 3.1) reach here and both of them are of class Yes.

# Decision Tree Induction

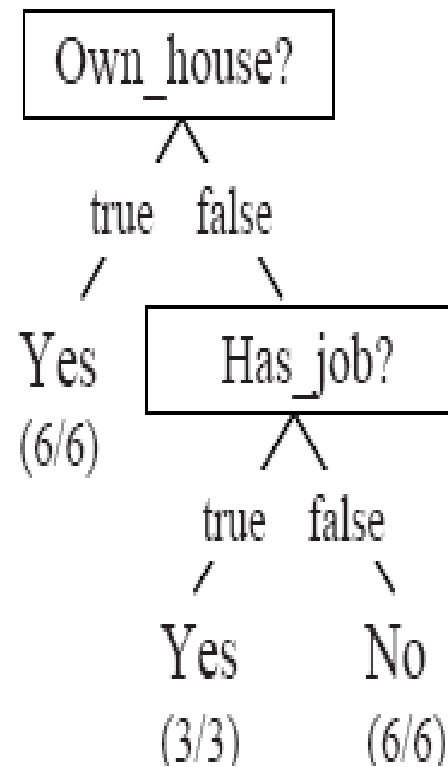


- To use the decision tree in **testing**, we traverse the tree top-down according to the attribute values of the given test instance until we reach a leaf node.
- The class of the leaf is the predicted class of the test instance.



- A decision tree is constructed by partitioning the training data so that the resulting subsets are as pure as possible.
- A **pure subset** is one that contains only training examples of a single class.
- There are many possible trees that can be learned from the data.
- For example, Fig. 3.3 gives another decision tree, which is much smaller and is also able to partition the training data perfectly according to their classes





**Fig. 3.3.** A smaller tree for the data set in [Table 3.1](#)



- For most real-life data sets, the examples that reach a particular leaf node are not of the same class, i.e.,  $x \neq y$ .



- **Example 4:** The tree in [Fig. 3.3](#) generates three rules.  
“,” means “and”.
- $\text{Own\_house} = \text{true} \rightarrow \text{Class} = \text{Yes}$  [sup=6/15, conf=6/6]
- $\text{Own\_house} = \text{false}, \text{Has\_job} = \text{true} \rightarrow \text{Class} = \text{Yes}$  [sup=3/15, conf=3/3]
- $\text{Own\_house} = \text{false}, \text{Has\_job} = \text{false} \rightarrow \text{Class} = \text{No}$  [sup=6/15, conf=6/6].



- We can see that these rules are of the same format as association rules.
- However, the rules above are only a small subset of the rules that can be found in the data of [Table 3.1](#).
- For instance, the decision tree in [Fig. 3.3](#) does not find the following rule:
- Age = young, Has\_job = false  $\rightarrow$  Class = No [sup=3/15, conf=3/3].

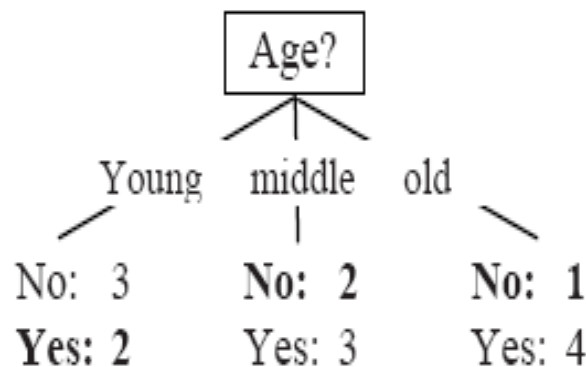


- Thus, we say that a decision tree only finds a subset of rules that exist in data, which is sufficient for classification.

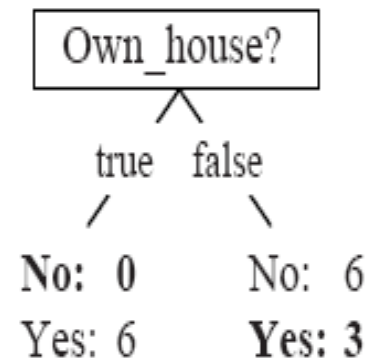
# Impurity Function



**Example 5:** Fig. 3.5 shows two possible root nodes for the data in Table 3.1.



(A)



(B)



$$\textit{entropy}(D) = - \sum_{j=1}^{|C|} \text{Pr}(c_j) \log_2 \text{Pr}(c_j)$$

$$\sum_{j=1}^{|C|} \text{Pr}(c_j) = 1,$$

**Example 6:** Assume we have a data set  $D$  with only two classes, positive and negative. Let us see the entropy values for three different compositions of positive and negative examples:

1. The data set  $D$  has 50% positive examples ( $\Pr(\text{positive}) = 0.5$ ) and 50% negative examples ( $\Pr(\text{negative}) = 0.5$ ).

$$\text{entropy}(D) = -0.5 \times \log_2 0.5 - 0.5 \times \log_2 0.5 = 1.$$

2. The data set  $D$  has 20% positive examples ( $\Pr(\text{positive}) = 0.2$ ) and 80% negative examples ( $\Pr(\text{negative}) = 0.8$ ).

$$\text{entropy}(D) = -0.2 \times \log_2 0.2 - 0.8 \times \log_2 0.8 = 0.722.$$

3. The data set  $D$  has 100% positive examples ( $\Pr(\text{positive}) = 1$ ) and no negative examples, ( $\Pr(\text{negative}) = 0$ ).

$$\text{entropy}(D) = -1 \times \log_2 1 - 0 \times \log_2 0 = 0.$$





- We can see a trend: When the data becomes purer and purer, the entropy value becomes smaller and smaller.
- In fact, it can be shown that for this binary case (two classes), when  $\Pr(\text{positive}) = 0.5$  and  $\Pr(\text{negative}) = 0.5$  the entropy has the maximum value, i.e., 1 bit.
- When all the data in  $D$  belong to one class the entropy has the minimum value, 0 bit.



- $\Pr(c_j)$  is the probability of class  $c_j$  in data set  $D$ , which is the number of examples of class  $c_j$  in  $D$  divided by the total number of examples in  $D$ .

1. The data set  $D$  has 50% positive examples ( $\Pr(\text{positive}) = 0.5$ ) and 50% negative examples ( $\Pr(\text{negative}) = 0.5$ ).

$$\text{entropy}(D) = -0.5 \times \log_2 0.5 - 0.5 \times \log_2 0.5 = 1.$$

2. The data set  $D$  has 20% positive examples ( $\Pr(\text{positive}) = 0.2$ ) and 80% negative examples ( $\Pr(\text{negative}) = 0.8$ ).

$$\text{entropy}(D) = -0.2 \times \log_2 0.2 - 0.8 \times \log_2 0.8 = 0.722.$$

3. The data set  $D$  has 100% positive examples ( $\Pr(\text{positive}) = 1$ ) and no negative examples, ( $\Pr(\text{negative}) = 0$ ).

$$\text{entropy}(D) = -1 \times \log_2 1 - 0 \times \log_2 0 = 0.$$



- The **entropy** measures the amount of **impurity** or disorder in the data.
- That is exactly what we need in decision tree learning.

1. Given a data set  $D$ , we first use the entropy function (Equation 2) to compute the impurity value of  $D$ , which is  $entropy(D)$ . The **impurityEval-1** function in line 7 of Fig. 3.4 performs this task.
2. Then, we want to know which attribute can reduce the impurity most if it is used to partition  $D$ . To find out, every attribute is evaluated (lines 8–10 in Fig. 3.4). Let the number of possible values of the attribute  $A_i$  be  $v$ . If we are going to use  $A_i$  to partition the data  $D$ , we will divide  $D$  into  $v$  disjoint subsets  $D_1, D_2, \dots, D_v$ . The entropy after the partition is

$$entropy_{A_i}(D) = \sum_{j=1}^v \frac{|D_j|}{|D|} \times entropy(D_j). \quad (3)$$

The **impurityEval-2** function in line 9 of Fig. 3.4 performs this task.

3. The information gain of attribute  $A_i$  is computed with:

$$gain(D, A_i) = entropy(D) - entropy_{A_i}(D). \quad (4)$$

**Example 7:** Let us compute the gain values for attributes **Age**, **Own\_house** and **Credit\_Rating** using the whole data set  $D$  in [Table 3.1](#), i.e., we evaluate for the root node of a decision tree.

First, we compute the entropy of  $D$ . Since  $D$  has 6 No class training examples, and 9 Yes class training examples, we have

$$\text{entropy}(D) = -\frac{6}{15} \times \log_2 \frac{6}{15} - \frac{9}{15} \times \log_2 \frac{9}{15} = 0.971.$$

We then try **Age**, which partitions the data into 3 subsets (as **Age** has three possible values)  $D_1$  (with **Age**=young),  $D_2$  (with **Age**=middle), and  $D_3$  (with **Age**=old). Each subset has five training examples. In [Fig. 3.5](#), we also see the number of No class examples and the number of Yes examples in each subset (or in each branch).

$$\begin{aligned}
 entropy_{Age}(D) &= \frac{5}{15} \times entropy(D_1) + \frac{5}{15} \times entropy(D_2) + \frac{5}{15} \times entropy(D_3) \\
 &= \frac{5}{15} \times 0.971 + \frac{5}{15} \times 0.971 + \frac{5}{15} \times 0.722 = 0.888.
 \end{aligned}$$

Likewise, we compute for **Own\_house**, which partitions  $D$  into two subsets,  $D_1$  (with **Own\_house=true**) and  $D_2$  (with **Own\_house=false**).

$$\begin{aligned}
 entropy_{Own\_house}(D) &= \frac{6}{15} \times entropy(D_1) + \frac{9}{15} \times entropy(D_2) \\
 &= \frac{6}{15} \times 0 + \frac{9}{15} \times 0.918 = 0.551.
 \end{aligned}$$

Similarly, we obtain  $entropy_{Has\_job}(D) = 0.647$ , and  $entropy_{Credit\_rating}(D) = 0.608$ . The gains for the attributes are:

$$gain(D, \text{Age}) = 0.971 - 0.888 = 0.083$$

$$gain(D, \text{Own\_house}) = 0.971 - 0.551 = 0.420$$

$$gain(D, \text{Has\_job}) = 0.971 - 0.647 = 0.324$$

$$gain(D, \text{Credit\_rating}) = 0.971 - 0.608 = 0.363.$$



- Own\_house is the best attribute for the root node





The idea is the following:

- 1. Given a data set  $D$ , we first use the entropy function (Equation 2) to compute the impurity value of  $D$ , which is  $entropy(D)$ .
- The **impurityEval-1** function in line 7 of Fig. 3.4 performs this task.



- Then, we want to know which attribute can reduce the impurity most if it is used to partition  $D$ .
- To find out, every attribute is evaluated (lines 8–10 in Fig. 3.4).
- Let the number of possible values of the attribute  $A_i$  be  $v$ .
- If we are going to use  $A_i$  to partition the data  $D$ , we will divide  $D$  into  $v$  disjoint subsets  $D_1, D_2, \dots, D_v$ . The entropy after the partition is



$$\textit{entropy}_A(D) = \sum_{j=1}^v \frac{|D_j|}{|D|} \times \textit{entropy}(D_j).$$

# Information gain



$$\text{gain}(D, A_i) = \text{entropy}(D) - \text{entropy}_{A_i}(D).$$

```

Algorithm decisionTree( $D, A, T$ )
1   if  $D$  contains only training examples of the same class  $c_j \in C$  then
2       make  $T$  a leaf node labeled with class  $c_j$ ;
3   elseif  $A = \emptyset$  then
4       make  $T$  a leaf node labeled with  $c_j$ , which is the most frequent class in  $D$ 
5   else //  $D$  contains examples belonging to a mixture of classes. We select a single
6       // attribute to partition  $D$  into subsets so that each subset is purer
7        $p_0 = \text{impurityEval-1}(D)$ ;
8       for each attribute  $A_i \in A$  ( $=\{A_1, A_2, \dots, A_k\}$ ) do
9            $p_i = \text{impurityEval-2}(A_i, D)$ 
10      endfor
11      Select  $A_g \in \{A_1, A_2, \dots, A_k\}$  that gives the biggest impurity reduction,
          computed using  $p_0 - p_i$ ;
12      if  $p_0 - p_g < \text{threshold}$  then //  $A_g$  does not significantly reduce impurity  $p_0$ 
13          make  $T$  a leaf node labeled with  $c_j$ , the most frequent class in  $D$ .
14      else //  $A_g$  is able to reduce impurity  $p_0$ 
15          Make  $T$  a decision node on  $A_g$ ;
16          Let the possible values of  $A_g$  be  $v_1, v_2, \dots, v_m$ . Partition  $D$  into  $m$ 
              disjoint subsets  $D_1, D_2, \dots, D_m$  based on the  $m$  values of  $A_g$ .
17          for each  $D_j$  in  $\{D_1, D_2, \dots, D_m\}$  do
18              if  $D_j \neq \emptyset$  then
19                  create a branch (edge) node  $T_j$  for  $v_j$  as a child node of  $T$ ;
20                  decisionTree( $D_j, A - \{A_g\}, T_j$ ) //  $A_g$  is removed
21              endif
22          endfor
23      endif
24  endif

```

**Fig. 3.4.** A decision tree learning algorithm

# Classifier Evaluation



- **Evaluation Methods**
- **Precision, Recall, F-score and Breakeven Point**
- **Receiver Operating Characteristic Curve**
- **Lift Curve**

# Classifier Evaluation



- After a classifier is constructed, it needs to be evaluated for accuracy.

## Evaluation Methods

- **Holdout Set:** The available data  $D$  is divided into two disjoint subsets, the **training set**  $D_{train}$  and the **test set**  $D_{test}$ ,  $D = D_{train} \cup D_{test}$  and  $D_{train} \cap D_{test} = \emptyset$ .
- The test set is also called the **holdout set**.

# Classifier Evaluation



- The training set is used for **learning a classifier** and the test set is used for **evaluating the classifier**.
- To partition  $D$  into training and test sets, we can use a few approaches:
  1. We **randomly sample a set of training examples from  $D$  for learning and use the rest for testing.**



# Classifier Evaluation



2. If the data is collected over time, **then we can use the earlier part of the data for training/learning and the later part of the data for testing.**

# Classifier Evaluation



- **Multiple Random Sampling:** When the available data set is small, using the above methods can be unreliable
- One approach to deal with the problem is to perform the above random sampling  $n$  times.
- Each time a different training set and a different test set are produced.
- This produces  $n$  accuracies. The final estimated accuracy on the data is the average of the  $n$  accuracies.

# Classifier Evaluation



- **Cross-Validation:** When the data set is small, the  **$n$ -fold cross-validation** method is very commonly used.
- In this method, the available data is partitioned into  $n$  equal-size disjoint subsets.
- Each subset is then used as the test set and the remaining  $n-1$  subsets are combined as the training set to learn a classifier.
- This procedure is then run  $n$  times, which gives  $n$  accuracies.

# Classifier Evaluation



- The final estimated accuracy of learning from this data set is the average of the  $n$  accuracies.
- 10-fold and 5-fold cross-validations are often used.

# Classifier Evaluation



- Special case of cross-validation is the **leave-one-out cross-validation**.
- In this method, each fold of the cross validation has only a single test example and all the rest of the data is used in training.
- That is, if the original data has  $m$  examples, then this is  $m$ -fold cross-validation.
- This method is normally used when the available data is very small. It is not efficient for a
- large data set as  $m$  classifiers need to be built

# Classifier Evaluation



- **Precision, Recall, F-score and Breakeven Point**
- In some applications, we are only interested in one class.
- This is particularly true for text and Web applications.
- For eg **we may be interested in only the documents or web pages of a particular topic.**

# Classifier Evaluation



- Also, in classification network intrusion and financial fraud detection, we are typically interested in only the minority class.
- The class that the user is interested in is commonly
- called the **positive class, and the rest negative classes** (the negative classes may be combined into one **negative class**).
- Accuracy is not a suitable measure in such cases because we may achieve a very high accuracy but
- may not identify a single intrusion.

# Precision, Recall, F-score and Breakeven Point



- **Precision** and **recall** are more suitable in such applications because they measure how precise and how complete the classification is on the positive class.
- **A confusion matrix contains information about actual and predicted results given by a classifier.**



# Precision, Recall, F-score and Breakeven Point



**Table 3.2.** Confusion matrix of a classifier

	Classified positive	Classified negative
Actual positive	TP	FN
Actual negative	FP	TN

where

*TP*: the number of correct classifications of the positive examples (**true positive**)

*FN*: the number of incorrect classifications of positive examples (**false negative**)

*FP*: the number of incorrect classifications of negative examples (**false positive**)

*TN*: the number of correct classifications of negative examples (**true negative**)

Based on the confusion matrix, the precision ( $p$ ) and recall ( $r$ ) of the positive class are defined as follows:

# Precision, Recall, F-score and Breakeven Point



- Based on the confusion matrix, the precision ( $p$ ) and recall ( $r$ ) of the positive class are defined as follows:

# Precision, Recall, F-score and Breakeven Point



$$p = \frac{TP}{TP + FP} \quad r = \frac{TP}{TP + FN}$$

# Precision, Recall, F-score and Breakeven Point



- Precision  $p$  is the number of correctly classified positive examples divided by the total number of examples that are classified as positive.
- Recall  $r$  is the number of correctly classified positive examples divided by the total number of actual positive examples in the test set.
- The intuitive meanings of these two measures are quite obvious.

# Precision, Recall, F-score and Breakeven Point



**Table 3.2.** Confusion matrix of a classifier

	Classified positive	Classified negative
Actual positive	TP	FN
Actual negative	FP	TN

where

*TP*: the number of correct classifications of the positive examples (**true positive**)

*FN*: the number of incorrect classifications of positive examples (**false negative**)

*FP*: the number of incorrect classifications of negative examples (**false positive**)

*TN*: the number of correct classifications of negative examples (**true negative**)

Based on the confusion matrix, the precision ( $p$ ) and recall ( $r$ ) of the positive class are defined as follows:



- A test data set has 100 positive examples and 1000 negative examples.
- After classification using a classifier, we have the following confusion matrix.
- Confusion matrix of a classifier.

**Table 3.3.** Confusion matrix of a classifier

	Classified positive	Classified negative
Actual positive	1	99
Actual negative	0	1000

This confusion matrix gives the precision  $p = 100\%$  and the recall  $r = 1\%$  because we only classified one positive example correctly and classified no negative examples wrongly. ■

# F-score



- If we need a single measure to compare different classifiers, the **F-score** is often used.

# F-score



$$F = \frac{2pr}{p+r}.$$



# F-score



- The F-score (also called the **F1-score**) is the harmonic mean of precision and recall.

# F-score



$$F = \frac{2}{\frac{1}{p} + \frac{1}{r}}.$$

# Break even point



- The break even point is when the precision and the recall are equal.
- This measure assumes that the test cases can be ranked by the classifier based on their likelihoods
- of being positive.

## Precision, Recall, F-score and Breakeven Point

**Example 11:** A test data set has 100 positive examples and 1000 negative examples. After classification using a classifier, we have the following confusion matrix ([Table 3.3](#)),

Table 3.3. Confusion matrix of a classifier

	Classified positive	Classified negative
Actual positive	1	99
Actual negative	0	1000

This confusion matrix gives the precision  $p = 100\%$  and the recall  $r = 1\%$  because we only classified one positive example correctly and classified no negative examples wrongly. ■

# Receiver Operating Characteristic Curve



- A receiver operating characteristic (ROC) curve is a plot of the **true positive rate** against the **false positive rate**.
- It is also commonly used to evaluate classification results on the positive class in two-class classification problems.

# Receiver Operating Characteristic Curve



- The classifier needs to rank the test cases according to their likelihoods of belonging to the positive class with the most likely positive case ranked at the top

# Receiver Operating Characteristic Curve



- The true positive rate ( $TPR$ ) is defined as the fraction of actual positive cases that are correctly classified.
- The false positive rate ( $FPR$ ) is defined as the fraction of actual negative cases that are classified to the positive class.

# Receiver Operating Characteristic Curve



$$TPR = \frac{TP}{TP + FN}.$$



# Receiver Operating Characteristic Curve



$$FPR = \frac{FP}{TN + FP}.$$

# Receiver Operating Characteristic Curve



- *TPR* is basically the recall of the positive class and is also called **sensitivity** in statistics.
- There is also another measure in statistics called **specificity**, which is the **true negative rate** (*TNR*), or the recall of the negative class.
- *TNR* is defined as follows

# Receiver Operating Characteristic Curve



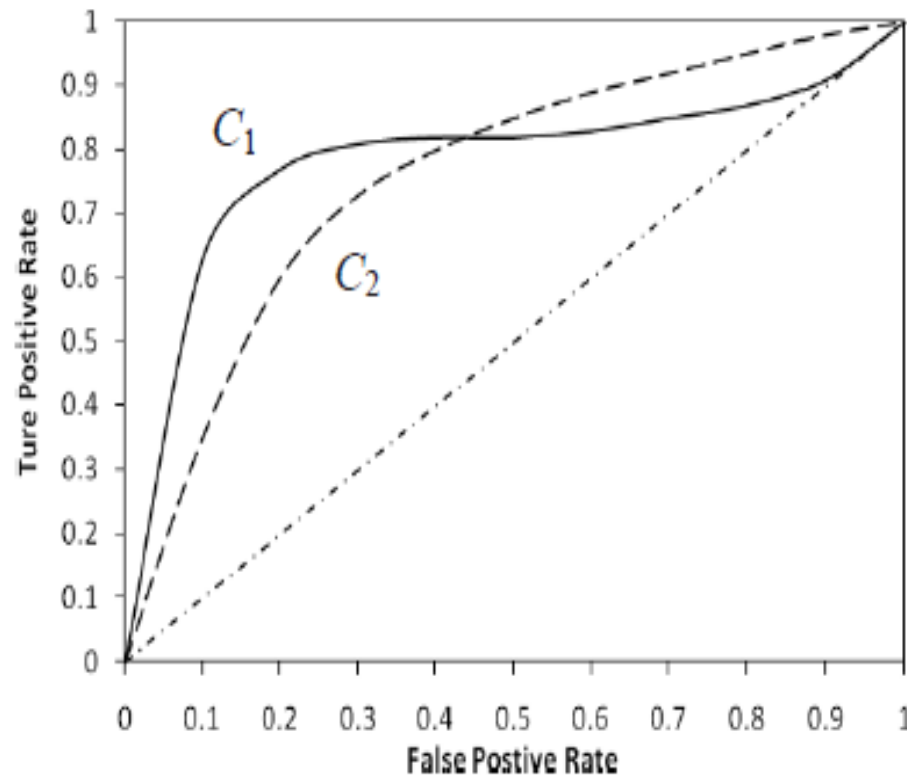
$$TNR = \frac{TN}{TN + FP}.$$

# Receiver Operating Characteristic Curve



$$FPR = 1 - \textit{specificity}.$$

# Receiver Operating Characteristic Curve



**Fig. 3.8.** ROC curves for two classifiers ( $C_1$  and  $C_2$ ) on the same data

# Receiver Operating Characteristic Curve



- We want to know which classifier is better.
- The answer is that when  $FPR$  is less than 0.43,  $C1$  is better, and when  $FPR$  is greater than 0.43,  $C2$  is better.

# Lift Curve



- The **lift curve** (also called the **lift chart**) is similar to the ROC curve.
- It is also for evaluation of two-class classification tasks, where the positive class is the target of interest and usually the rare class.
- It is often used in direct marketing applications to link classification results to costs and profits

# Lift Curve



- For example, a mail order company wants to send promotional materials to potential customers to sell an expensive watch.
- Since printing and postage cost money, the company needs to build a classifier to identify likely buyers, and only sends the promotional materials to them.
- The question is how many should be sent.
- To make the decision, the company needs to balance the cost and profit (if a watch is sold, the company makes a certain profit, but to send each letter there is a fixed cost). The lift curve provides a nice tool to enable the marketer to make the decision.



# Lift Curve



- A company wants to send promotional materials to potential buyers to sell an expensive brand of watches.
- It builds a classification model and tests it on a test data of 10,000 people (test cases) that they collected in the past.
- After classification and ranking, it decides to divide the
- test data into 10 bins with each bin containing 10% of the test cases or 1,000 cases.
- Out of the 1,000 cases in each bin, there are a certain number of positive cases (e.g., past buyers).

# Lift Curve



- The detailed results are listed in Table 3.5, which includes the number (#) of positive cases and the percentage (%) of positive cases in each bin, and the cumulative percentage for that bin.
- The cumulative percentages are used in drawing the lift curve which is given in Fig. 3.10.
- We can see that the lift curve is way above the baseline, which means that the learning is highly effective

# Lift Curve



- Suppose printing and postage cost \$1.00 for each letter, and the sale of each watch makes \$100 (assuming that each buyer only buys one watch).
- If the company wants to send promotional letters to 3000 people, it will make \$36,000, i.e.,
- $\$100 \times (210 + 120 + 60) - \$3,000 = \$36,000$

# Lift Curve

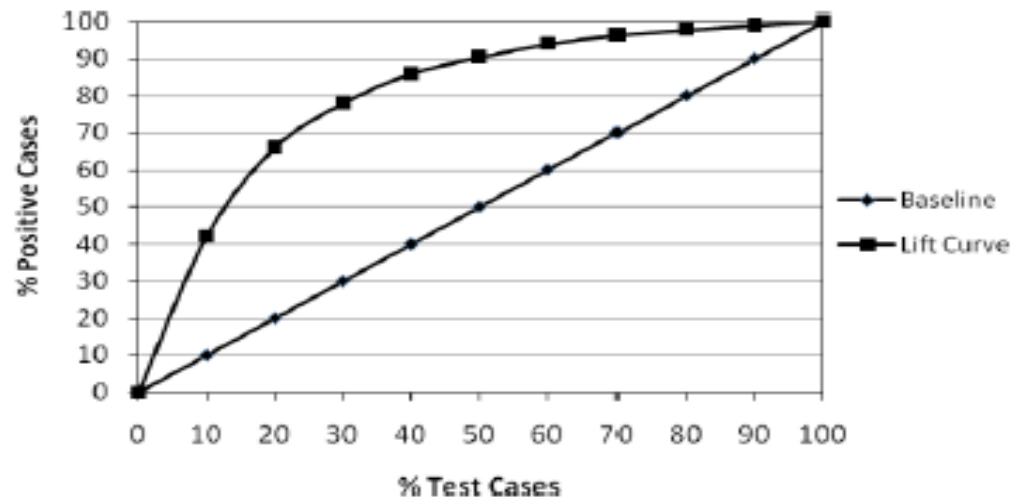


If the company wants to send promotional letters to 3000 people, it will make \$36,000, i.e.,

$$\$100 \times (210 + 120 + 60) - \$3,000 = \$36,000$$

**Table 3.5.** Classification results for the 10 bins

Bin	1	2	3	4	5	6	7	8	9	10
# of test cases	1000	1000	1000	1000	1000	1000	1000	1000	1000	1000
# of positive cases	210	120	60	40	22	18	12	7	6	5
% of positive cases	42.0%	24.0%	12%	8%	4.4%	3.6%	2.4%	1.4%	1.2%	1.0%
% cumulative	42.0%	66.0%	78.0%	86.0%	90.4%	94.0%	96.4%	97.8%	99.0%	100.0%



**Fig. 3.10.** Lift curve for the data shown in Table 3.5

# Rule Induction



- The set of rules can be used for classification as the tree.
- The process of learning such rules is called **rule induction** or **rule learning**.

## Sequential Covering.

- The basic idea of sequential covering is to learn a list of rules sequentially, one at a time, to cover the training data.
- After each rule is learned, the training examples covered by the rule are removed.
- Only the remaining data are used to find subsequent rules.
- Recall that a rule covers an example if the example satisfies the conditions of the rule

## Rule Induction- *Algorithm 1 (Ordered Rules)* (*Learn-one-rule-1*)



- ***Algorithm 1 (Ordered Rules)***
- In each iteration, a rule of any class may be found.
- Thus rules of different classes may intermix in the final rule list.
- The ordering of rules is important.
- $D$  is the training data.
- *RuleList* is the list of rules, which is initialized to empty set (line 1).
- *Rule* is the best rule found in each iteration.

# Rule Induction



- The function `learn-one-rule-1()` learns the *Rule* (lines 2 and 6).
- The stopping criteria for the while-loop can be of various kinds.
- Here we use  $D = \text{NULL}$  or *Rule* is  $\text{NULL}$  (a rule is not learned).
- Once a rule is learned from the data, it is inserted into *RuleList* at the end (line 4).
- All the training examples that are covered by the rule are removed from the data (line 5).

# Rule Induction



- The remaining data is used to find the next rule and so on.
- After rule learning ends, a **default class** is inserted at the end of *RuleList*.
- This is because there may still be some training examples that are not covered by any rule as no good rule can be found from them.
- or because some test cases may not be covered by any rule and thus cannot be classified.



### Algorithm sequential-covering-1( $D$ )

```
1   $RuleList \leftarrow \emptyset$ ;  
2   $Rule \leftarrow \text{learn-one-rule-1}(D)$ ;  
3  while  $Rule$  is not NULL AND  $D \neq \emptyset$  do  
4       $RuleList \leftarrow$  insert  $Rule$  at the end of  $RuleList$ ;  
5      Remove from  $D$  the examples covered by  $Rule$ ;  
6       $Rule \leftarrow \text{learn-one-rule-1}(D)$   
7  endwhile  
8  insert a default class  $c$  at the end of  $RuleList$ , where  $c$  is the majority class  
   in  $D$ ;  
9  return  $RuleList$ 
```

Fig. 3.11. The first rule learning algorithm based on sequential covering

# Rule Induction



The final list of rules is as follows:

$$\langle r_1, r_2, \dots, r_k, \text{default-class} \rangle$$

where  $r_i$  is a rule.

## Rule Induction-*Algorithm 2 (Ordered Classes)*

(Learn-one-rule-2)



- This algorithm learns all rules for each class together.
- After rule learning for one class is completed, it moves to the next class.
- Thus all rules for each class appear together in the rule list. The sequence of rules for each class is unimportant, but the rule subsets for different classes are ordered.
- Typically, the algorithm finds rules for the least frequent class first, then the second least frequent class and so on.
- This ensures that some rules are learned for rare classes.
- Otherwise, they may be dominated by frequent classes and end up with no rules if considered after frequent classes.

# Rule Induction

**Algorithm** sequential-covering-2( $D, C$ )

```
1  RuleList  $\leftarrow \emptyset$ ;                                // empty rule set at the beginning
2  for each class  $c \in C$  do
3      prepare data ( $Pos, Neg$ ), where  $Pos$  contains all the examples of class
         $c$  from  $D$ , and  $Neg$  contains the rest of the examples in  $D$ ;
4      while  $Pos \neq \emptyset$  do
5           $Rule \leftarrow \text{learn-one-rule-2}(Pos, Neg, c)$ ;
6          if  $Rule$  is NULL then
7              exit-while-loop
8          else  $RuleList \leftarrow$  insert  $Rule$  at the end of  $RuleList$ ;
9              Remove examples covered by  $Rule$  from ( $Pos, Neg$ )
10         endif
11     endwhile
12 endfor
13 return RuleList
```

Fig. 3.12. The second rule learning algorithm based on sequential covering

# Rule Induction



- The algorithm is given in Fig. 3.12. The data set  $D$  is split into two subsets,
- $Pos$  and  $Neg$ , where  $Pos$  contains all the examples of class  $c$  from  $D$ , and  $Neg$  the rest of the examples in  $D$  (line 3).
- $c$  is the class that the algorithm is working on now.
- Two stopping conditions for rule learning of
- each class are in line 4 and line 6.
- The other parts of the algorithm are quite similar to those of the first algorithm in Fig. 3.11

## Rule Learning: Learn-One-Rule Function

We now present the function `learn-one-rule()`, which works as follows: It starts with an empty set of conditions. In the first iteration, one condition is added. In order to find the best condition to add, all possible conditions are tried, which form **candidate rules**. A **condition** is of the form  $A_i \text{ op } v$ , where  $A_i$  is an attribute and  $v$  is a value of  $A_i$ . We also called it an **attribute-value** pair. For a discrete attribute,  $\text{op}$  is “=”. For a continuous attribute,  $\text{op} \in \{>, \leq\}$ . The algorithm evaluates all the candidates to find the best one (the rest are discarded). After the first best condition is added, it tries to add the second condition and so on in the same fashion until some stopping condition is satisfied. Note that we omit the rule class here because it is implied, i.e., the majority class of the data covered by the conditions.

# Rule Learning: Learn-One-Rule Function



- keeping  $k$  best sets of conditions ( $k > 1$ ) in each iteration.
- This is called the **beam search** ( $k$  beams)

# Rule Learning: Learn-One-Rule Function



- This function uses beam search (Fig. 3.13).
- The number of beams is  $k$ .
- *BestCond* stores the conditions of the rule to be returned.
- The class is omitted as it is the majority class of the data covered by *BestCond*.
- *candidate-CondSet* stores the current best condition sets (which are the frontier beams) and its size is less than or equal to  $k$ .
- Each condition set contains a set of conditions connected by “and” (conjunction). *newCandidateCondSet* stores all the new candidate condition sets after adding each attribute-value
- pair (a possible condition) to every candidate in *candidateCondSet* (lines 5–11). Lines 13–17 update the *BestCond*.



# Rule Learning: Learn-One-Rule Function



- Specifically, an evaluation function is used to assess whether each new candidate condition set is better
- than the existing best condition set *BestCond* (line 14).
- If so, it replaces the current *BestCond* (line 15). Line 18 updates *candidateCondSet*, which selects
- $k$  new best condition sets (new beams).

**Function learn-one-rule-1( $D$ )**

```
1    $BestCond \leftarrow \emptyset$ ; // rule with no condition.
2    $candidateCondSet \leftarrow \{BestCond\}$ ;
3    $attributeValuePairs \leftarrow$  the set of all attribute-value pairs in  $D$  of the form
   ( $A_i \text{ op } v$ ), where  $A_i$  is an attribute and  $v$  is a value or an interval;
4   while  $candidateCondSet \neq \emptyset$  do
5        $newCandidateCondSet \leftarrow \emptyset$ ;
6       for each candidate  $cond$  in  $candidateCondSet$  do
7           for each attribute-value pair  $a$  in  $attributeValuePairs$  do
8                $newCond \leftarrow cond \cup \{a\}$ ;
9                $newCandidateCondSet \leftarrow newCandidateCondSet \cup \{newCond\}$ 
10          endfor
11      endfor
12      remove duplicates and inconsistencies, e.g.,  $\{A_i = v_1, A_i = v_2\}$ ;
13      for each candidate  $newCond$  in  $newCandidateCondSet$  do
14          if  $evaluation(newCond, D) > evaluation(BestCond, D)$  then
15               $BestCond \leftarrow newCond$ 
16          endif
17      endfor
18       $candidateCondSet \leftarrow$  the  $k$  best members of  $newCandidateCondSet$ 
        according to the results of the evaluation function;
19  endwhile
20  if  $evaluation(BestCond, D) - evaluation(\emptyset, D) > threshold$  then
21      return the rule: " $BestCond \rightarrow c$ " where  $c$  is the majority class of the data
        covered by  $BestCond$ 
22  else return NULL
23  endif
```

**Fig. 3.13.** The learn-one-rule-1 function

# Rule Learning: Learn-One-Rule Function



**Function** *evaluation*(*BestCond*, *D*)

- 1  $D' \leftarrow$  the subset of training examples in  $D$  covered by  $BestCond$ ;
- 2  $entropy(D') = -\sum_{j=1}^{|C|} \Pr(c_j) \log_2 \Pr(c_j)$ ;
- 3 **return**  $-entropy(D')$  // since entropy measures impurity.

Fig. 3.14. The entropy based evaluation function

## *Learn-One-Rule-2*



- In the `learn-one-rule-2()` function (Fig. 3.14), a rule is first generated and then it is pruned. This method starts by splitting the positive and negative training data *Pos* and *Neg*, into growing and pruning sets. The growing sets, *GrowPos* and *GrowNeg*, are used to generate a rule, called *BestRule*.
- The pruning sets, *PrunePos* and *PruneNeg* are used to prune the rule because *BestRule* may overfit the data.

**growRule()** function: growRule() generates a rule (called *BestRule*) by repeatedly adding a condition to its condition set that maximizes an evaluation function until the rule covers only some positive examples in *GrowPos* but no negative examples in *GrowNeg*. This is basically the same as lines 4–17 in Fig. 3.13, but without beam search (i.e., only the best rule is kept in each iteration). Let the current partially developed rule be *R*:

$$R: \quad av_1, \dots, av_k \rightarrow class$$

where each  $av_j$  is a condition (an attribute-value pair). By adding a new condition  $av_{k+1}$ , we obtain the rule  $R^+ : av_1, \dots, av_k, av_{k+1} \rightarrow class$ . The evaluation function for  $R^+$  is the following **information gain** criterion (which is different from the gain function used in decision tree learning):

$$gain(R, R^+) = p_1 \times \left( \log_2 \frac{p_1}{p_1 + n_1} - \log_2 \frac{p_0}{p_0 + n_0} \right) \quad (14)$$

where  $p_0$  (respectively,  $n_0$ ) is the number of positive (negative) examples covered by *R* in *Pos* (*Neg*), and  $p_1$  ( $n_1$ ) is the number of positive (negative) examples covered by  $R^+$  in *Pos* (*Neg*). The GrowRule() function simply re-

# Classification Based on Associations



- **Class association rules (CAR)**, may be used for classification.
- Classification Using Class Association Rules
- A class association rule (CAR) is an association rule with only a class label on the right-hand side of the rule .
- $\text{Own\_house} = \text{false}, \text{Has\_job} = \text{true} \rightarrow \text{Class} = \text{Yes}$   
[sup=3/15, conf=3/3],
- There is no difference between rules from a decision tree (or a rule induction system) and CARs

# Classification Based on Associations



- The differences are in the mining processes and the final rule sets.
- CAR mining finds all rules in data that satisfy the user-specified minimum support (minsup) and minimum confidence (minconf) constraints.
- A decision tree or a rule induction system finds only a **subset** of the rules (expressed as a tree or a list of rules) for classification.

# Classification Based on Associations



**Example 15:** Recall that the decision tree in Fig. 3.3 gives the following three rules:

Own\_house = true  $\rightarrow$  Class = Yes [sup=6/15, conf=6/6]

Own\_house = false, Has\_job = true  $\rightarrow$  Class=Yes [sup=3/15, conf=3/3]

Own\_house = false, Has\_job = false  $\rightarrow$  Class=No [sup=6/15, conf=6/6].

However, there are many other rules that exist in data, e.g.,

Age = young, Has\_job = true  $\rightarrow$  Class=Yes [sup=2/15, conf=2/2]

Age = young, Has\_job = false  $\rightarrow$  Class=No [sup=3/15, conf=3/3]

Credit\_rating = fair  $\rightarrow$  Class=No [sup=4/15, conf=4/5]

and many more, if we use minsup =  $2/15 = 13.3\%$  and minconf = 70%. ■



# Classification Based on Associations



- Decision tree learning and rule induction do not use the minsup or minconf constraint.
- Thus, some rules that they find can have very low supports, which, of course, are likely to be pruned because the chance that they overfit the training data is high.

# Classification Based on Associations



- CAR mining does not use continuous (numeric) attributes, while decision trees deal with continuous attributes naturally.
- Rule induction can use continuous attributes as well.

# *Mining Class Association Rules for Classification*



- Issues related to CAR mining for classification.
- **Rule Pruning:** CAR rules are highly redundant, and many of them are not statistically significant (which can cause overfitting).
- Rule pruning is thus needed.
- **Multiple Minimum Class Supports:**
- A single minsup is inadequate for mining CARs because many practical classification data sets have uneven class distributions, i.e., some classes cover a large proportion of the data, while others cover only a very small proportion.

# *Mining Class Association Rules for Classification*

- Suppose we have a dataset with two classes,  $Y$  and  $N$ . 99% of the data belong to the  $Y$  class, and only 1% of the data belong to the  $N$  class.
- If we set  $\text{minsup} = 1.5\%$ , we will not find any rule for class  $N$ .
- To solve the problem, we need to lower down the  $\text{minsup}$ .
- Suppose we set  $\text{minsup} = 0.2\%$ . Then, we may find a huge number of overfitting rules for class  $Y$  because  $\text{minsup} = 0.2\%$  is too low for class  $Y$ .

# *Mining Class Association Rules for Classification*



- Multiple minimum class supports can be applied to deal with the problem.
- We can assign a different **minimum class support**  $minsup_i$  for each class  $c_i$ , i.e., all the rules of class  $c_i$  must satisfy  $minsup_i$ .
- Alternatively, we can provide one single total  $minsup$ , denoted by  $t\_minsup$ , which is then
- distributed to each class according to the class distribution.

# *Mining Class Association Rules for Classification*

- $minsup_i = t\_minsup * sup(ci)$ .
- where  $sup(ci)$  is the support of class  $ci$  in training data.
- The formula gives frequent classes higher minsups and infrequent classes lower minsups.
- **Parameter Selection:** The parameters used in CAR mining are the minimum supports and the minimum confidences.
- One minimum confidence is sufficient as long as it is not set too high.

# *Mining Class Association Rules for Classification*



- **Data Formats:** The algorithm for CAR mining given in Sect. 2.5.2 is for mining transaction data sets.
- However, many classification data sets are in the table format.

# *Mining Class Association Rules for Classification*

## ***Classifier Building***

- **Use the Strongest Rule:**. The strength of a rule
- can be measured in various ways, e.g., based on confidence,  $\chi^2$  test, test, or a combination of both support and confidence values.



## *Mining Class Association Rules for Classification*

**Select a Subset of the Rules to Build a Classifier:** The representative method of this category is the one used in the CBA system. The method is similar to the sequential covering method, but applied to class association rules with additional enhancements as discussed above.

Let the set of all discovered CARs be  $S$ . Let the training data set be  $D$ . The basic idea is to select a subset  $L (\subseteq S)$  of high confidence rules to cover the training data  $D$ . The set of selected rules, including a default class, is then used as the classifier. The selection of rules is based on a total order defined on the rules in  $S$ .

# *Mining Class Association Rules for Classification*

**Definition:** Given two rules,  $r_i$  and  $r_j$ ,  $r_i \succ r_j$  (also called  $r_i$  precedes  $r_j$  or  $r_i$  has a higher precedence than  $r_j$ ) if

1. the confidence of  $r_i$  is greater than that of  $r_j$ , or
2. their confidences are the same, but the support of  $r_i$  is greater than that of  $r_j$ , or
3. both the confidences and supports of  $r_i$  and  $r_j$  are the same, but  $r_i$  is generated earlier than  $r_j$ .

A CBA classifier  $L$  is of the form:

$$L = \langle r_1, r_2, \dots, r_k, \text{default-class} \rangle$$

# A simple classifier building algorithm

## Algorithm CBA( $S, D$ )

```
1   $S = \text{sort}(S);$            // sorting is done according to the precedence >
2   $RuleList = \emptyset;$       // the rule list classifier
3  for each rule  $r \in S$  in sequence do
4      if  $D \neq \emptyset$  AND  $r$  classifies at least one example in  $D$  correctly then
5          delete from  $D$  all training examples covered by  $r$ ;
6          add  $r$  at the end of  $RuleList$ 
7      endif
8  endfor
9  add the majority class as the default class at the end of  $RuleList$ 
```

Fig. 3.16. A simple classifier building algorithm

# Support Vector Machines



- **Support vector machines** (SVM) is another type of learning system ,which has many desirable qualities that make it one of most popular algorithms.
- SVM is considered as the the most accurate algorithm for text classification.

# Support Vector Machines



- The class that the user is interested in is commonly
- called the **positive class**, and the rest **negative classes** (the negative classes
- may be combined into one negative class).

# Support Vector Machines

In general, SVM is a **linear learning** system that builds two-class classifiers. Let the set of training examples  $D$  be

$$\{(\mathbf{x}_1, y_1), (\mathbf{x}_2, y_2), \dots, (\mathbf{x}_n, y_n)\},$$

where  $\mathbf{x}_i = (x_{i1}, x_{i2}, \dots, x_{ir})$  is a  $r$ -dimensional **input vector** in a real-valued space  $X \subseteq \mathcal{R}^r$ ,  $y_i$  is its **class label** (output value) and  $y_i \in \{1, -1\}$ . 1 denotes the positive class and -1 denotes the negative class. Note that we use slightly different notations in this section. We use  $y$  instead of  $c$  to represent a class because  $y$  is commonly used to represent a class in the SVM literature. Similarly, each data instance is called an **input vector** and denoted by a bold face letter. In the following, we use bold face letters for all vectors.

To build a classifier, SVM finds a linear function of the form

# Support Vector Machines



- SVM are based on the idea of finding hyperplanes that best divides a data set into two classes.
- In general we plot each data item as a point in  $r$  dimensional space.
- $r$  is the number of features we have.
- The value of each feature is the value of a particular coordinate

# Support Vector Machines

To build a classifier, SVM finds a linear function of the form

$$f(\mathbf{x}) = \langle \mathbf{w} \cdot \mathbf{x} \rangle + b \quad (35)$$

so that an input vector  $\mathbf{x}_i$  is assigned to the positive class if  $f(\mathbf{x}_i) \geq 0$ , and to the negative class otherwise, i.e.,

$$y_i = \begin{cases} 1 & \text{if } \langle \mathbf{w} \cdot \mathbf{x}_i \rangle + b \geq 0 \\ -1 & \text{if } \langle \mathbf{w} \cdot \mathbf{x}_i \rangle + b < 0 \end{cases} \quad (36)$$

Hence,  $f(\mathbf{x})$  is a real-valued function  $f: X \subseteq \mathcal{R}^r \rightarrow \mathcal{R}$ .  $\mathbf{w} = (w_1, w_2, \dots, w_r) \in \mathcal{R}^r$  is called the **weight vector**.  $b \in \mathcal{R}$  is called the **bias**.  $\langle \mathbf{w} \cdot \mathbf{x} \rangle$  is the **dot product** of  $\mathbf{w}$  and  $\mathbf{x}$  (or **Euclidean inner product**). Without using vector notation, Equation (35) can be written as:



# Support Vector Machines

$$f(x_1, x_2, \dots, x_r) = w_1x_1 + w_2x_2 + \dots + w_rx_r + b,$$

where  $x_i$  is the variable representing the  $i$ th coordinate of the vector  $\mathbf{x}$ . For convenience, we will use the vector notation from now on.

In essence, SVM finds a hyperplane

$$\langle \mathbf{w} \cdot \mathbf{x} \rangle + b = 0 \tag{37}$$

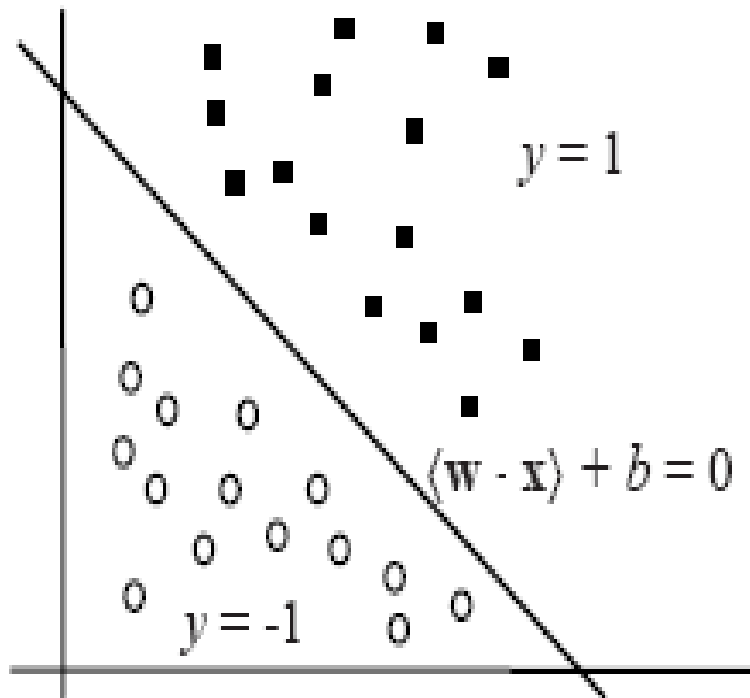
that separates positive and negative training examples. This hyperplane is called the **decision boundary** or **decision surface**.

# Support Vector Machines

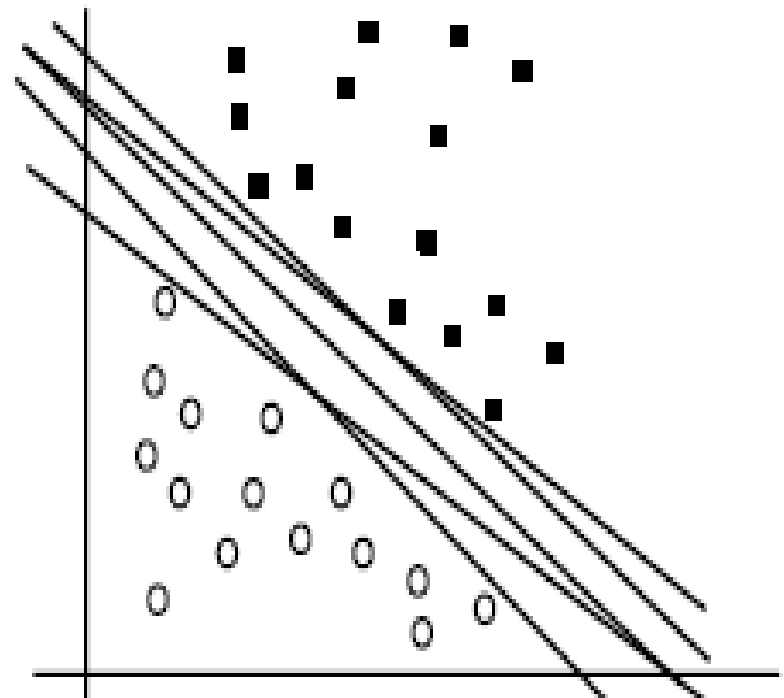


- Geometrically, the hyperplane  $\langle \mathbf{w} \cdot \mathbf{x} \rangle + b = 0$  divides the input space into two half spaces: one half for positive examples and the other half for negative examples.
- A hyperplane is commonly called **a line** in a 2-dimensional space and **a plane** in a 3-dimensional space

# Support Vector Machines



(A)



(B)

Fig. 3.19. (A) A linearly separable data set and (B) possible decision boundaries

# Support Vector Machines



- How can we find the hyperplane
- The distance between the hyperplane and the nearest data point is known as the margin.
- The goal is to choose a hyperplane with a greatest possible margin between hyperplane and any point within the training set

# Support Vector Machines



## Linear SVM: Separable Case

- It is assumed that the positive and negative data points are linearly separable.

## Linear SVM: Non-separable Case

- The training data is almost always noisy, i.e., containing errors due to various reasons.
- In this case positive and negative datas are not separable.
- There is a negative point (circled) in the positive region, and a positive point in the negative region.

# Linear SVM separable case



- Linear SVM: Separable Case
- From linear algebra, we know that in  $\langle \mathbf{w} \cdot \mathbf{x} \rangle + b = 0$ ,
- $\mathbf{w}$  defines a direction perpendicular to the hyperplane .
- $\mathbf{w}$  is also called the **normal vector** (or simply **normal**) of the hyperplane.

# Linear SVM separable case

**Definition (Linear SVM: Separable Case):** Given a set of linearly separable training examples,

$$D = \{(\mathbf{x}_1, y_1), (\mathbf{x}_2, y_2), \dots, (\mathbf{x}_n, y_n)\},$$

learning is to solve the following constrained minimization problem,

$$\begin{aligned} \text{Minimize : } & \frac{\langle \mathbf{w} \cdot \mathbf{w} \rangle}{2} \\ \text{Subject to : } & y_i(\langle \mathbf{w} \cdot \mathbf{x}_i \rangle + b) \geq 1, \quad i = 1, 2, \dots, n \end{aligned} \tag{44}$$

Note that the constraint  $y_i(\langle \mathbf{w} \cdot \mathbf{x}_i \rangle + b) \geq 1, \quad i = 1, 2, \dots, n$  summarizes:

$$\begin{aligned} \langle \mathbf{w} \cdot \mathbf{x}_i \rangle + b &\geq 1 & \text{for } y_i = 1 \\ \langle \mathbf{w} \cdot \mathbf{x}_i \rangle + b &\leq -1 & \text{for } y_i = -1. \end{aligned}$$

# Linear SVM separable case



- Since SVM maximizes the margin between positive and negative data points, let us find the margin.
- Let  $d_+$  (respectively  $d_-$ ) be the shortest distance
- from the separating hyperplane ( $\langle \mathbf{w} \cdot \mathbf{x} \rangle + b = 0$ ) to the closest positive (negative) data point.



# Linear SVM separable case



- The **margin** of the separating hyperplane is
- $d_+ + d_-$ . SVM looks for the separating hyperplane with the largest margin, which is also called the **maximal margin hyperplane**, as the final **decision boundary**

# Linear SVM separable case

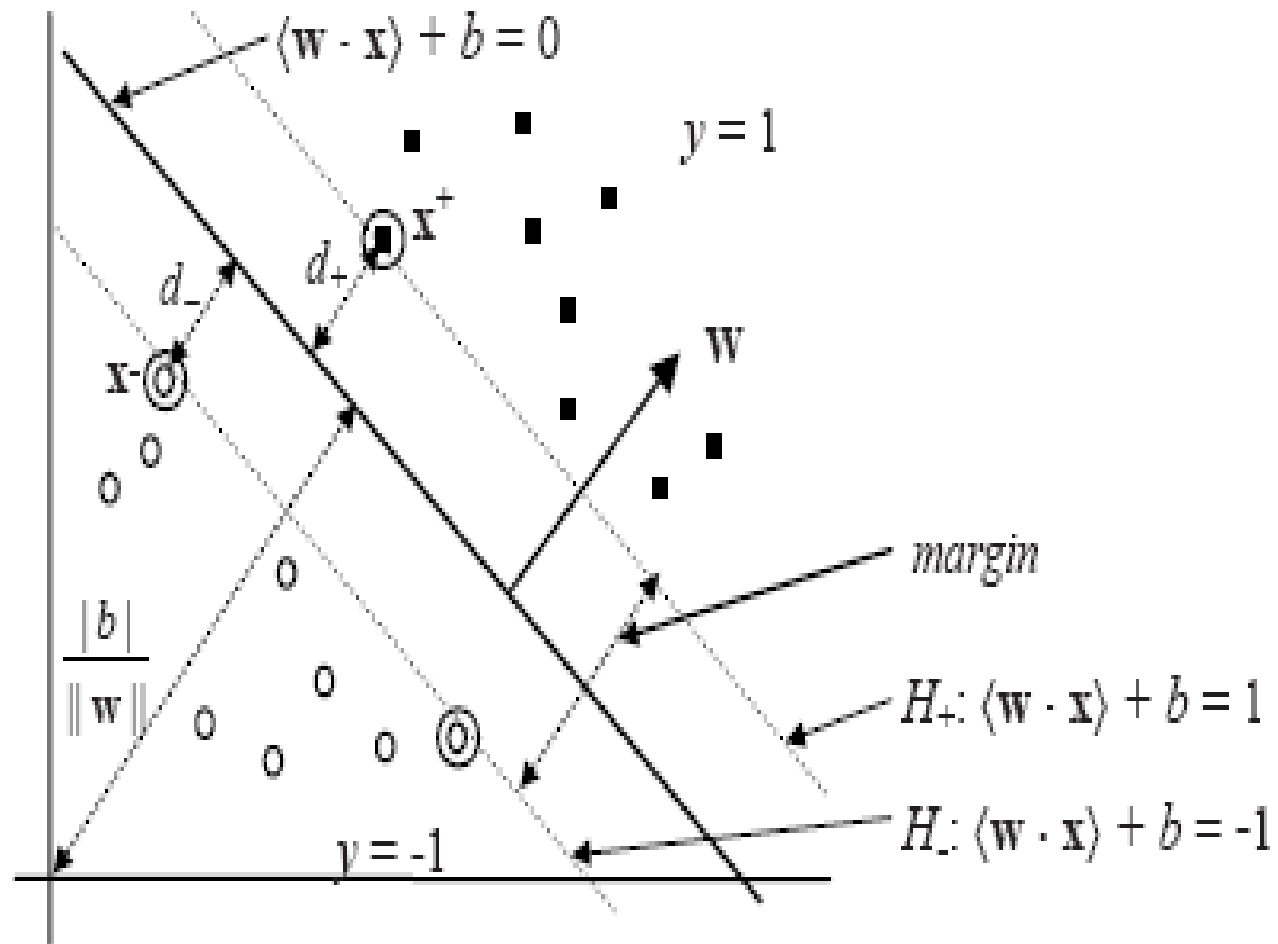


Fig. 3.20. Separating hyperplanes and margin of SVM: Support vectors are circled

## Linear SVM separable case

Let us consider a positive data point  $(\mathbf{x}^+, 1)$  and a negative data point  $(\mathbf{x}^-, -1)$  that are closest to the hyperplane  $\langle \mathbf{w} \cdot \mathbf{x} \rangle + b = 0$ . We define two parallel hyperplanes,  $H_+$  and  $H_-$ , that pass through  $\mathbf{x}^+$  and  $\mathbf{x}^-$  respectively.  $H_+$  and  $H_-$  are also parallel to  $\langle \mathbf{w} \cdot \mathbf{x} \rangle + b = 0$ . We can rescale  $\mathbf{w}$  and  $b$  to obtain

$$H_+: \quad \langle \mathbf{w} \cdot \mathbf{x}^+ \rangle + b = 1 \quad (38)$$

$$H_-: \quad \langle \mathbf{w} \cdot \mathbf{x}^- \rangle + b = -1 \quad (39)$$

such that

$$\begin{aligned} \langle \mathbf{w} \cdot \mathbf{x}_i \rangle + b &\geq 1 && \text{if } y_i = 1 \\ \langle \mathbf{w} \cdot \mathbf{x}_i \rangle + b &\leq -1 && \text{if } y_i = -1, \end{aligned}$$

which indicate that no training data fall between hyperplanes  $H_+$  and  $H_-$ .

# Linear SVM Non separable case

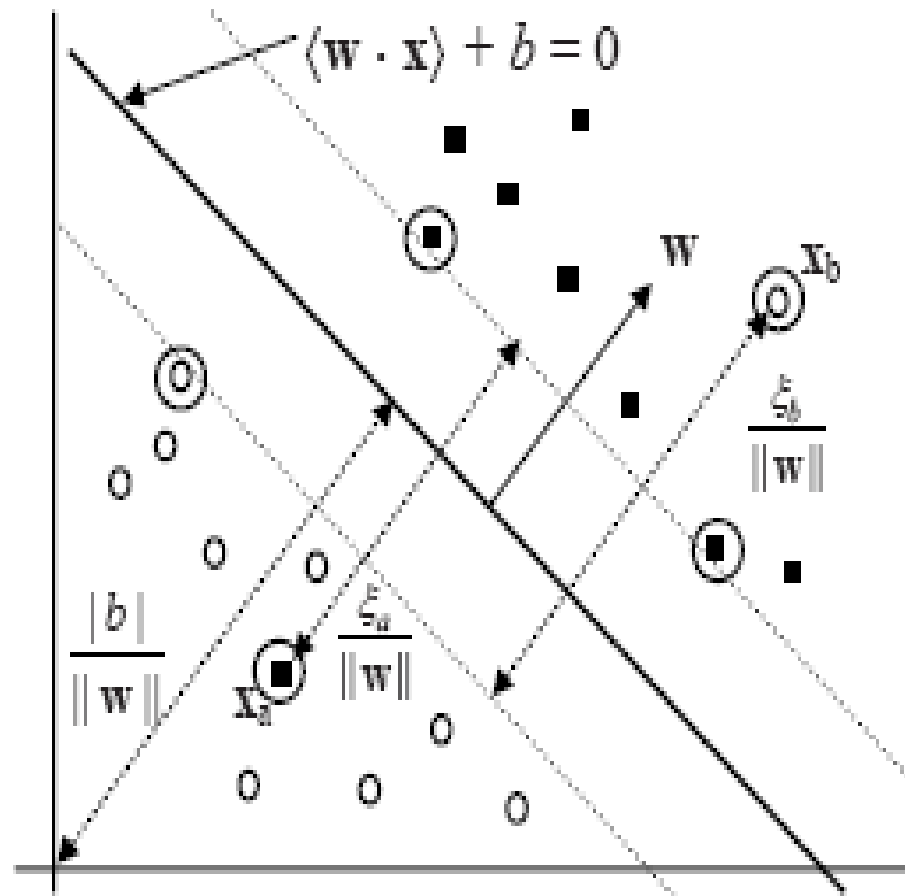


Fig. 3.21. The non-separable case:  $x_a$  and  $x_b$  are error data points

## Linear SVM Non separable case

To allow errors in data, we can relax the margin constraints by introducing **slack** variables,  $\xi_i (\geq 0)$  as follows:

$$\begin{aligned}\langle \mathbf{w} \cdot \mathbf{x}_i \rangle + b &\geq 1 - \xi_i & \text{for } y_i = 1 \\ \langle \mathbf{w} \cdot \mathbf{x}_i \rangle + b &\leq -1 + \xi_i & \text{for } y_i = -1.\end{aligned}$$

Thus we have the new constraints:

$$\begin{aligned}\text{Subject to: } & y_i(\langle \mathbf{w} \cdot \mathbf{x}_i \rangle + b) \geq 1 - \xi_i, i=1, 2, \dots, n, \\ & \xi_i \geq 0, i=1, 2, \dots, n.\end{aligned}$$

The geometric interpretation is shown in [Fig. 3.21](#), which has two error data points  $\mathbf{x}_a$  and  $\mathbf{x}_b$  (circled) in wrong regions.

# Unsupervised Learning



- Supervised learning discovers **patterns in the data** that relate **data attributes to a class attribute**.
- These patterns are then utilized to predict the values of the class attribute of future data instances.
- However, in some applications, the data have no class attributes.
- The user wants to explore the data to find some intrinsic structures in them.

# Unsupervised Learning



- Clustering is one technology for finding such structures.
- It organizes data instances into **similarity groups**, called **clusters**
- The data instances in the same cluster are similar to each other and data instances in different clusters are very different from each other.

# Unsupervised Learning



- Clustering is often called **unsupervised learning**.



# Clustering- Basic Concepts



- **Clustering** is the process of organizing data instances into groups whose members are similar in some way.
- A **cluster** is therefore a collection of data instances which are “similar” to each other and are “dissimilar” to data instances in other clusters.

# Clustering- Basic Concepts



- In the clustering literature, a data instance is also
- called an **object** as the instance may represent an object in the real world.
- It is also called a **data point** as it can be seen as a point in an  $r$  dimension space, where  $r$  is the number of attributes in the data.

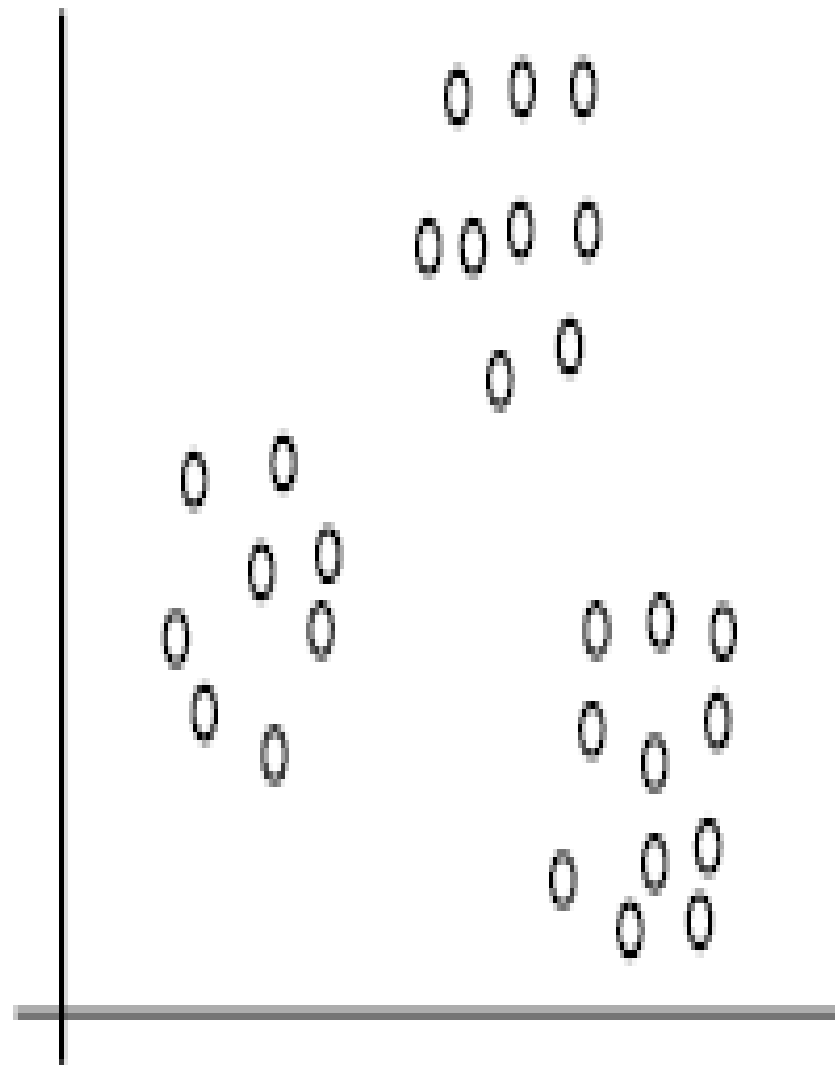


Fig. 4.1. Three natural groups or clusters of data points

# Clustering- Basic Concepts



- Clustering needs a similarity function to measure how similar two data points (or objects) are,
- or a **distance function** to measure the distance between two data points.

# The $k$ -means clustering algorithm



**Algorithm**  $k$ -means( $k, D$ )

- 1 choose  $k$  data points as the initial centroids (cluster centers)
- 2 **repeat**
- 3 **for** each data point  $\mathbf{X} \in D$  **do**
- 4 compute the distance from  $\mathbf{X}$  to each centroid;
- 5 assign  $\mathbf{X}$  to the closest centroid // a centroid represents a cluster
- 6 **endfor**
- 7 re-compute the centroid using the current cluster memberships
- 8 **until** the stopping criterion is met.

# K-means Clustering



- The  $k$ -means algorithm is the best known **partitional clustering algorithm**.
- It is perhaps also the most widely used among all clustering algorithms due to its simplicity and efficiency

# K-means Clustering



- Let the set of data points (or instances)  $D$  be  $\{\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_n\}$ , where  $\mathbf{x}_i = (x_{i1}, x_{i2}, \dots, x_{ir})$  is a vector in a real-valued space  $X \subseteq \mathbb{R}_r$ , and  $r$  is the number of attributes in the data (or the number of dimensions of the **data space**).

# K-means Clustering



- The  $k$ -means algorithm partitions the given data into  $k$  clusters.
- Each cluster has a cluster **center**, which is also called the cluster **centroid**.
- The centroid, usually used to represent the cluster, is simply the mean of all the data points in the cluster, which gives the name to the algorithm, i.e., since there are  $k$  clusters, thus  $k$  means. [Fig.](#) gives the  $k$  means clustering algorithm.



# K-means Clustering



- At the beginning, the algorithm randomly selects  $k$  data points as the **seed** centroids.
- It then computes the distance between each seed centroid and every data point.
- Each data point is assigned to the centroid that is
- closest to it.

# K-means Clustering



- A centroid and its data points therefore represent a cluster.
- Once all the data points in the data are assigned, the centroid for each cluster is re-computed using the data points in the current cluster.
- This process repeats until a stopping criterion is met.

# K-means Clustering




The stopping (or convergence) criterion can be any one of the following:

1. no (or minimum) re-assignments of data points to different clusters.
2. no (or minimum) change of centroids.
3. minimum decrease in the **sum of squared error** (SSE),

# K-means Clustering



$$SSE = \sum_{j=1}^k \sum_{\mathbf{x} \in C_j} dist(\mathbf{x}, \mathbf{m}_j)^2,$$


$$SSE = \sum_{j=1}^k \sum_{\mathbf{x} \in C_j} \text{dist}(\mathbf{x}, \mathbf{m}_j)^2, \quad (1)$$

where  $k$  is the number of required clusters,  $C_j$  is the  $j$ th cluster,  $\mathbf{m}_j$  is the centroid of cluster  $C_j$  (the mean vector of all the data points in  $C_j$ ), and  $\text{dist}(\mathbf{x}, \mathbf{m}_j)$  is the distance between data point  $\mathbf{x}$  and centroid  $\mathbf{m}_j$ .

# K-means Clustering



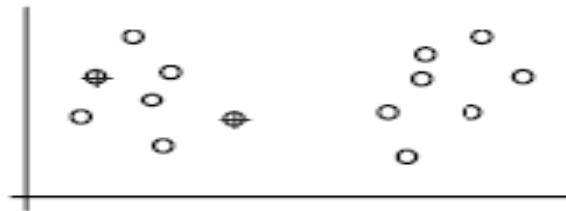
The  $k$ -means algorithm can be used for any application data set where the **mean** can be defined and computed. In **Euclidean space**, the mean of a cluster is computed with:

$$\mathbf{m}_j = \frac{1}{|C_j|} \sum_{\mathbf{x}_i \in C_j} \mathbf{x}_i, \quad (2)$$

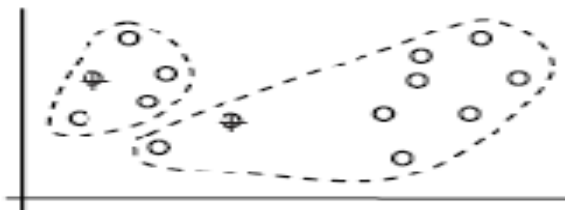
where  $|C_j|$  is the number of data points in cluster  $C_j$ . The distance from a data point  $\mathbf{x}_i$  to a cluster mean (centroid)  $\mathbf{m}_j$  is computed with

$$\begin{aligned} \text{dist}(\mathbf{x}_i, \mathbf{m}_j) &= \|\mathbf{x}_i - \mathbf{m}_j\| \\ &= \sqrt{(x_{i1} - m_{j1})^2 + (x_{i2} - m_{j2})^2 + \dots + (x_{ir} - m_{jr})^2}. \end{aligned} \quad (3)$$

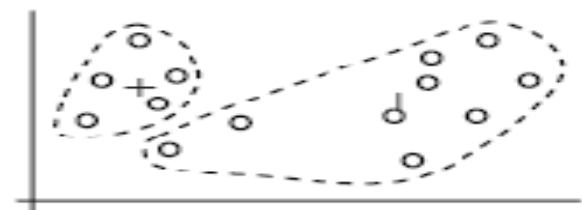
# K-means Clustering



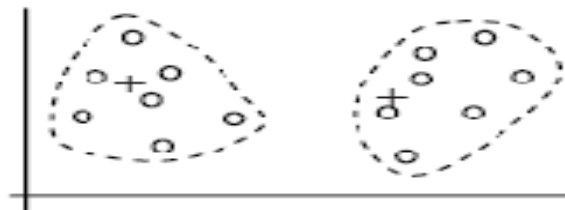
(A). Random selection of  $k$  seeds (or centroids)



Iteration 1: (B). Cluster assignment



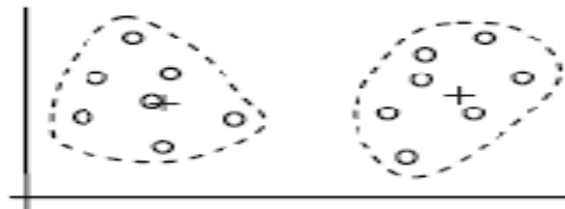
(C). Re-compute centroids



Iteration 2: (D). Cluster assignment



(E). Re-compute centroids



Iteration 3: (F). Cluster assignment



(G). Re-compute centroids

# K-means Clustering



- One problem with the  $k$ -means algorithm is that some clusters may become empty during the clustering process since no data point is assigned to them.
- Such clusters are called **empty clusters**.
- To deal with an empty cluster, we can choose a data point as the replacement centroid, e.g., a data
- point that is furthest from the centroid of a large cluster.
- If the sum of the squared error (SSE) is used as the stopping criterion, the cluster with the largest squared error may be used to find another centroid.



# Disk Version of the K-means Algorithm



- The  $k$ -means algorithm may be implemented in such a way that it does not need to load the entire data set into the main memory, which is useful for large data sets.
- Notice that the centroids for the  $k$  clusters can be computed incrementally in each iteration because the summation can be calculated separately first.

# Disk Version of the K-means Algorithm



- During the clustering process, the number of data points in each cluster can be counted incrementally as well.
- This gives us a disk based implementation of the algorithm , which produces exactly the same clusters but with the data on disk.

# Disk Version of the K-means Algorithm

**Algorithm** disk- $k$ -means( $k, D$ )

```
1  Choose  $k$  data points as the initial centriods  $\mathbf{m}_j, j = 1, \dots, k$ ;
2  repeat
3      initialize  $\mathbf{s}_j \leftarrow \mathbf{0}, j = 1, \dots, k$ ;           //  $\mathbf{0}$  is a vector with all 0's
4      initialize  $n_j \leftarrow 0, j = 1, \dots, k$ ;         //  $n_j$  is the number of points in cluster  $j$ 
5      for each data point  $\mathbf{x} \in D$  do
6           $j \leftarrow \arg \min_{i \in \{1, 2, \dots, k\}} \text{dist}(\mathbf{x}, \mathbf{m}_i)$ ;
7          assign  $\mathbf{x}$  to the cluster  $j$ ;
8           $\mathbf{s}_j \leftarrow \mathbf{s}_j + \mathbf{x}$ ;
9           $n_j \leftarrow n_j + 1$ ;
10     endfor
11      $\mathbf{m}_j \leftarrow \mathbf{s}_j / n_j, j = 1, \dots, k$ ;
12 until the stopping criterion is met
```

# Disk Version of the K-means Algorithm



- Line 3 initializes vector  $\mathbf{s}_j$  which is used to incrementally compute the sum.
- Line 4 initializes  $n_j$  which records the number of data points assigned to cluster  $j$  (line 9).
- Lines 6 and 7 perform exactly the same tasks as lines 4 and 5 in the original algorithm.(k means)
- Line 11 re-computes the centroids, which are used in the next iteration.

# Strengths and weaknesses of K means Clustering



## Strengths

- The main strengths of the  $k$ -means algorithm are its simplicity and efficiency.
- It is easy to understand and easy to implement.
- Its time complexity is  $O(tkn)$ , where  $n$  is the number of data points,  $k$  is the number of clusters, and  $t$  is the number of iterations.
- Since both  $k$  and  $t$  are normally much smaller than  $n$ , the  $k$ -means algorithm is considered a linear algorithm in the number of data points.

# Strengths and weaknesses of K means Clustering



- The weaknesses and ways to address them are as follows:
- The algorithm is only applicable to data sets where the notion of the **mean** is defined.
- Thus, it is difficult to apply to categorical data sets.
- There is, however, a variation of the *k*-means algorithm called **k-modes**, which clusters categorical data.

# Strengths and weaknesses of K means Clustering



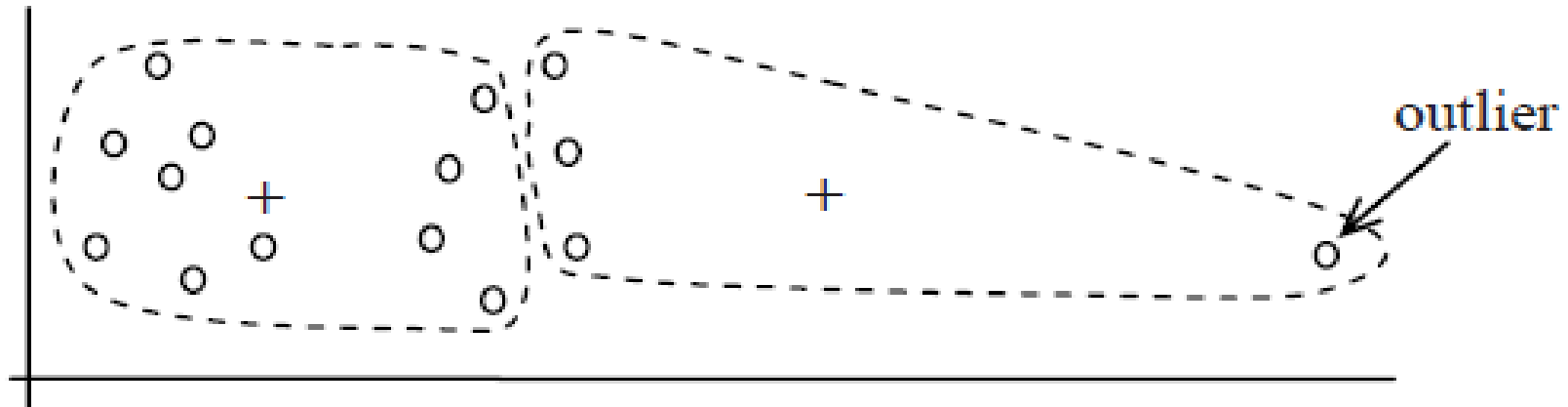
- The user needs to specify the number of clusters  $k$  in advance.
- In practice, several  $k$  values are tried and the one that gives the most desirable result is selected.
- The algorithm is sensitive to **outliers**.
- Outliers are data points that are very far away from other data points.
- Outliers could be errors in the data recording or some special data points with very different values



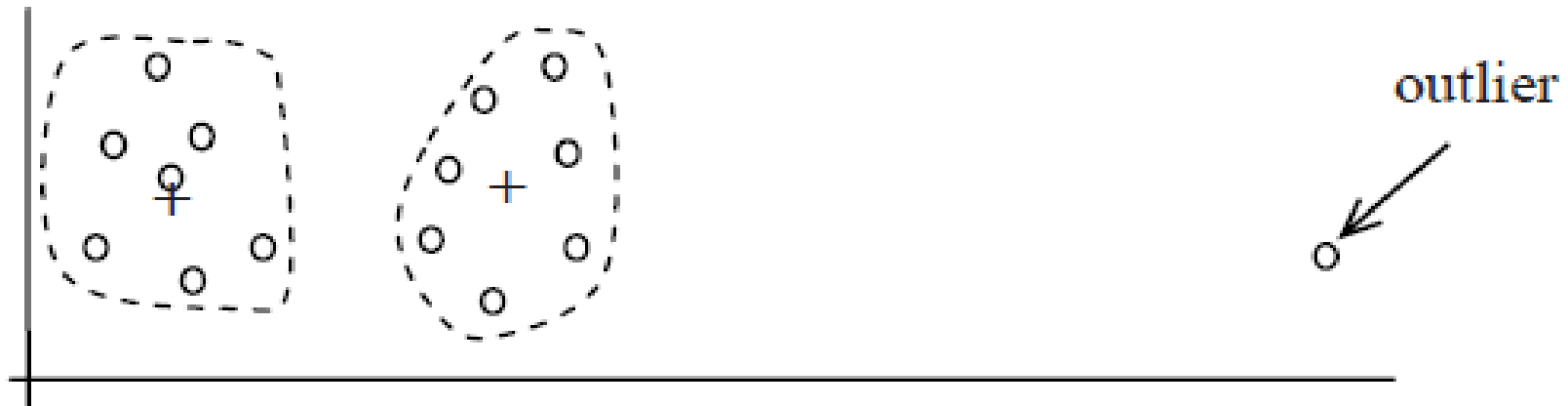
- There are several methods for dealing with outliers.
- One simple method is to remove some data points in the clustering process that are much further away from the centroids than other data points



# Clustering with and without the effect of outliers



(A): Undesirable clusters



(B): Ideal clusters

# Hierarchical Clustering



- Single-Link Method
- Complete-Link Method
- Average-Link Method

# Hierarchical Clustering



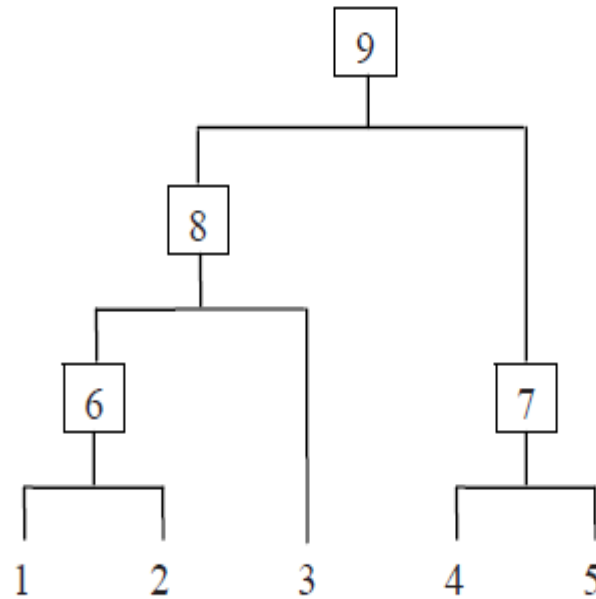
- Hierarchical clustering clusters by producing a nested sequence of clusters like a **tree** (also called a **dendrogram**).
- Singleton clusters (individual data points) are at the bottom of the tree and one root cluster is at the top, which covers all data points.

# Hierarchical Clustering



- Each internal cluster node contains child cluster nodes.
- Sibling clusters partition the data points covered by their common parent.

# An illustration of hierarchical clustering



# Hierarchical Clustering



- At the bottom of the tree, there are 5 clusters (5 data points).
- At the next level, cluster 6 contains data points 1 and 2, and cluster 7 contains data points 4 and 5.
- As we move up the tree, we have fewer and fewer clusters.
- Since the whole clustering tree is stored, the user can choose to view clusters at any level of the tree

# Hierarchical Clustering



There are two main types of hierarchical clustering methods:

- **Agglomerative (bottom up) clustering:** It builds the dendrogram (tree) from the bottom level, and merges the most similar (or nearest) pair of clusters at each level to go one level up.
- The process continues until all the data points are merged into a single cluster (i.e., the root cluster).

# Hierarchical Clustering



- **Divisive (top down) clustering:** It starts with all data points in one cluster, the root.
- It then splits the root into a set of child clusters.
- Each child cluster is recursively divided further until only singleton clusters of individual data points remain, i.e., each cluster with only a single point.



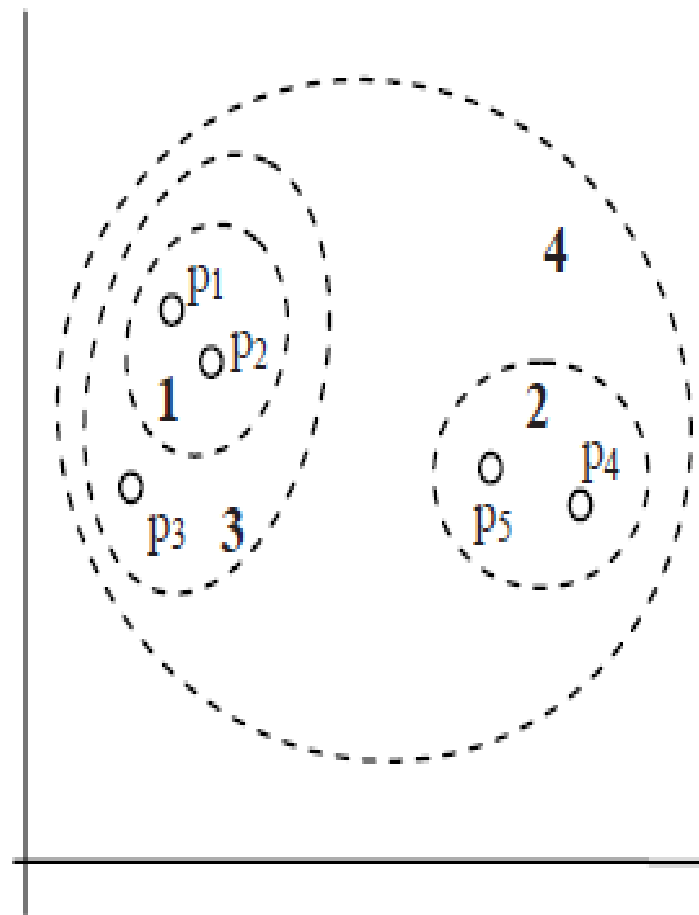
# The general agglomerative algorithm



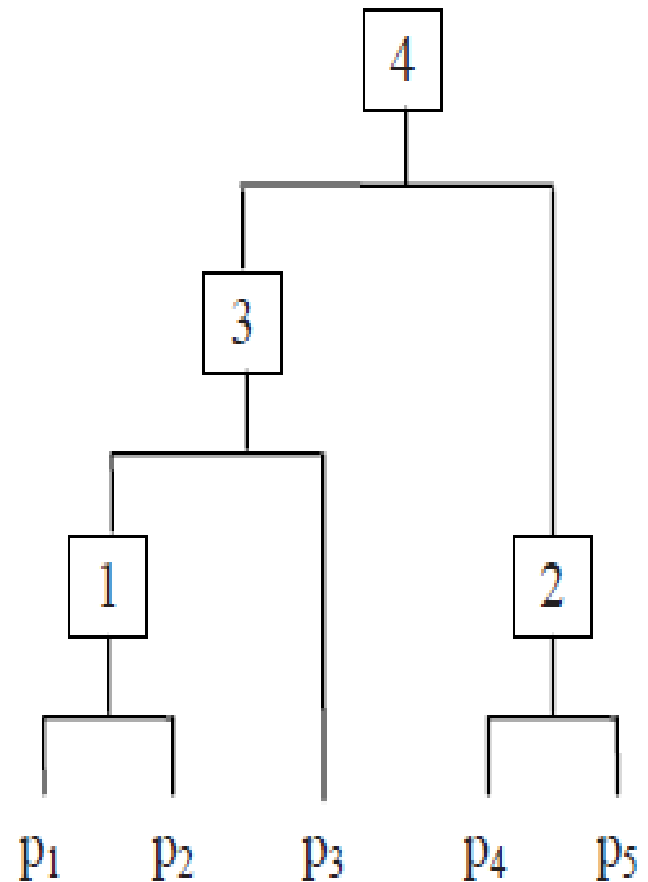
## **Algorithm** Agglomerative( $D$ )

- 1 Make each data point in the data set  $D$  a cluster,
- 2 Compute all pair-wise distances of  $x_1, x_2, \dots, x_n \in D$  ;
- 3 repeat**
- 3 find two clusters that are nearest to each other;
- 4 merge the two clusters form a new cluster  $c$ ;
- 5 compute the distance from  $c$  to all other clusters;
- 12 until** there is only one cluster left.

**Example 9:** Fig. 4.13 illustrates the working of the algorithm. The data points are in a 2-dimensional space. Fig. 4.13(A) shows the sequence of nested clusters, and Fig. 4.13(B) gives the dendrogram. ■



(A). Nested clusters



(B) Dendrogram

# Single-Link Method



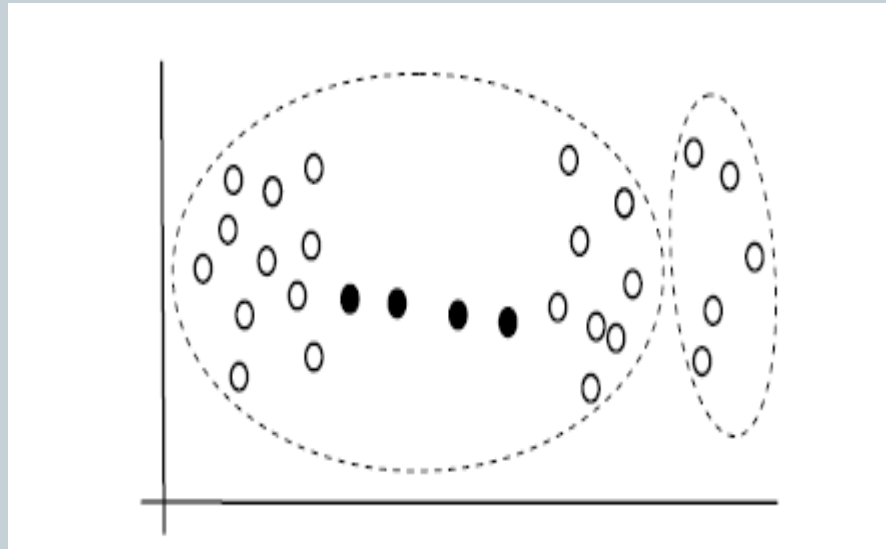
- In **single-link** (or **single linkage**) hierarchical clustering, the distance between two clusters is the distance between two closest data points in the two clusters (one data point from each cluster).
- In other words, the singlelink clustering merges the two clusters in each step whose two nearest data points (or members) have the smallest distance, i.e., the two clusters with the **smallest minimum** pairwise distance.

# Single-Link Method



- The single-link method is suitable for finding non-elliptical shape clusters.
- However, it can be sensitive to noise in the data, which may cause the **chain effect** and produce straggly clusters.

# Single-Link Method

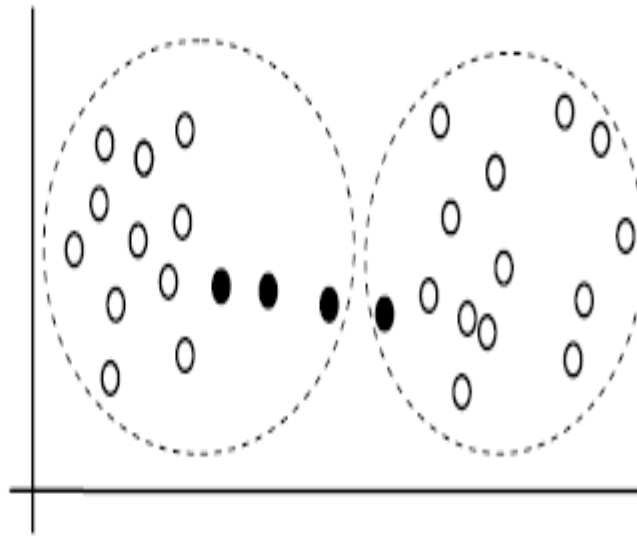


# Complete-Link Method



- In **complete-link** (or **complete linkage**) clustering, the distance between two clusters is the **maximum** of all pair-wise distances between the data points in the two clusters.
- In other words, the complete-link clustering merges the two clusters in each step whose two furthest data points have the smallest distance, i.e., the two clusters with the **smallest maximum** pair-wise distance.

# Complete-Link Method



**Fig. 4.15.** Clustering using the complete-link method

# Average-Link Method



- In this method, the distance between two clusters is the average distance of all pair-wise distances between the data points in two clusters.





- Apart from the above three popular methods, there are several others.
- The following two methods are also commonly used:
- **Centroid method:** In this method, the distance between two clusters is the distance between their centroids.
- **Ward's method:** In this method, the distance between two clusters is defined as the increase in the sum of squared error (distances) from that of two clusters to that of one merged cluster

# Strengths and Weaknesses of Hierarchical Clustering



Hierarchical clustering has several advantages compared to the  $k$ -means and other partitioning clustering methods.

## Advantages

- It is able to take any form of distance or similarity function.
- Moreover, unlike the  $k$ -means algorithm which only gives  $k$  clusters at the end, the hierarchy of clusters from hierarchical clustering enables the user to explore clusters at any level of detail (or granularity).

# Strengths and Weaknesses of Hierarchical Clustering



- Agglomerative hierarchical clustering often produces better clusters than the *k*-means method. It can also find clusters of arbitrary shapes, e.g., using the single-link method

# Strengths and Weaknesses of Hierarchical Clustering



## Weaknesses

- The single-link method may suffer from the chain effect, and the complete-link method is sensitive to outliers.
- The main shortcomings of all hierarchical clustering methods are their computation complexities and space requirements, which are at least quadratic.
- Compared to the  $k$ -means algorithm, this is very inefficient and not practical for large data set